Choosing Between WebRTC and LL-HLS for Low-Latency Streaming: A Controlled Experimental Comparison

Aleksander Adamkowski
Malmö University
Malmö, Sweden
aleks.adamkowski@gmail.com

Djordje Dimitrov
Malmö University
Malmö, Sweden
dimitrovdjordje@gmail.com

Abstract

Ultra-low-latency live video streaming is increasingly important for Internet of Things (IoT) applications such as remote monitoring, teleoperation, and live sports analysis, where end-to-end delay must be minimized while maintaining scalable delivery. In practice, teams must choose between architectures that prioritize ultra-low latency and those that prioritize broadcast scalability, but the trade-offs are often discussed qualitatively rather than validated under comparable experimental conditions.

This paper presents a controlled experimental comparison of Web Real-Time Communication (WebRTC) and Low-Latency HTTP Live Streaming (LL-HLS), two representative architectures with fundamentally different design principles. The goal is to provide empirical evidence that supports architecture selection and deployment planning under latency, audience size, and resource constraints. Both protocols are evaluated using aligned encoding parameters and increasing viewer concurrency. Server-side scalability is analyzed through CPU and memory usage, while client-side Quality of Experience (QoE) is assessed using end-to-end latency and Time to First Frame (TTFF).

Results show that WebRTC consistently achieves sub-second latency and fast startup, but exhibits increasing server-side resource consumption as client count grows. In contrast, LL-HLS scales efficiently in terms of server resources but operates at higher latency and startup delay. These findings quantify the latency–scalability trade-off and provide practical guidance for selecting streaming architectures in latency-sensitive IoT systems.

1 Introduction

Live video streaming is a core building block in many Internet of Things (IoT) systems, including remote surveillance, industrial monitoring, smart-city infrastructure, and live sports analysis. In these settings, video is often consumed not only for passive viewing, but also as input to time-sensitive decisions and workflows—ranging from operator oversight to automated responses—which makes end-to-end delay and startup responsiveness system-level concerns [4, 14].

Two protocol families dominate low-latency live delivery on the web. Segment-based HTTP streaming—such as Low-Latency HLS (LL-HLS)—builds on an encode-once, serve-many model that integrates well with existing HTTP/CDN infrastructure and scales naturally to large audiences [1, 8]. Real-time communication protocols—most notably WebRTC—avoid segment boundaries and can achieve very low delay and fast startup by delivering media as a continuous stream with real-time feedback and congestion control [1, 5, 12, 13]. However, WebRTC typically maintains more per-client transport state (e.g., security context and congestion-control state), which can increase server resource usage as concurrency grows [9].

Although these trade-offs are widely discussed, practical deployment planning is hindered by the fact that protocol comparisons are often conducted under non-aligned configurations (e.g., different encoder settings, players, buffering targets, or network environments). In addition, prior work emphasizes that segment-based low-latency streaming faces structural latency sources related to segment production and publication, which makes sub-second operation difficult to sustain without compromising robustness [1–3, 10].

This paper presents a controlled experimental comparison of WebRTC and LL-HLS under aligned encoding parameters and a shared local network environment. We evaluate both protocols across client-side Quality of Experience (QoE) and server-side scalability dimensions. On the client side, we measure end-to-end latency and Time to First Frame (TTFF). On the server side, we quantify CPU utilization, memory consumption as the number of concurrent viewers increases. This framing reflects real-world architectural decision-making: a protocol that performs well for a single viewer may become costly at scale, while a protocol that scales efficiently may introduce responsiveness overhead that is unacceptable for interactive use cases.

Contributions. This paper provides (1) a benchmark methodology for WebRTC and LL-HLS with aligned encoding settings; (2) a combined evaluation of scalability (CPU and memory) and QoE (latency, TTFF) under increasing load; and (3) practical guidance for selecting between WebRTC and LL-HLS depending on audience size, responsiveness requirements, and infrastructure constraints.

2 Background and Foundations

2.1 Live Video Streaming Fundamentals

Live video streaming refers to the real-time delivery of video content from a source to one or more viewers over a packet-switched network. Unlike video on demand (VoD), where content is pre-recorded and distributed ahead of playback, live streaming must minimize the delay between capture and rendering to preserve temporal relevance. Prior work commonly models live streaming as an end-to-end pipeline comprising video capture, encoding, transport, distribution, and playback [4, 14].

Key design concerns in live video streaming include end-to-end latency, scalability, robustness to network variation, and Quality of Experience (QoE). Latency is commonly defined as the glass-to-glass delay from capture to display (Figure 1). QoE is assessed using metrics such as startup delay and delivered video quality [1, 4].

Live streaming systems differ primarily in whether media is delivered as discrete HTTP segments or as continuous real-time

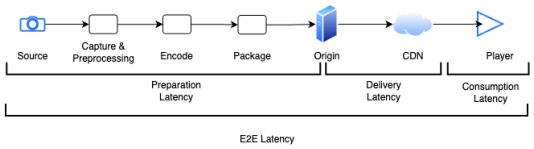

Figure 1: Sources of end-to-end latency in a live video streaming system, spanning capture, processing, transport, and playback. Adapted from [1].

streams, a distinction that fundamentally affects latency, scalability, and system complexity.

2.2 Low-Latency HTTP Streaming (LL-HLS)

HTTP Live Streaming (HLS) is an adaptive bitrate streaming protocol that delivers media over standard HTTP infrastructure by dividing content into a sequence of media segments referenced through playlist files. Originally developed by Apple, HLS has become one of the most widely deployed streaming protocols due to its broad device support, compatibility with Content Delivery Networks (CDNs), and operational simplicity [8].

Low-Latency HLS (LL-HLS) extends the traditional HLS model with mechanisms that reduce delivery delay by enabling finer-grained and earlier media delivery. Clients can request and receive partial media data while content is still being encoded, reducing time-to-playback without changing the fundamental HTTP-based delivery model.

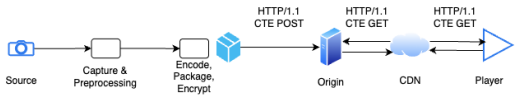

Figure 2: Illustration of the basic HTTP request–response model, where clients retrieve resources using HTTP `GET` requests and servers respond with requested content.

The LL-HLS architecture preserves the encode-once, serve-many paradigm of traditional HLS. Once media segments or partial segments are produced, they can be cached and distributed to multiple clients without maintaining per-viewer transport state. This design enables efficient reuse of encoded content across the delivery infrastructure and supports scalable live distribution.

2.3 WebRTC

Web Real-Time Communication (WebRTC) is an open standard that enables real-time transmission of audio, video, and arbitrary data between endpoints. Originally designed for interactive applications such as video conferencing and voice communication, WebRTC integrates media capture, encoding, and transport using RTP over UDP, together with congestion control, jitter buffering, and real-time feedback mechanisms standardized by the IETF [12, 13].

Unlike segment-based streaming protocols, WebRTC delivers media as a continuous stream of audio and video frames without relying on playlists or discrete media segments. Session establishment is performed through an external signaling mechanism, typically

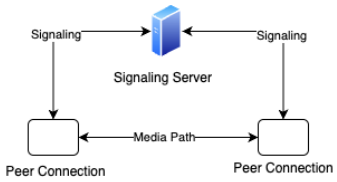

Figure 3: Simplified WebRTC architecture showing signaling-based session establishment and direct peer-to-peer media delivery.

implemented using HTTP or WebSocket, after which media flows directly between endpoints over encrypted RTP channels, as illustrated in Figure 3.

To support real-time delivery, WebRTC maintains per-connection transport state for each participating endpoint, including encryption context, congestion control state, jitter buffering, packet reordering, and network adaptation parameters. This stateful design enables fine-grained adaptation to individual network conditions and supports bidirectional media flows required for interactive communication.

WebRTC further incorporates built-in mechanisms for bandwidth estimation, packet loss recovery, and dynamic bitrate adjustment based on real-time feedback. Together, these features allow WebRTC to maintain low and stable latency under varying network conditions, making it well suited for time-critical and interactive media delivery scenarios.

2.4 Scalability in Live Video Streaming Systems

Scalability in live video streaming systems refers to the ability to support increasing numbers of concurrent viewers without proportional growth in resource consumption or degradation in Quality of Experience. Scalability is strongly influenced by architectural choices and the underlying delivery protocol.

Segment-based streaming protocols such as LL-HLS naturally support scalability through an encode-once, serve-many model. Once media segments or partial segments are generated, they can be reused across clients and efficiently distributed through HTTP servers or CDNs with minimal per-viewer state. This architecture enables large-scale broadcast deployments with predictable server-side resource usage [1, 3].

In contrast, real-time communication protocols such as WebRTC originated from peer-to-peer communication paradigms, where each client maintains a separate session. Without additional forwarding infrastructure, this per-client state can lead to linear growth in server-side resource consumption as audience size increases [9]. To mitigate these limitations, hybrid architectures and forwarding mechanisms are commonly employed to share media streams across clients while preserving low-latency delivery characteristics.

Understanding these scalability trade-offs is essential when selecting streaming architectures for specific application requirements, particularly when balancing ultra-low latency against resource efficiency, operational complexity, and audience size.

2.5 Quality of Experience (QoE)

Quality of Experience (QoE) describes the overall perceived quality of a streaming service from the viewer's perspective and reflects how technical system behavior translates into user satisfaction. In live video streaming, QoE is primarily influenced by temporal aspects of delivery, including startup delay and end-to-end latency, as well as delivered video quality [1].

Startup delay refers to the time between a viewer initiating playback and the rendering of the first video frame, while end-to-end latency represents the time difference between media capture and playback of the same content at the client. These metrics are distinct: a stream may start quickly but still exhibit high latency, or vice versa. Unlike startup delay, which affects only the beginning of playback, end-to-end latency persists throughout the session and may vary over time depending on buffering and delivery mechanisms [1].

QoE differs from Quality of Service (QoS), which focuses on network- and system-level metrics such as throughput, packet loss, or jitter. Instead, QoE captures how such technical properties impact the viewer's perceived experience, particularly in latency-sensitive live streaming scenarios. Prior work highlights a fundamental trade-off between buffering and latency: increasing buffer depth improves robustness, but at the cost of higher end-to-end delay, while aggressive latency reduction reduces buffering slack and increases sensitivity to timing variations [1].

In this work, QoE is evaluated using objective metrics that are widely accepted as practical proxies for user experience in systems-oriented streaming research. The selected metrics—end-to-end latency and Time to First Frame (TTFF)—capture key QoE factors emphasized in prior low-latency streaming studies and are well suited for controlled, repeatable evaluation at scale [1]. These metrics also align with the requirements of latency-sensitive IoT and live sports streaming applications, where immediacy and fast startup are critical.

3 Project Statement and Research Questions

Ultra-low-latency live video streaming remains a fundamental challenge for modern IoT applications. While multiple streaming architectures aim to reduce end-to-end delay, it remains unclear which approaches best balance latency, scalability and startup performance under realistic system constraints.

This project focuses on an experimental comparison of two representative low-latency streaming architectures: WebRTC and Low-Latency HLS (LL-HLS). These protocols embody fundamentally different design philosophies—continuous real-time delivery versus segment-based HTTP streaming—and are commonly considered for latency-sensitive applications with diverging scalability requirements.

The primary objective of this work is to evaluate how WebRTC and LL-HLS behave under controlled conditions as the number of concurrent viewers increases. The evaluation focuses on end-to-end latency, startup delay (Time to First Frame) and server-side resource utilization, with particular emphasis on scalability characteristics relevant to IoT-oriented live video systems.

To ensure comparability, both architectures are implemented using aligned encoding parameters and evaluated in a controlled local network environment.

The project is guided by the following questions:

(1) **RQ1**: How do WebRTC and LL-HLS differ in server-side scalability characteristics as the number of concurrent viewers increases?
(2) **RQ2**: How do WebRTC and LL-HLS compare in terms of client-side Quality of Experience under increasing viewer concurrency?

4 Literature Review

4.1 Low-Latency Requirements in Live Streaming

Ultra-low-latency live video streaming has become increasingly important in applications such as remote surveillance, industrial monitoring, teleoperation, and live sports production. In these scenarios, timely delivery of visual information is critical for enabling real-time interaction, responsive system behavior, and synchronized decision-making. As a result, end-to-end latency requirements are often below one second, which is substantially stricter than those of traditional broadcast-oriented streaming systems.

Reducing latency in live streaming systems introduces fundamental trade-offs between responsiveness, playback stability, and scalability. Aggressively minimizing buffering and startup delay improves perceived immediacy but increases sensitivity to network jitter and timing variations, potentially leading to playback interruptions. These trade-offs motivate architectural approaches that address latency at the system level rather than through buffering alone [1, 15].

In addition to steady-state latency, startup performance plays a key role in perceived responsiveness. Time to First Frame (TTFF) captures how quickly playback begins after initiation, while end-to-end latency reflects how closely the displayed content aligns with real-world events. These metrics are influenced by protocol design choices such as connection setup, buffering strategy, and media delivery granularity, motivating systematic evaluation of streaming architectures under increasing load [1].

4.2 Segment-Based HTTP Streaming Performance

Segment-based HTTP streaming protocols form the foundation of most large-scale live video delivery systems. Protocols such as Low-Latency HLS (LL-HLS) and Low-Latency DASH (LL-DASH) enable efficient one-to-many distribution through compatibility with existing web infrastructure and Content Delivery Networks (CDNs), making them the dominant choice for broadcast-oriented applications.

However, empirical studies consistently identify fundamental latency limitations. Bentaleb et al. [2] demonstrated that even with HTTP chunked transfer encoding, segment-based delivery cannot eliminate serialization delays introduced by the media production pipeline. Their measurements showed that encoders must generate media sequentially, creating an unavoidable latency floor regardless of network conditions.

LL-HLS and LL-DASH introduce optimizations including partial segments, preload hints, and aggressive playlist updates. Despite these enhancements, achieving sub-second latency remains challenging. Bentaleb et al. [3] evaluated multiple adaptive bitrate strategies across LL-HLS and LL-DASH implementations, consistently observing end-to-end delays of two to three seconds under optimized conditions.

Li et al. [10] analyzed systemic bottlenecks in real-time Internet video pipelines and identified segment generation and playlist propagation as persistent sources of delay. Their findings reinforce that incremental optimizations to segment-based protocols face diminishing returns due to the fundamental encode–publish–fetch cycle.

A comprehensive survey by Bentaleb et al. [1] concluded that achieving one-second end-to-end latency requires architectural changes rather than parameter tuning within existing segment-based frameworks. While LL-HLS provides strong scalability through stateless, reusable content distribution, its segmentation paradigm inherently conflicts with ultra-low-latency objectives.

4.3 WebRTC for Low-Latency Streaming

Building on its real-time communication capabilities, multiple studies have evaluated WebRTC as a transport mechanism for low-latency live video streaming. Its continuous media delivery model and minimal buffering requirements position it as a strong candidate for applications where immediacy and temporal consistency are critical.

Francisco et al. [5] claimed WebRTC-based streaming pipelines can demonstrated sub-second to low-second end-to-end latency under favourable network conditions. By eliminating segment boundaries and reducing initial buffering enables WebRTC has significantly lower latency than LL-HLS or LL-DASH in comparable scenarios. The paper highlighted WebRTC's suitability for latency-critical applications where close alignment between capture and playback is required.

However, scalability remains a documented limitation. Jansen et al. [9] performed an extensive performance analysis of WebRTC server implementations and measured CPU and memory consumption under increasing numbers of concurrent connections. Their findings revealed that per-connection state—including encryption context, congestion control, and individual jitter buffers—leads to linear or super-linear growth in server-side resource usage as audience size increases. The authors concluded that naïve one-to-many WebRTC deployments scale poorly compared to stateless HTTP-based streaming architectures.

To address these limitations, forwarding-based designs have been proposed. Grozev et al. [7] examined Selective Forwarding Unit (SFU) architectures that relay media streams to multiple receivers without full transcoding. Their work demonstrated that SFUs can significantly improve scalability by reducing redundant processing while preserving WebRTC's low-latency characteristics, albeit at the cost of increased architectural and operational complexity.

Collectively, prior work establishes that WebRTC achieves substantially lower end-to-end latency than segment-based HTTP streaming, but introduces scalability challenges that necessitate additional infrastructure or architectural patterns when serving larger audiences.

4.4 Research Gap and Motivation

Prior research on low-latency live video streaming has extensively analyzed both segment-based HTTP streaming and real-time communication protocols, establishing their fundamentally different architectural trade-offs. Studies on LL-HLS and LL-DASH consistently demonstrate strong scalability through stateless content delivery, but identify an inherent latency floor of two to three seconds caused by segment-based media production [1–3]. Conversely, research on WebRTC shows that continuous media delivery enables sub-second end-to-end latency, making it well-suited for interactive and time-critical applications [5]. However, empirical evaluations also reveal scalability concerns due to per-connection state and resource overhead [9].

While SFU-based WebRTC architectures have been proposed to mitigate these scalability limitations, existing work primarily focuses on conferencing scenarios with bidirectional media flows or provides qualitative architectural analysis without systematic performance evaluation [6, 7]. As a result, several critical questions remain unresolved: How do LL-HLS and WebRTC compare under controlled conditions with aligned encoding parameters and identical network environments? At what viewer concurrency levels do architectural trade-offs become operationally significant? How do protocol design choices simultaneously affect both server-side resource utilization and client-side Quality of Experience as load increases?

These questions are particularly relevant for emerging IoT-oriented live streaming applications where both sub-second latency and cost-effective scalability are essential requirements. In these domains, architectural decisions directly impact system responsiveness, infrastructure costs, and deployment feasibility. Existing literature provides valuable insights into individual protocol characteristics but lacks comprehensive side-by-side evaluation that captures practical trade-offs across the full spectrum from QoE metrics (latency, startup time) to resource efficiency (CPU, memory) under realistic load conditions.

This work addresses these gaps through a controlled experimental comparison of WebRTC and LL-HLS under systematically increasing client concurrency. The findings aim to inform architectural decisions for latency-sensitive streaming systems where both immediacy and resource efficiency are critical design constraints.

5 Research Method

5.1 Research Approach

This work follows a quantitative experimental benchmarking approach to evaluate and compare the performance characteristics of two live video streaming protocols: WebRTC and Low-Latency HLS (LL-HLS). The objective is to analyze how protocol design choices affect scalability, latency and startup behavior under controlled conditions.

Rather than optimizing either protocol, the study focuses on comparative evaluation under aligned encoding parameters, network conditions, and workload patterns. By systematically varying the number of concurrent clients while keeping all other variables

constant, the experiment aims to isolate protocol-level behavior and assess trade-offs relevant to real-world streaming system design.

This approach is suitable for systems research, where the goal is to understand performance trends and architectural implications rather than to derive analytical models or protocol proofs.

5.2 Experimental Design

The experimental design follows a controlled single-factor scaling experiment. The primary independent variable is the number of concurrent clients connected to the streaming server. All other parameters—including video resolution, frame rate, bitrate, encoding configuration, network environment, and client implementation—are kept constant over runs.

Two streaming pipelines are evaluated independently:

- A WebRTC-based live streaming pipeline with per-client peer connections.
- An LL-HLS-based pipeline using an encode-once, serve-many delivery model.

For each protocol, experiments are conducted across increasing client counts. The experiments are performed in a controlled local network environment to minimize external variability and ensure that observed differences are attributable to protocol behavior rather than network fluctuations.

5.3 Metrics and Measurement Methods

The evaluation considers both server-side scalability metrics and client-side Quality of Experience (QoE) metrics. Server-side measurements capture the computational cost of serving an increasing number of clients and include system-level CPU and memory utilization as well as process-level CPU usage and resident set size (RSS). Client-side measurements capture perceived performance from the viewer's perspective and include end-to-end latency and Time to First Frame (TTFF). Latency and startup metrics are measured using protocol-specific timestamping mechanisms, as detailed in the following sections.

For each experiment, the number of concurrent clients is fixed before measurements begin. Once all clients have successfully connected and reached a steady playback state, metrics are collected over a 30-second window to reflect representative steady-state behavior and limit the impact of short-lived transients. During this window, both client- and server-side metrics are sampled once per second and summarized using the mean over the collected samples, enabling direct comparison between WebRTC and LL-HLS under increasing load.

5.4 Experimental Setup – Server

5.4.1 Testbed and Hardware. A USB webcam served as the live video source and was directly connected to the server in the evaluation environment. The server was an Intel NUC 11 i3TNK equipped with an Intel Core i3 processor and 8 GB of RAM, running Ubuntu Linux and connected via Ethernet. This setup ensured stable local network conditions while enabling controlled multi-client load testing without introducing WAN variability.

5.4.2 Capture and Encoding Settings. The camera stream was captured at a resolution of 640×360 pixels and a frame rate of 30 frames per second, using a constant target bitrate of 1000 kbps. A low-latency H.264 encoding profile with a short Group of Pictures (GOP) structure was used to minimize encoder delay. These parameters were selected to balance visual quality and bandwidth usage while keeping end-to-end latency low and consistent across both protocols.

5.4.3 WebRTC Implementation. The WebRTC pipeline was implemented using a Python-based WebRTC library providing signaling, RTP/SRTP transport, and peer connection management. Signaling was handled via an HTTP server combined with Socket.IO. On the client side, a browser-based JavaScript application creates a receive-only RTCPeerConnection and sends an Session Description Protocol (SDP) offer to the server via Socket.IO. The server responds with an SDP answer and exchanges Interactive Connectivity Establishment (ICE) candidates before media transmission begins.

Each connected client maintains an independent peer connection, implying per-client RTP senders, SRTP encryption, congestion control, pacing, and RTCP feedback. This design reflects a typical one-to-one WebRTC deployment model and highlights the per-client resource costs associated with WebRTC streaming.

5.4.4 LL-HLS Implementation. The LL-HLS pipeline is built around a Python library that generates low-latency HLS playlists and media segments from an incoming MPEG-TS stream. A single FFmpeg instance encodes the webcam stream and feeds it into the LL-HLS handler, which produces a rolling window of segments and partial segments. Clients retrieve playlists and media via standard HTTP requests.

This architecture follows an encode-once, serve-many paradigm, where a single encoded output is shared across all clients. As a result, server scalability is primarily governed by HTTP I/O and network throughput rather than per-client encoding or transport state.

5.4.5 Timestamp Injection and Latency Support. For LL-HLS, the server records the wall-clock timestamp at which each segment or partial segment is published and exposes this information via a lightweight JSON endpoint. Clients retrieve this mapping and compute latency by comparing the playback time of a segment to its recorded publication time. For WebRTC, timestamping is integrated directly into the media pipeline. The server attaches per-frame send timestamps and transmits them to the client via a data channel. Clients perform a time-synchronization handshake to estimate clock offset and compute end-to-end latency by matching rendered frames with their corresponding send timestamps.

5.4.6 Server-side CPU and Memory Measurements. Server resource usage is measured using a `psutil`-based monitoring script that samples both system-level and process-level metrics for the active streaming server process (LL-HLS or WebRTC) during each test run. Each experiment is executed with a predefined target number of concurrent clients. The monitor waits until the server reports that all expected clients have successfully connected, and then begins sampling CPU and memory utilization. Measurements are collected for the predefined 30-second window to capture steady-state load under the target concurrency level.

5.5 Experimental Setup – Client

Prior to each experiment, both the server and client machines were synchronized using Network Time Protocol (NTP) to ensure consistent wall-clock timestamps. This minimizes clock drift and improves the accuracy of time-based latency measurements.

5.5.1 Client-Side Testbed and Load Generation. All client-side experiments were executed on a MacBook running Node.js. A benchmark driver launches multiple headless Chromium instances using Puppeteer, where each browser instance represents one independent viewer. Each instance loads the protocol-specific playback page and executes a fully functional browser playback pipeline.

This approach isolates decoding, buffering, and rendering behavior per viewer while enabling scalable and repeatable load generation. Unless otherwise stated, one headless browser instance corresponds to one client.

5.5.2 Latency Measurement. For LL-HLS, end-to-end latency is computed by matching the playback time of a media segment with its server-side publication timestamp. For each segment s, the latency is calculated as:

$$L_{\text{LL-HLS}}(s) = t_{\text{playback}}(s) - t_{\text{publish}}(s) \tag{1}$$

where $t_{\text{publish}}(s)$ denotes the wall-clock time at which segment s is published by the server, and $t_{\text{playback}}(s)$ denotes the wall-clock time at which playback of segment s begins on the client. This formulation yields an estimate of the end-to-end delay from media production to client playback.

For WebRTC, latency is measured at the granularity of individual video frames using per-frame timestamps transmitted via a data channel. Each frame f is assigned a send timestamp $t_{\text{send}}(f)$ at the server. The client maintains an estimate of the clock offset Δ between server and client clocks through periodic synchronization exchanges. End-to-end latency is then computed as:

$$L_{\text{WebRTC}}(f) = t_{\text{render}}(f) - (t_{\text{send}}(f) - \Delta) \tag{2}$$

where $t_{\text{render}}(f)$ denotes the wall-clock time at which frame f is rendered on the client. This approach provides a latency estimate under stable clock synchronization.

5.5.3 Time to First Frame. Time to First Frame (TTFF) is defined as the elapsed time between player initialization and the rendering of the first video frame. For a given client c, TTFF is computed as:

$$\text{TTFF}(c) = t_{\text{first-frame}}(c) - t_{\text{start}}(c) \tag{3}$$

For LL-HLS, t_{start} corresponds to the time at which the playback page is loaded and the HLS player is initialized. For WebRTC, t_{start} corresponds to the initiation of the peer connection setup, including signaling and offer–answer exchange. The timestamp $t_{\text{first-frame}}$ denotes the wall-clock time at which the first video frame is rendered.

TTFF captures startup latency and protocol-specific initialization overhead and is used to compare startup behavior between WebRTC and LL-HLS under increasing client concurrency.

5.6 Reproducibility

All experimental code, configuration files, and benchmarking scripts used in this study are publicly available to support reproducibility and further investigation. The complete implementation is released as an open-source repository at GitHub.

5.7 Ethical Considerations

This work does not involve human subjects, personal data, or user-identifiable information. All experiments are conducted using synthetic workloads generated through automated clients and locally captured video content. No real users, customer data, or production systems are involved.

The experimental setup and evaluation procedures are designed to be reproducible and transparent, and no deceptive or intrusive techniques are employed.

6 Experimental evaluation

6.1 Server-Side Results

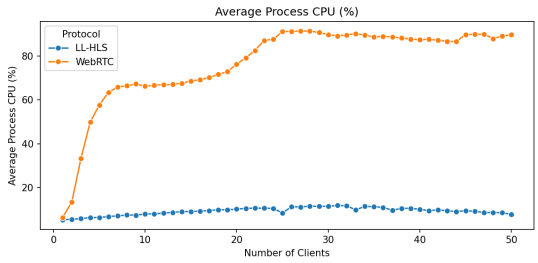

Figure 4: Average process CPU utilization as a function of the number of concurrent clients for WebRTC and LL-HLS.

WebRTC CPU utilization increases sharply with concurrency, while LL-HLS remains largely constant across the tested client range (Figure 4). This indicates that WebRTC becomes compute-bound as audience size grows, whereas LL-HLS imposes minimal incremental CPU cost per additional viewer.

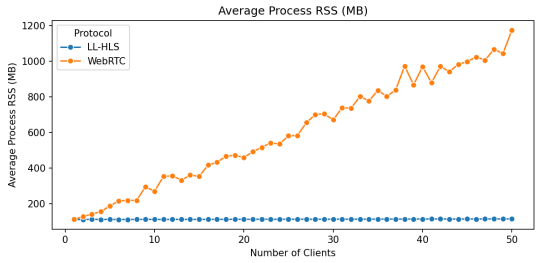

Figure 5: Average process resident set size (RSS) as a function of the number of concurrent clients for WebRTC and LL-HLS.

WebRTC memory consumption increases steadily with concurrency, consistent with per-connection state and buffering overhead (Figure 5). In contrast, LL-HLS maintains a largely constant memory

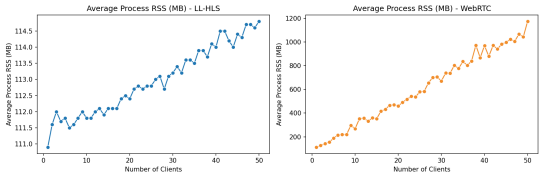

Figure 6: Average process resident set size (RSS) aggregated by protocol across all tested concurrency levels.

footprint, indicating minimal incremental server-side memory cost per additional viewer.

Figure 6 clarifies the magnitude of the difference: WebRTC exhibits an order-of-magnitude increase in RSS with concurrency, whereas LL-HLS remains nearly constant (approximately 111–114.5 MB).

Overall, WebRTC incurs substantially higher CPU and memory overhead as concurrency increases, whereas LL-HLS remains comparatively stable and resource-efficient.

6.2 Client-Side Results

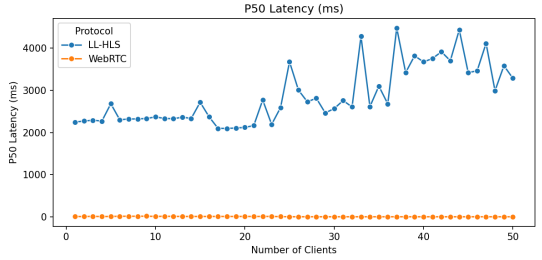

Figure 7: Median end-to-end latency (p50) as a function of the number of concurrent clients for WebRTC and LL-HLS.

WebRTC maintains a stable, sub-second median latency across all tested concurrency levels, while LL-HLS remains in the multi-second range with a mild increase as client count grows (Figure 7). This gap reflects a fundamental design trade-off: WebRTC favors immediacy for interactive use cases, whereas LL-HLS prioritizes scalable, robust HTTP delivery.

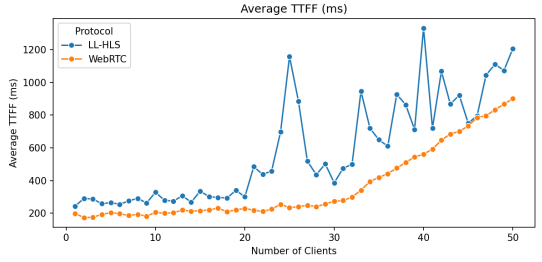

Figure 8: Average time to first frame (TTFF) as a function of the number of concurrent clients for WebRTC and LL-HLS.

LL-HLS exhibits consistently higher TTFF than WebRTC by approximately 200 ms across all tested concurrency levels (Figure 8). Both protocols show a modest TTFF increase as concurrency grows, suggesting a minor startup impact under load.

Taken together, the client-side results highlight a clear trade-off between the protocols. WebRTC provides lower end-to-end latency and faster startup, making it well suited for interactive and ultra-low-latency scenarios. Although, LL-HLS remains viable for use cases where multi-second latency is acceptable and large-scale delivery and operational simplicity are prioritized.

7 Discussion

7.1 Scalability Characteristics

7.1.1 Server-Side Scalability of WebRTC. The server-side results demonstrate that WebRTC exhibits limited scalability when deployed on a single server under increasing client concurrency. As the number of connected viewers grows, both CPU utilization and memory consumption increase substantially, eventually approaching resource saturation. This behavior is consistent across process-level and system-level metrics.

The primary reason for this scaling behavior lies in WebRTC's connection-oriented design. Each client establishes an independent peer connection that requires its own RTP sender, SRTP encryption context, congestion control and pacing logic, and RTCP feedback processing. In addition, per-client buffering and transport state must be maintained throughout the lifetime of the connection. As a result, server-side computation and memory usage increase approximately linearly with the number of concurrent viewers.

While this design enables fine-grained congestion control and supports interactive features such as low-latency delivery and bidirectional communication, it also imposes significant per-client overhead. In practical terms, the results indicate that WebRTC deployments become compute-bound as audience size grows, limiting the number of viewers that can be served efficiently from a single server instance without horizontal scaling or specialized infrastructure.

7.1.2 Server-Side Scalability of LL-HLS. In contrast to WebRTC, the server-side results show that LL-HLS scales efficiently with increasing client count. CPU and memory utilization remain relatively stable across the tested concurrency range, indicating that additional viewers do not significantly increase server-side computational load.

This behavior is a consequence of LL-HLS's encode-once, serve-many architecture. A single encoding pipeline produces a continuous stream of media segments and partial segments, which are then distributed to all clients via standard HTTP delivery. Once segments are generated, they can be reused by multiple clients without additional encoding or transport-specific state. As a result, server-side resource usage is largely independent of the number of viewers and is instead dominated by the cost of encoding and HTTP I/O. The results suggest that LL-HLS is well-suited for scenarios involving large audiences, where scalability and cost efficiency are primary concerns. CPU and memory usage remain comparatively low, allowing a single server to support substantially more viewers than a per-client streaming approach.

7.2 Latency and Startup Trade-offs

7.2.1 End-to-End Latency Characteristics. The client-side results show a clear distinction in end-to-end latency behavior between WebRTC and LL-HLS. Across all tested client counts, WebRTC consistently maintains sub-second latency, while LL-HLS operates in the multi-second range. These latency ranges remain largely stable as concurrency increases, indicating that neither protocol reaches a hard resource limit within the evaluated range.

WebRTC latency shows minimal variation with increasing client count, suggesting that the server is able to sustain real-time delivery as long as sufficient CPU and memory resources are available. LL-HLS exhibits a mild increase in latency as concurrency grows, but the overall magnitude remains within the expected range for low-latency HLS delivery.

Because the experiments were conducted on a local network, absolute latency values are lower than those expected in wide-area or cloud deployments. However, since both protocols traverse the same network path, the relative latency comparison remains valid. These results indicate that latency behavior is primarily governed by protocol design rather than transient network effects or server saturation under the tested conditions.

7.2.2 Startup Delay and Time to First Frame. The client-side results show a consistent difference in Time to First Frame (TTFF) between WebRTC and LL-HLS, with LL-HLS exhibiting higher startup delay across all tested client counts. This offset persists even at low concurrency levels, indicating that the baseline TTFF difference is inherent to protocol behavior rather than a result of server-side saturation.

In addition to this baseline gap, both protocols exhibit an increasing TTFF trend as the number of concurrent clients grows. Notably, this trend appears in both WebRTC and LL-HLS and follows a similar slope, suggesting a shared underlying cause. Given the experimental setup, this behavior is most plausibly attributed to client-side load generation rather than protocol-specific effects. Each additional client is implemented as a separate headless Chromium instance launched via Puppeteer on a single MacBook, increasing CPU scheduling contention and resource competition on the client host.

The presence of a similar TTFF increase across both protocols supports the interpretation that the observed growth in startup delay is dominated by client-side resource constraints introduced by the benchmarking methodology. As a result, the absolute TTFF values at higher client counts should be interpreted with caution. In real-world deployments, where viewers are distributed across independent client devices, such contention would not occur, and TTFF would be expected to remain largely independent of the total number of concurrent viewers.

Despite this client-side artifact, the relative TTFF difference between protocols remains consistent. LL-HLS requires playlist retrieval, segment or partial segment availability, buffering, and JavaScript-based parsing before playback can begin. WebRTC, in contrast, can begin rendering frames immediately after connection establishment and media reception. This structural difference explains the persistent TTFF gap observed across all concurrency levels, even as absolute TTFF values increase due to client-side load effects.

7.3 Answering the Research Questions

This subsection provides concise answers to the research questions defined in Section 3, based on the experimental results and the interpretations presented above.

7.3.1 RQ1: Server-Side Scalability Under Increasing Concurrency. WebRTC exhibits limited single-server scalability as concurrency increases: CPU utilization and memory consumption grow approximately linearly with the number of connected viewers due to per-client transport state, encryption contexts, congestion control, and buffering. In contrast, LL-HLS scales efficiently on a single server, with CPU and memory utilization remaining largely stable because segments are produced once and reused across viewers.

7.3.2 RQ2: Client-Side Quality of Experience Under Increasing Concurrency. WebRTC provides consistently lower end-to-end latency across the evaluated concurrency range, maintaining sub-second performance under load. In contrast, LL-HLS operates at higher end-to-end latency in the multi-second range under the low-latency configuration used in this work. WebRTC also achieves faster startup (lower TTFF), whereas LL-HLS incurs additional startup overhead due to playlist retrieval, segment availability, and player-side buffering. The TTFF increase observed at higher concurrency levels appears in both protocols and is likely influenced by client-side load generation (multiple headless Chromium instances on a single host), so absolute TTFF values at high concurrency should be interpreted with caution.

7.4 Architectural Implications of Protocol Choice

The experimental results expose a clear architectural trade-off between *latency* and *scalability*. WebRTC and LL-HLS are optimized for different operational goals, and the appropriate choice therefore depends on whether the primary constraint is responsiveness or audience size.

The scalability limitations of direct server-to-client WebRTC deployments raise an important architectural question: how can WebRTC be scaled to support larger audiences in scenarios where the latency characteristics of HTTP-based streaming, such as LL-HLS, are insufficient? For applications that demand ultra-low latency but must serve more than a small number of viewers, additional architectural mechanisms are required to balance immediacy with distribution efficiency.

Although WebRTC was originally designed for real-time communication, recent research and industry practice increasingly explore its use as a low-latency media distribution protocol [6, 9]. In particular, Selective Forwarding Units (SFUs) enable WebRTC to scale beyond simple peer-to-peer scenarios by decoupling media ingestion from media distribution.

In an SFU-based architecture, a media source publishes a single encoded stream to the SFU, which then forwards this stream to multiple receivers without decoding or re-encoding (Figure 9). This preserves WebRTC's low-latency transport characteristics while reducing per-viewer server workload compared to direct server-to-client WebRTC, shifting the bottleneck toward packet forwarding and available network egress.

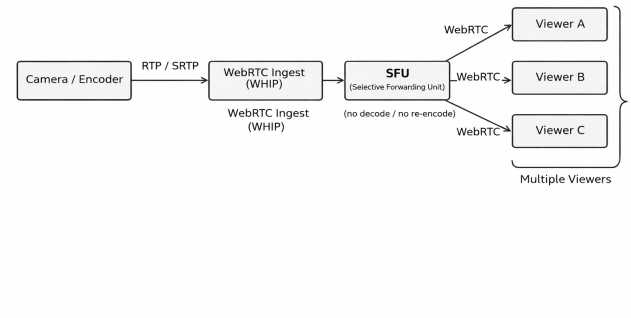

Figure 9: Simplified SFU-based WebRTC architecture illustrating single-stream ingestion and multi-client distribution without re-encoding.

Several studies indicate that SFU-based WebRTC can be a viable alternative to HTTP-based streaming for low-latency broadcast scenarios, particularly when interactivity, synchronization, or time-critical delivery is required [7, 9]. However, SFU-based designs also introduce challenges, including increased signaling complexity, state management, and the need for specialized infrastructure [6].

7.5 Limitations and Threats to Validity

This section discusses factors that may affect the interpretation and generalizability of the experimental results. While the experimental setup is designed to enable controlled and repeatable comparison between WebRTC and LL-HLS, certain limitations are inherent to the chosen methodology and environment.

7.5.1 Client-Side Load Generation. All client-side load is generated using multiple headless Chromium instances launched via Puppeteer on a single MacBook. As the number of concurrent clients increases, client-side CPU scheduling contention may affect playback initialization and rendering performance. This effect is most visible in Time to First Frame (TTFF), which exhibits an increasing trend for both protocols as concurrency grows.

Because this trend appears consistently across both WebRTC and LL-HLS, it is likely a benchmarking artifact rather than a protocol-specific behavior. In real-world deployments, where viewers are distributed across independent devices, such contention would not occur, and TTFF would be expected to remain largely independent of total viewer count.

7.5.2 LL-HLS Configuration Sensitivity. The LL-HLS pipeline in this work follows Apple's recommended low-latency configuration, including short segment durations, partial segments, and minimal server-side buffering. While this configuration minimizes latency, it reduces buffering slack and may increase sensitivity to timing variations and transient delivery delays.

Further tuning of segment size, part duration, and client buffer thresholds can shift the latency–robustness balance, but was outside the scope of this study. More broadly, LL-HLS fundamentally relies on segment-based delivery, which introduces discrete availability boundaries that can limit how low latency can be pushed in practice under conservative player buffering policies.

7.5.3 Local Network Environment. During the experiments, the streaming server was connected via wired Ethernet, while the client machine generating viewer load was connected via Wi-Fi. This reflects a common real-world deployment scenario, where servers typically use wired connections and end users often access streams over wireless networks.

Wireless connectivity may introduce additional variability in terms of latency and packet delivery compared to a fully wired setup. However, both WebRTC and LL-HLS clients were evaluated under identical network conditions, and all comparative results are therefore subject to the same client-side network characteristics. As a result, while absolute latency values may be influenced by Wi-Fi behavior, the relative performance differences between the protocols remain valid.

7.5.4 Measurement Window Duration. Metrics are collected over fixed 30-second steady-state windows after all clients have successfully connected. While sufficient to capture representative behavior under load, this duration may not reveal longer-term effects such as gradual memory pressure or adaptive congestion behavior.

Longer measurement periods or repeated runs over extended time horizons could provide additional insight into long-term system behavior but were outside the scope of this study.

7.5.5 Scope of Architectural Evaluation. This work focuses on direct server-to-client streaming without intermediate distribution layers such as Selective Forwarding Units (SFUs) or Content Delivery Networks (CDNs). While this approach simplifies comparison and highlights fundamental protocol characteristics, it does not capture the full range of deployment architectures used in production systems.

7.6 Reflection and Future Work

7.6.1 Reflection. This project has highlighted the complexity of multimedia and live video streaming systems. Working with media segmentation, frame generation, encoding pipelines, and network delivery revealed how many interconnected layers are involved even in seemingly straightforward streaming tasks. While this complexity made the implementation challenging at times, it also made the work engaging and technically rewarding.

A key insight is that live streaming technologies are highly adaptable and can be optimized for use cases beyond their original design goals. By adjusting architectural choices and delivery mechanisms, protocols such as LL-HLS and WebRTC can be repurposed to meet specific latency and scalability requirements. This flexibility has been particularly motivating and has deepened our understanding of system-level trade-offs.

The prototypes and evaluation framework developed in this project provide a strong foundation for future work. In the planned master's thesis, additional functionality such as Digital Video Recording (DVR) will be explored, which is expected to introduce new challenges related to buffering, storage, and latency management. However, the core implementations and the combined client- and server-side evaluation performed here offer a significant advantage for addressing these challenges in a structured manner.

Collaboration within the team has functioned well throughout the project. Although periods such as vacations and the Christmas break temporarily reduced momentum, we were able to complete all planned components and extend the scope beyond the initial objectives. The combination of independent work, regular coordination, and shared version control enabled an efficient and productive workflow.

7.6.2 Future Work. This project serves as a pre-study and foundation for exploring low-latency live video streaming architectures. To stay within the defined scope and time frame, the work deliberately omits Digital Video Recording (DVR) functionality and QUIC-based transport mechanisms. However, both represent important directions for future research and are crucial to the planned continuation of this work in the master's thesis, particularly for IoT and time-sensitive applications that require both sub-second end-to-end latency and broadcast-oriented features such as pausing, scrubbing, and time shifting.

Segment-based protocols such as LL-HLS and LL-DASH provide strong scalability and native support for DVR functionality, but suffer from an inherent latency floor caused by media segmentation and buffering mechanisms [1–3]. As a result, these protocols may be insufficient for applications that demand ultra-low latency while maintaining strict temporal alignment with real-world events. Conversely, WebRTC enables continuous media delivery with significantly lower latency and fast startup [5, 9], but lacks native support for DVR and large-scale broadcast distribution.

Future work will therefore investigate hybrid streaming architectures that combine the low-latency delivery characteristics of WebRTC with the scalability and DVR capabilities traditionally associated with segment-based protocols. In this context, emerging QUIC-based web APIs such as WebTransport and WebCodecs represent a promising direction for building flexible, low-latency media pipelines that support fine-grained buffering and time-shifted playback. A systematic experimental evaluation of QUIC-based approaches alongside WebRTC and LL-HLS is expected to provide deeper insight into the trade-offs between latency, scalability, and DVR functionality in IoT-oriented live streaming systems [1, 11].

References

[1] Abdelhak Bentaleb, May Lim, Mehmet N. Akcay, Ali C. Begen, Sarra Hammoudi, and Roger Zimmermann. 2025. Toward One-Second Latency: Evolution of Live Media Streaming. *IEEE Communications Surveys &; Tutorials* (2025), 1–1. doi:10.1109/comst.2025.3555514

[2] Abdelhak Bentaleb, Christian Timmerer, Ali C. Begen, and Roger Zimmermann. 2019. Bandwidth prediction in low-latency chunked streaming. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video* (Amherst, Massachusetts) *(NOSSDAV '19)*. Association for Computing Machinery, New York, NY, USA, 7–13. doi:10.1145/3304112.3325611

[3] Abdelhak Bentaleb, Zhengdao Zhan, Farzad Tashtarian, May Lim, Saad Harous, Christian Timmerer, Hermann Hellwagner, and Roger Zimmermann. 2022. Low Latency Live Streaming Implementation in DASH and HLS. In *Proceedings of the 30th ACM International Conference on Multimedia* (Lisboa, Portugal) *(MM '22)*. Association for Computing Machinery, New York, NY, USA, 7343–7346. doi:10.1145/3503161.3548544

[4] Nhu-Ngoc Dao, Anh-Tien Tran, Ngo Hoang Tu, Tran Thien Thanh, Vo Nguyen Quoc Bao, and Sungrae Cho. 2022. A Contemporary Survey on Live Video Streaming from a Computation-Driven Perspective. *Comput. Surveys* 54, 10s (2022), 1–38. doi:10.1145/3519552

[5] Nelson Francisco, Olie Baumann, Julien Le Tanou, and Richard Fliam. 2024. Ultra-low Latency Video Delivery Over WebRTC Data Channels. In *Proceedings of the 3rd Mile-High Video Conference* (Denver, CO, USA) *(MHV '24)*. Association for Computing Machinery, New York, NY, USA, 88–89. doi:10.1145/3638036.3640247

[6] Boris Grozev. 2020. Efficient and Scalable Video Conferences With Selective Forwarding Units and WebRTC. Technical report available on ResearchGate.

[7] Boris Grozev, Lyubomir Marinov, Varun Singh, and Emil Ivov. 2015. Last N: Relevance-based selectivity for forwarding video in multimedia conferences. In *Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*.

[8] Apple Inc. 2023. *Enabling Low-Latency HTTP Live Streaming (HLS)*. https://developer.apple.com/documentation/http-live-streaming/enabling-low-latency-http-live-streaming

[9] Bart Jansen, Timothy Goodwin, Varun Gupta, Fernando Kuipers, and Gil Zussman. 2017. Performance Evaluation of WebRTC-based Video Conferencing. *ACM SIGMETRICS Performance Evaluation Review* 45, 3 (2017), 56–68. doi:10.1145/3199524.3199534

[10] Qing Li, Xun Tang, Junkun Peng, Yuanzheng Tan, and Yong Jiang. 2023. Latency Reducing in Real-Time Internet Video Transport: A Survey. *SSRN Electronic Journal* (2023). doi:10.2139/ssrn.4654242

[11] Daniel Mejías, Inhar Yeregui, Ángel Martín, Roberto Viola, Pablo Angueira, and Jon Montalbán. 2025. Streaming Remote rendering services: Comparison of QUIC-based and WebRTC Protocols. arXiv:2505.22132 [cs.NI] https://arxiv.org/abs/2505.22132

[12] Colin Perkins et al. 2015. RTP Topologies. https://www.rfc-editor.org/rfc/rfc7667

[13] Colin Perkins et al. 2021. WebRTC Media Transport and Use of RTP. https://www.rfc-editor.org/rfc/rfc8830

[14] Leonardo Peroni and Sergey Gorinsky. 2024. An End-to-End Pipeline Perspective on Video Streaming in Best-Effort Networks: A Survey and Tutorial. *arXiv preprint* arXiv:2403.05192 (2024). https://arxiv.org/abs/2403.05192

[15] Liyang Sun, Tongyu Zong, Siquan Wang, Yong Liu, and Yao Wang. 2021. Towards Optimal Low-Latency Live Video Streaming. *IEEE/ACM Transactions on Networking* 29, 5 (2021), 2327–2338. doi:10.1109/TNET.2021.3087625