Learning to Switch: Adaptive Transport Protocols for High-Performance Video Streaming

Sapna Chaudhary, Kartikeya Sehgal, Sandip Chakraborty, *Senior Member, IEEE* and Mukulika Maity, *Member, IEEE*

***Abstract*—Following the standardization of QUIC, video streaming applications have increasingly adopted it as their preferred transport protocol. Although QUIC generally provides superior Quality of Experience (QoE) in dynamic and lossy network environments, TCP often outperforms it in high-speed scenarios, highlighting a key limitation: no single transport protocol consistently ensures optimal QoE across diverse network conditions. Furthermore, existing browsers lack mechanisms to dynamically switch between transport protocols in response to network variability. To address this gap, we present INTELSWITCH, a novel framework that enables adaptive switching between TCP and QUIC for video streaming. INTELSWITCH comprises two main components: modifications to the DASH player and Chromium browser to support in-session protocol switching, and a deep reinforcement learning (DRL) based decision engine that selects the optimal protocol based on real-time network and host conditions. We implement INTELSWITCH by extending `dash.js` and Chromium with protocol-aware playback capabilities and evaluate its performance under high-resolution streaming workloads and frequent network fluctuations. Our results demonstrate that INTELSWITCH consistently enhances QoE compared to static protocol choices, achieving the best improvements of up to 47% over TCP, 114% over QUIC, and 58% over a HEURISTIC-based solution.**

***Index Terms*—TCP, QUIC, Adaptive, DASH, Video streaming, Browser.**

I. INTRODUCTION

A significant portion of today's Internet traffic is dominated by video, accounting for nearly 65% of the total global traffic [1]. This surge in bandwidth-hungry applications has compelled service providers to continually explore new transport technologies that can sustain high performance and optimize user Quality of Experience (QoE). One such advancement is QUIC, a transport-layer protocol originally developed by Google, which has been standardized to improve web and video streaming performance, particularly in networks with high packet loss or constrained bandwidth. Since its adoption, major industry players such as Google, Cloudflare, and Meta have deployed QUIC widely to enhance application QoE. At the same time, the proliferation of high-speed access technologies such as WiFi 6/7 and 5G, offering bandwidths from hundreds of Mbps to multiple Gbps, is reshaping the Internet delivery landscape. Indeed, industry forecasts [1] predict that by 2028, the majority of global Internet traffic will traverse 5G networks. These developments highlight an urgent need for transport protocols that can not only withstand lossy and variable conditions but also fully exploit the capacity and stability of next-generation high-speed infrastructures.

Despite its widespread adoption, QUIC faces challenges in high-speed networks. Studies [2], [3], [4] show that host-side processing overhead in its user-space implementation limits throughput, with Zhang *et al.* [2] reporting up to 45% reduction and a 9% bitrate drop for video streaming compared to TCP. Bi *et al.* [4] further highlight server-side overhead under concurrent HTTP/3 sessions, while QUIC's reliance on UDP exposes it to ISP rate-limiting, further degrading performance. In contrast, QUIC excels in lossy or variable networks: Google's seminal study [5] showed 15–18% lower buffering on YouTube, and subsequent works [6], [7], [3] confirm its superiority over TCP in poor or fluctuating conditions. These results underscore a key limitation: no single transport protocol consistently achieves optimal QoE across all environments, motivating adaptive protocol selection for robust streaming performance.

Notably, several prior works [8], [9], [10] have explored adaptive protocol switching, mainly for web page delivery. Zhang *et al.* [8] proposed *WiseTrans*, which selects protocols based on measured response times across discrete categories (2G, 3G, 4G, WiFi) using a supervised machine learning model, with a follow-up study [11] adding online comparisons. However, WiseTrans is limited by its static model, coarse network categorization, lack of real-world traces, exclusion of high-speed networks (e.g., 5G), poor handling of delayed QoE effects in video, and probing overhead that can degrade user experience. Zhou *et al.* [9] introduced *FlexHTTP*, which combines network and content features to select between HTTP/2 and HTTP/3, but lacks rigorous empirical validation. Complementing these, Ganji *et al.* [10] proposed *Dynamic Transport Selection*, a probabilistic framework that adapts protocol choice to prevailing conditions. Despite these efforts, all focus on general web services rather than video streaming. Critically, none address streaming-specific challenges such as buffer dynamics, segment quality variability, and playback stability. This gap highlights the need for an adaptive protocol selection mechanism designed for video streaming, capable of learning from evolving network conditions and refining its strategy over time. Fig. 1 depicts a scenario where a user streaming video while walking experiences transition from WiFi to 5G network, followed by a handover from 5G to 4G network, during which bursty packet loss occurs. Such network dynamics make it difficult for a single transport protocol to perform well across all scenarios. Adaptive transport protocols can address this challenge by reducing rebuffering

S. Chaudhary, Kartikeya Sehgal, and M. Maity are with the Indraprastha Institute of Information Technology (IIIT) Delhi, India 110020, Emails: {sapnac, kartikeya22244, mukulika}@iiitd.ac.in

S. Chakraborty is with the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Kharagpur, India 721302, Email: sandipc@cse.iitkgp.ac.in

and sustaining higher video bitrates, thereby improving the overall QoE. For video providers, this can translate into greater user satisfaction and engagement, which in turn can drive higher retention, increased watch time, and ultimately greater revenue.

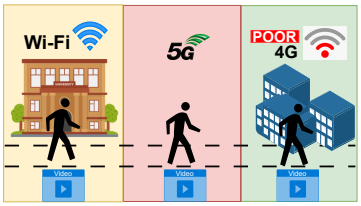


Fig. 1: Scenario: Network change from WiFi to 5G and then to 4G while streaming a video.

Motivated by these opportunities and to overcome the aforementioned limitations, in this paper, we propose INTELSWITCH, an adaptive transport protocol selection mechanism tailored for video streaming applications. Unlike prior solutions, INTELSWITCH operates entirely at the client side and requires no server-side modifications. It continuously monitors network and QoE-related metrics through the DASH player (Dynamic Adaptive Streaming over HTTP, an adaptive bitrate protocol used widely for video streaming) and employs a decision engine to dynamically select the most suitable transport protocol under varying network and host conditions. To realize this, INTELSWITCH incorporates two key components: (1) modifications to both the video player and the browser to support real-time adaptive protocol switching, and (2) a protocol decision engine that recommends the optimal protocol at the granularity of individual video segments. To overcome the shortcomings of supervised learning, INTELSWITCH leverages a deep reinforcement learning (DRL) model trained on real-world network measurements, enabling adaptive, data-driven decision-making that reflects authentic network behavior and QoE dynamics. We implement INTELSWITCH by extending the Chromium browser and the `dash.js` player with protocol-aware playback capabilities.

We conduct extensive evaluation of INTELSWITCH across diverse and highly dynamic network conditions, including scenarios with extreme bandwidth fluctuations. We compare INTELSWITCH against static TCP, static QUIC, and a throughput-based HEURISTIC across different network traces and perform both replay-based and real-time-based evaluation to capture both controlled and real environments. Our results show that INTELSWITCH consistently outperforms static protocol choices by intelligently switching between TCP and QUIC to maintain higher QoE. In the replay-based emulated setup, INTELSWITCH achieves a mean improvement of up to 47% over TCP, 114% over QUIC, and up to 18% over a HEURISTIC-based solution. In the case of a real-time based scenario for a 5G-mmWave network, INTELSWITCH achieves a mean percent improvement of upto 21% over TCP, 81% over QUIC, and 58% over HEURISTIC in overall QoE. Furthermore, we demonstrate its effectiveness across widely used ABR algorithms in `dash.js`, namely *BolaRule*, *ThroughputRule*, and *Dynamic*.

In summary, this paper makes the following contributions:

1) We demonstrate that the QoE of video streaming applications can significantly degrade on high-speed links when QUIC is used as the transport protocol, primarily due to client-side processing overhead. Through a series of pilot experiments, we observe that no single static transport protocol is optimal across all network conditions.
2) We propose INTELSWITCH, a deep reinforcement learning (DRL) based framework that dynamically selects a suitable transport protocol (TCP or QUIC) for each video segment based on network conditions and application QoE. We implement INTELSWITCH by extending both the `dash.js` player and the Chromium browser to enable adaptive protocol switching. We contributed to about 140 and 503 lines of code to `dash.js` and the Chromium browser.
3) We conduct extensive evaluations of INTELSWITCH under 9 diverse network patterns covering varied types of networks, from controlled to real 5G networks. We compare its performance against static protocol choices (TCP and QUIC) and a HEURISTIC baseline. INTELSWITCH improves the application QoE by upto 47% over only TCP, 114% over only QUIC and 58% over the HEURISTIC baseline.

II. RELATED WORK

We categorize related work into three areas: (1) application QoE over TCP vs. QUIC, (2) QUIC performance in high-speed networks, and (3) adaptive transport protocol switching.

A. QUIC vs TCP Performance

Video Streaming Performance: Google's seminal work [5] showed that QUIC reduced YouTube playback rebuffering by 18% on desktop and 15.3% on mobile. Kakhki *et al.* [6] similarly reported improved QoE, particularly for high-quality content. Arisu *et al.* [12] analyzed interface switches (e.g., $WiFi \leftrightarrow LTE$, $WiFi \leftrightarrow 3G$) and found QUIC consistently reduced rebuffering and improved average bitrates, with the largest gains in scenarios where adaptation algorithms lacked buffer-awareness. Szabó *et al.* [13] showed that QUIC reduced initial buffering time by 6–49%, depending on video and network characteristics. Bhat *et al.* [14] compared QUIC and TCP for HTTP Adaptive Streaming (HAS) algorithms and found limited benefits, as most algorithms were designed for TCP behavior. In contrast, Zinner *et al.* [15] demonstrated that QUIC, particularly with 0-RTT, improved playback startup time. Tanya *et al.* [3] further showed QUIC's advantages in low-bandwidth, high-RTT settings, where it reduced both stall frequency and duration. Overall, while QUIC provides clear benefits in lossy or variable networks, recent studies highlight that it does not consistently achieve optimal performance in high-speed environments, particularly above 500 Mbps.

QUIC under High-Speed Networks: Zhang *et al.* [2] showed that QUIC underperforms in high-bandwidth settings, with bitrate reductions of up to 9.8% due to receiver-side

processing overhead from excessive packet handling and user-space acknowledgments. Tanya *et al.* [3] highlighted trade-offs between TCP and QUIC: while QUIC achieves higher throughput for small files ($\leq$ 20 MB), TCP outperforms it for larger transfers (>20 MB up to 2 GB), as QUIC's overheads outweigh its 0-RTT benefits. Profiling a 200 MB download further revealed that QUIC consumes nearly twice the CPU resources of TCP for similar throughput. Kempf *et al.* [16] benchmarked QUIC on 10G networks, observing large performance variations across implementations, with packet I/O as the dominant bottleneck and performance strongly tied to CPU architecture and core allocation. Similarly, Bi *et al.* [4] found that in high-bandwidth networks, QUIC's computational overhead shifts the bottleneck from the network to the CPU. Consequently, HTTP/3 (over QUIC), despite being the successor to HTTP/1.1 (over TCP), can achieve lower throughput when CPU resources are constrained. This degradation stems from QUIC's CPU-intensive operations such as ACK processing, packet transmission, and encryption, making CPU efficiency critical for sustaining performance in high-speed environments.

B. Adaptive Transport Protocol Switching

Beyond TCP vs. QUIC performance comparisons, several works have explored adaptive transport protocol switching to improve web service efficiency. Zhang *et al.* [8] proposed *WiseTrans*, which leverages machine learning and historical network data to address spatial heterogeneity and make request-level protocol-switching decisions. A follow-up study [11] introduced real-time experiments to compare protocol performance before switching. However, *WiseTrans* is limited by its static design: the model is trained on fixed data, reducing adaptability to evolving networks, and relies on discrete categories (2G, 3G, 4G, WiFi) that fail to capture real-world variability. Zhou *et al.* [9] proposed *FlexHTTP*, which selects between HTTP/2 and HTTP/3 using both global and local information to update its classifier. While more adaptive in principle, FlexHTTP lacks rigorous empirical validation. Moving away from machine learning, Ganji *et al.* [10] introduced *Dynamic Transport Selection*, a probabilistic framework for protocol adaptation. Yet, it does not incorporate application-specific performance metrics. Despite their differences, all these approaches share a critical limitation: they focus on general web services rather than video streaming. Importantly, none address the unique challenges of adaptive streaming, such as buffer dynamics, segment quality variation, and playback stability. This gap underscores the need for a transport protocol selection mechanism tailored to video streaming, capable of learning from evolving conditions and continuously refining its strategy over time.

III. Motivation and Design Challenges

In this section, we first motivate the need for an adaptive protocol switching mechanism. Next, we conduct experiments to see the performance of TCP and QUIC for video streaming under different types of networks. Finally, we present the challenges for designing such a solution.

A. Motivation

Video streaming performance is highly sensitive to the choice of transport protocol, with QUIC (HTTP/3) and TCP (HTTP/2) each offering distinct strengths depending on the network environment. This asymmetry creates an important challenge: neither protocol alone consistently ensures the best Quality of Experience (QoE), motivating the need for adaptive protocol switching.

QUIC's strengths in challenging networks: QUIC was designed to improve performance in unreliable or lossy environments, which are common in mobile and wireless settings. By enabling faster connection establishment (0-RTT), multiplexing streams to avoid head-of-line (HoL) blocking, and handling losses more efficiently than TCP, QUIC significantly reduces rebuffering events and increases average playback bitrates. Multiple studies confirm its superiority over TCP in such scenarios [5], [12], [13], [6], [7], [3].

QUIC's challenges in high-speed networks: In contrast, QUIC struggles in modern high-speed, low-latency environments such as WiFi 6/7 and 5G. Its user-space implementation of ACK processing and encryption introduces substantial computational overhead and limiting throughput [2]. At gigabit speeds, packet I/O becomes the dominant bottleneck [16], [3], with QUIC consuming nearly twice the CPU resources of TCP to deliver comparable throughput [3], [4]. As a result, TCP often outperforms QUIC on well-provisioned links.

Dynamic trade-offs across conditions: These contrasting performance profiles imply that a static choice of transport protocol is inherently suboptimal. TCP is better suited for high-bandwidth, low-latency environments where CPU overheads dominate, while QUIC excels in lossy, high-latency, and mobile networks. However, real-world conditions are highly dynamic, with performance shifting due to user mobility, fluctuating network load, and varying host resources. This variability motivates the need for an intelligent, adaptive framework that can switch between TCP and QUIC based on real-time network and QoE metrics, ensuring consistently high streaming performance across heterogeneous environments.

B. Pilot Study

Motivated by prior findings, we investigate the performance gaps between TCP and QUIC in high-bitrate video streaming. To this end, we conduct motivation experiments under diverse network conditions, ranging from high-bandwidth to lossy environments. The setup has three components: (a) a client running a DASH player with BOLA as the ABR algorithm, (b) a server hosting video segments, and (c) a network emulator (mahimahi [17]). We use a video from the FFmpeg filter testing set [18], encoded at bitrates from 500 Kbps to 700 Mbps, and evaluate across five networks (1) high-bandwidth Ethernet (1 Gbps), (2 & 3) variable high-bandwidth 5G mid-band drive and mmWave walk traces [19], (4) variable low-bandwidth 4G walk trace [20], and (5) poor-bandwidth with 5% packet loss and 50 ms delay emulated using `tc netem` [21].

Fig. 2(a) shows the average QoE of TCP (HTTP/2) and QUIC (HTTP/3) across these scenarios. On Ethernet, TCP reaches the maximum 700 Mbps bitrate, while QUIC is capped

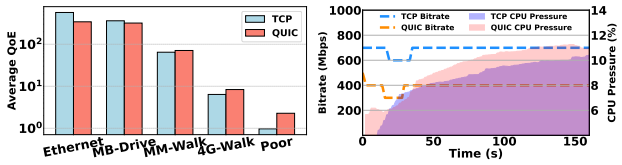


Fig. 2: (a) Average QoE for TCP and QUIC across different network scenarios and (b) Bitrate (Mbps) vs Time (seconds) of TCP and QUIC while streaming a video at 1Gbps bandwidth and the corresponding CPU pressure (%).

around 400 Mbps. In the 5G mid-band drive (MB-Drive), TCP again outperforms QUIC, particularly above 500 Mbps, consistent with prior work attributing QUIC's limits to client-side overhead [2]. We thus monitor CPU pressure i.e, the fraction of time tasks are stalled due to CPU contention (PSI [22]). Fig. 2(b) further support this: despite lower throughput, QUIC exhibits equal or higher CPU pressure than TCP, confirming earlier findings that user-space processing bottlenecks hinder QUIC at high bandwidths [3], [2], [4]. Conversely, in poor or lossy networks (4G-walk), QUIC outperforms TCP due to its ability to avoid head-of-line (HoL) blocking and recover faster from losses. In moderately fluctuating yet provisioned networks (e.g., stable 4G/5G), the benefits of QUIC are less pronounced.

These results highlight a clear trade-off: TCP dominates in high-speed environments, while QUIC excels in lossy conditions. This motivates the need for a dynamic switching mechanism that selects the most suitable transport protocol based on real-time network and host conditions. Realizing such a mechanism requires coordinated changes to both the video player and browser infrastructure.

C. Design Challenges

Deciding the optimal transport protocol in real-world network environments is a non-trivial problem due to several key challenges. **(1) Dynamic network conditions** make video streaming performance highly sensitive to fluctuations in bandwidth, RTT, and packet loss, particularly in mobile or wireless environments. Existing approaches that rely on coarse network categories (e.g., 2G/3G/4G/WiFi) or static supervised models fail to capture temporal dependencies and unseen states, highlighting the need for continuous, adaptive learning from real-time conditions. **(2) Handling video streaming characteristics** introduces additional complexity, as adaptive video streaming involves intricate buffer dynamics, segment quality variations, and playback stability considerations that directly influence user experience. This necessitates an adaptive transport protocol selection mechanism capable of learning from evolving network and playback contexts to optimize long-term QoE. **(3) Overhead of protocol switching** presents another difficulty – switching protocols mid-stream often requires new handshakes, potentially causing playback delays or interruptions that degrade QoE, thereby demanding efficient mechanisms to minimize disruptions. Moreover, **(4) accurate network metric estimation** remains a significant hurdle since browsers operate at the application layer and

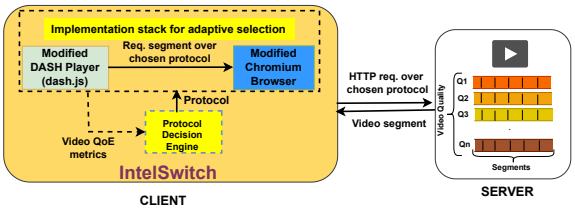


Fig. 3: A schematic view of INTELSWITCH for video streaming application. The coloured box indicates a client-only solution. Two major building blocks: 1) Protocol decision engine, 2) Implementation stack.

lack direct access to precise network metrics such as RTT, bandwidth capacity, and loss rate. Instead, they rely on noisy or delayed indirect estimates, complicating real-time assessment of network states and their effects on playback performance. The **(5) complexity of Chromium browser modification** further amplifies the challenge, as implementing dynamic protocol switching necessitates intricate code changes across multiple layers of Chromium's large, multi-layered architecture, where errors can adversely impact QoE. Finally, **(6) deployment without server modifications** is essential, as practical solutions must function entirely on the client side without requiring server-side changes, relying solely on browser-level data and interactions. Next, we discuss the design of INTELSWITCH to address these challenges.

IV. INTELSWITCH: SYSTEM OVERVIEW

We design INTELSWITCH, a fully client-side system that dynamically selects the transport protocol during video streaming to optimize user QoE. Fig. 3 shows its schematic overview. INTELSWITCH comprises of two key components: (a) a **protocol decision engine** that adaptively selects the optimal transport protocol, and (b) an **implementation stack** that enables seamless protocol switching within the video player and browser. The system operates as follows: (i) the video player reports QoE metrics to the decision engine; (ii) the engine determines the optimal protocol and returns the decision; (iii) the player issues segment requests specifying the chosen protocol; and (iv) the modified browser transmits the request to the server over the corresponding transport protocol.

To address the challenges discussed earlier, each component of INTELSWITCH is carefully engineered. **(1) Adaptive protocol selection under dynamic conditions:** The protocol decision engine leverages deep reinforcement learning to continuously adapt protocol selection based on real-time network and playback metrics. By modeling temporal relationships between network characteristics (bandwidth, RTT, loss) and QoE indicators (buffer level, bitrate, dropped frames), it overcomes the limitations of static or coarse-grained approaches. **(2) Handling video streaming specifics:** The DASH player is instrumented to monitor playback metrics such as buffer occupancy, segment quality, and playback stability at the segment level, enabling protocol decisions that balance bitrate selection and rebuffering risk to ensure long-term QoE stability. **(3) Overcoming protocol switching overhead:** To

avoid re-establishment delays when switching between TCP and QUIC, INTELSWITCH pre-initializes both connections at session startup, assigning audio to TCP and video to QUIC, and reuses these persistent paths for seamless mid-stream switching without interrupting playback. **(4) Bridging the visibility gap in network metrics:** Since browsers lack direct access to low-level network statistics, INTELSWITCH derives indirect indicators such as throughput estimates, buffer levels, and rebuffering events from `dash.js`, allowing the decision engine to make robust decisions from incomplete observations. **(5) Seamless integration with browser architecture:** The Chromium browser is modified to accept a custom protocol field from the DASH player and to initiate network requests over TCP or QUIC accordingly. These modifications are confined to critical interfaces to preserve stability, concurrency safety, and compatibility with Chromium's native behavior. **(6) Minimal deployment overhead:** Because all modifications reside on the client side, within `dash.js` and Chromium, INTELSWITCH requires no changes to video servers, enabling straightforward deployment across heterogeneous devices and networks.

Overall, INTELSWITCH unifies adaptive learning, efficient protocol management, and lightweight instrumentation to deliver robust, QoE-aware video streaming across dynamic network environments. The next sections detail the design of the protocol decision engine and the implementation of the adaptive protocol switching stack.

V. PROTOCOL DECISION ENGINE

In this section, we present the *Protocol Decision Engine*, a Deep Reinforcement Learning (DRL) based mechanism that adaptively selects the transport protocol to optimize video streaming performance. The engine operates independently of the underlying video player and browser, which execute the protocol decisions generated by it.

A. Need for DRL-based protocol decision engine

Video streaming performance is shaped by multiple interacting factors, including network variability, video characteristics, and the underlying transport protocol, making it highly unpredictable. Static, rule-based decision engines cannot capture this complexity, while prior supervised learning approaches [8] rely on labeled data derived from prior knowledge or fixed thresholds, limiting their adaptability and scalability. Moreover, supervised models generalize poorly to unseen or rapidly changing environments. In contrast, RL eliminates the need for predefined labels by learning directly through interaction with the environment, selecting actions such as choosing between TCP and QUIC, and optimizing them based on rewards that reflect QoE, stability, or efficiency. Consequently, we adopt a DRL approach for protocol selection, which enables adaptive, label-free learning and robust performance across dynamic and heterogeneous network conditions.

B. Problem Formulation

At each time step t, the decision engine predicts the most suitable transport protocol for a client streaming video in a given network environment. The decision depends on both (1) network parameters, such as throughput, and (2) QoE-related parameters, including current bitrate, bitrate variation, and rebuffering events. Thus, the decision process requires continuous feedback evaluation and adaptive action selection.

Given the stochastic nature of Internet traffic, we formulate protocol selection as a Markov Decision Process (MDP), where an agent interacts with the environment, selects actions, and receives rewards. A finite MDP is defined by a tuple (S, A, P, R), where: (1) S is the state space (all possible environment states), (2) A is the set of discrete actions, (3) P defines the state transition probabilities $P(s'|s, a)$, and (4) R is the reward function mapping state-action pairs to scalar rewards. In our case, the agent observes a state vector at each time t, representing network and QoE conditions, and selects an action choosing between HTTP/2 or HTTP/3 based on action probabilities. We solve this MDP using an actor–critic network (A3C). Note that the RL-based ABR algorithms, such as Pensieve [23], also used an A3C network for bitrate-selection. In adaptive video streaming, the behavior of the system is tightly coupled with the choice of transport protocol. For instance, selecting QUIC over TCP can result in different throughput, latency, and buffer dynamics. To adapt effectively to such changes, we employ *Proximal Policy Optimization* (PPO) [24], an on-policy, policy-gradient reinforcement learning algorithm that enhances training stability and sample efficiency through controlled policy updates. We replace the vanilla A3C update rule with PPO, retaining the same network architecture as Pensieve while benefiting from PPO's improved convergence stability and reliability.

C. Decision Engine Offline Training

To the best of our knowledge, no existing dataset captures video streaming over both TCP and QUIC across diverse network conditions, including high-bandwidth settings such as 5G mid-band, mmWave, and Ethernet. To fill this gap, we collected a comprehensive dataset covering 1 Gbps Ethernet, 5G mmWave, 5G mid-band (walking and driving) [19], 4G walking [20], and poor network conditions derived from [23] and emulated using `tc netem` [21] with 5% packet loss and 50 ms delay. Each scenario includes parallel TCP and QUIC sessions, forming the basis for offline training of our decision engine. Fig. 4 illustrates the overall workflow of the DRL-based protocol decision engine, and Table I summarizes the selected state features used for training.

TABLE I: Protocol decision engine states information

Abbreviation	Definition
CB (Current Bitrate)	Bitrate of the most recently played video segment.
Rebuf (Rebuffering)	Duration of the most recent rebuffering event.
T (Throughput)	Measured throughput experienced by the application.

Input State: At each time step t, the agent extracts the state vector $s_t = \langle \vec{\mathbf{CB}}, \vec{\mathbf{Rebuf}}, \vec{\mathbf{T}} \rangle$ of a segment k from the collected TCP and QUIC dataset.

Action Space: The action space A is the best protocol for the next segments for each client. The decision engine policy π maps the state to a discrete action $A = \{0, 1\}$, where 0 : HTTP/2 and 1 : HTTP/3. The policy π maps s_t to an action.

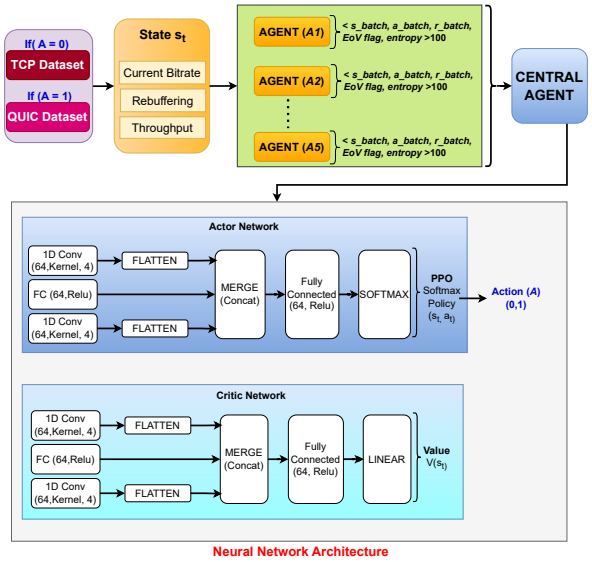


Fig. 4: Protocol Decision Engine: We employ a A3C network with PPO-based policy updates

Reward: The reward is defined as a function of the segment's current bitrate, the rebuffering duration, and the throughput experienced at time t. The R_t is defined as:

$$\begin{aligned} R_t &= b_t - r_t \\ b_t &= \log_2(CB_t) \\ r_t &= \log_2(1 + (\beta \cdot \text{Rebuf}_t)) \end{aligned} \tag{1}$$

Here, b_t denotes the bitrate of the segment at time t, while r_t represents the rebuffering experienced during the download of that segment of bitrate b. Rebuffering is penalized with β set to the maximum available bitrate (Mbps). We use logarithmic scaling to balance the reward magnitudes across bandwidth extremes, preventing bias towards high-throughput scenarios (e.g., Ethernet) and enabling consistent learning across heterogeneous conditions.

D. Neural Network Architecture

We employ an A3C framework with multiple agents running in parallel to explore the environment and asynchronously update a shared global model. The actor and critic networks share the same input state but perform different roles: the actor outputs action probabilities, while the critic estimates the expected reward. Both networks use a hybrid architecture combining fully connected (FC) and one-dimensional convolutional (Conv1D) layers. Each input state includes three feature channels: (1) scalar features (e.g., rebuffering) processed by FC layers with 64 ReLU neurons, and (2) temporal features (e.g., bitrate and throughput histories) processed by Conv1D layers with 64 filters and a kernel size of 4. The resulting outputs are concatenated and passed through another FC layer. The actor then applies a softmax layer to produce protocol selection probabilities, while the critic outputs a scalar value $V(s_t)$ representing the state's expected return. After selecting an action, each agent computes the reward and transmits the `<state, action, reward>` tuple to the central agent for model updates.

The overall procedure for transport protocol switching is summarized in Algorithm 1, and the mathematical description of the algorithm is discussed in the next subsection.

Algorithm 1 PPO-Based Protocol Decision Engine

Initialize: Actor network π_θ, Critic network V_ϕ, entropy weight $\beta \leftarrow 0.001$
Set learning rates $\alpha_\theta = 0.0003$, $\alpha_\phi = 0.001$
Set epoch counter $k \leftarrow 0$ and training batch = 100
while training not converged **do**
  Step 1: Collect Trajectories
  for each agent i in parallel **do**
    Run policy π_θ in environment
    for each timestep t **do**
      Observe state s_t of TCP and QUIC
      Sample action $a_t \sim \pi_\theta(a|s_t)$
      Receive reward r_t
      Store tuple $(s_t, a_t, r_t, e_v, e_r)$
    end for
    Send trajectory $\tau_i = \{(s_t, a_t, r_t, e_v, e_r)\}$ to central agent
  end for
  Step 2: Compute Returns and Advantages
  for each trajectory in batch $D_k = \{\tau_i\}$ **do**
    for each timestep t (in reverse) **do**
      Compute TD-error: $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
      Compute GAE: $A_t = \delta_t + \gamma\lambda A_{t+1}$
      Compute return: $\hat{R}_t = A_t + V(s_t)$
    end for
  end for
  Step 3: Optimize Policy and Value Network
  for each timestep t **do**
    Compute probability ratio: $r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$
    Compute clipped policy objective L_{clip} using r_t and advantage $\hat{A}_t$
    Compute entropy $H(\pi_\theta)$ from action distribution
    Actor loss: $L_{\text{actor}} = -(L_{\text{clip}} + \beta \cdot H(\pi_\theta))$
    Critic loss: $L_{\text{critic}} = (V_\phi(s_t) - \hat{R}_t)^2$
  end for
  Step 4: Update Network Parameters
  Update actor: $\theta \leftarrow \theta - \alpha_\theta \nabla_\theta L_{\text{actor}}$
  Update critic: $\phi \leftarrow \phi - \alpha_\phi \nabla_\phi L_{\text{critic}}$
  Update each agent with the updated actor and critic
  Step 5: Logging
  Log average reward, TD loss, and entropy
  Increment epoch: $k \leftarrow k + 1$
end while

E. PPO-Based Protocol Decision Engine Training:

To enhance and speed up the training, we use multiple agents in our A3C architecture. At each timestamp t, each agent interacts with the environment and collects trajectories, from which policy and value gradients are computed by following the procedure below:

1) Collection of Trajectories: Each agent periodically sends the collected trajectories to the central agent in the form of batches $(s_t, a_t, r_t, e_v, e_r)$, where s_t denotes the observed states, a_t the chosen actions, r_t the corresponding rewards, e_v an end-of-video indicator, and e_r the entropy record.

2) Cumulative Return and Advantage Computation: Upon receiving trajectory batches $\tau = (s_0, a_0, r_0, e_{v0}, e_{r0} \ldots)$ from all agents, the central agent computes the cumulative returns:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2}$$

where δ_t is the temporal difference error. While δ_t provides a one-step advantage estimate, using it alone introduces bias, and full Monte Carlo returns yield high variance. To balance this trade-off, we employ Generalized Advantage Estimation (GAE), which uses the computed δ_t to recursively estimate the

advantage A_t. Here, γ is the discount factor controlling the influence of future rewards, and λ adjusts the bias–variance trade-off by determining how much of the future advantage is incorporated. $V(s_t)$ and $V(s_{t+1})$ denote the critic's predicted state values, with $V(s_{t+1}) = 0$ at terminal states. The recursion proceeds backward through each trajectory, producing smooth and reliable advantage estimates that stabilize learning.

$$A_t = \delta_t + \gamma\lambda A_{t+1} \tag{3}$$

The critic's final return target, R_t is obtained by combining the estimated advantage with the predicted network value.

$$\hat{R}_t = A_t + V(s_t) \tag{4}$$

3) Optimization of Policy and Value Network: The estimated advantages obtained via GAE serve as inputs to the PPO update rule. PPO uses a clipped surrogate objective to stabilize training by preventing large policy updates. The clipped surrogate objective is defined as:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(R_t(\theta)\hat{A}_t,\ \text{clip}(R_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]. \tag{4}$$

where $R_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the ratio of the probability of selecting an action in the new policy to the old policy. $\hat{A}_t$ is the estimated advantage at time step t, and ϵ is the clipping parameter. This objective compares two terms inside the minimum function. The first term, $R_t(\theta)\hat{A}_t$, is the standard policy gradient update that encourages actions with positive advantage and discourages actions with negative advantage. The second term clip$((R_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)$ prevents the policy from changing too aggressively in a single update. If the new policy moderately increases the probability of an action, PPO permits the update. However, if the increase is too large, PPO clips the update to prevent overfitting and ensure training stability. The actor is trained using PPO's clipped surrogate loss with $\epsilon = 0.2$, which constrains the policy update by clipping the probability ratio to the range $[1-\epsilon, 1+\epsilon]$, ensuring stable updates and preventing destructive policy shifts. We further enhance the clipped surrogate objective by adding an entropy bonus. The purpose of this term is to encourage the agent to maintain sufficient exploration during training, instead of getting stuck with one action. Therefore, the entropy of the softmax output is computed as:

$$\mathcal{H}(\pi_\theta) = -\sum_a \pi_\theta(a \mid s_t) \log \pi_\theta(a \mid s_t). \tag{5}$$

where the summation runs over all possible actions a. Here, $\pi_\theta(a \mid s_t)$ denotes the probability of selecting action a under the current policy π with parameters θ and the negative sign ensures that entropy is non-negative. Further, combining the clipped surrogate objective with the entropy regularization term yields the final actor loss. Therefore, the total objective optimized by the actor becomes:

$$\mathcal{L}_{\text{actor}} = -(L^{\text{CLIP}}(\theta) + \beta \cdot \mathbb{E}_t \left[\mathcal{H}(\pi_\theta)\right]) \tag{6}$$

While the actor is updated to improve the policy and encourages exploration, the critic is updated to minimize the prediction error of state values. Specifically, we use AdamOptimizer to minimize both actor and critic losses. For the actor, it minimizes the negative PPO objective plus entropy, which helps to improve the policy while encouraging exploration. For the critic, it minimizes the mean squared error between the estimated return and the critic's value, helping it predict returns more accurately. The critic network is updated by minimizing the mean squared error (MSE) between the predicted value $V_\phi(s_t)$ and the GAE return target $\hat{R}_t$. The MSE is averaged over the batch to ensure stable gradients during training.

$$\mathcal{L}_{\text{critic}} = \mathbb{E}_t \left[\left(V_\phi(s_t) - \hat{R}_t \right)^2 \right] \tag{7}$$

4) Updation of Network Parameters: After computing the actor and critic losses, the corresponding gradients are sent to the central agent. The central agent aggregates gradients across agents and applies updates to the shared global actor and critic networks. These gradients are then applied sequentially to the shared global networks. Mathematically, for each agent i , the actor parameters θ and critic parameters ϕ are updated as:

$$\theta \leftarrow \theta - \alpha_\theta \nabla_\theta \mathcal{L}^i_{\text{actor}} \tag{8}$$

$$\phi \leftarrow \phi - \alpha_\phi \nabla_\phi \mathcal{L}^i_{\text{critic}} \tag{9}$$

where α_θ and α_ϕ are the actor and critic learning rates. Moreover, $\mathcal{L}^i_{\text{actor}}$ is the gradient of the actor loss with respect to the actor's parameters θ, and $\mathcal{L}^i_{\text{critic}}$ is the gradient of the critic loss with respect to the critic's parameters ϕ.

F. Offline Training Justification

We rely on offline training using real-world measurements instead of live training for several reasons. First, concurrent multi-agent training in real systems (e.g., high-bandwidth setups) would artificially constrain bandwidth, distorting protocol behavior. Second, live training requires substantial infrastructure for traffic generation, QoE computation, and real-time decision making, making it computationally infeasible. Offline training avoids these issues while still capturing protocol dynamics across diverse environments.

VI. Implementation

We present the modified video player and browser that implement the decision engine's transport decisions and generate corresponding media requests (video and audio) to the streaming server. Our implementation builds on `dash.js` version 5.0.0, an open-source MPEG-DASH player[1], and a customized Chromium browser[2] version 126.0.1666.0. The codebase and the implementation details are available for public use[3]. We first outline the baseline architectures of `dash.js` and Chromium, followed by our design modifications and their underlying rationale.

[1]https://github.com/Dash-Industry-Forum/dash.js. All links of the paper are last accessed on January 14, 2026.

[2]https://source.chromium.org/

[3]https://github.com/sapna2504/IntelSwitch-adaptive-protocol-switching/

A. Background

DASH player: In the `dash.js` video player (Fig. 5), the *ScheduleController* queries the *ABRController* to decide whether to change the segment quality based on throughput, buffer occupancy, and related metrics. Upon a quality decision, the *MEDIA_FRAGMENT_NEEDED* event is triggered, prompting the *StreamProcessor* to generate the corresponding segment request. This request passes through the *FragmentModel* and *HTTPLoader*, which employs either *XHRLoader* or *FetchLoader* to fetch the segment from the server. Once downloaded, the segment returns via *FragmentModel* to the *BufferController*, which appends it to the media buffer for playback. As the DASH player operates within a web browser, seamless browser integration is crucial for enabling protocol-aware streaming.

Chromium browser: When the DASH player requests a video segment, the browser translates it into an HTTP transaction with the streaming server. Internally, this fragment request passes through the *HTTPLoader* and *XHRLoader*, where the latter initiates the actual transfer using XMLHttpRequest (XHR). The XHR call is first handled by Chromium's rendering engine, *Blink*, where it is converted into the browser's internal request format. The request then passes to the renderer's loader, which manages network resource fetching. Using Chromium's *mojom* interprocess communication (IPC) system, the request is sent from the renderer to the browser's *Network Service Layer*. The browser then constructs the final network request and selects the transport protocol – HTTP/2 over TCP or HTTP/3 over QUIC. For this, Chromium maintains two parallel jobs: *main_job* (TCP) and *alt_job* (QUIC). If the server supports QUIC or another alternate protocol, the request binds to *alt_job*; otherwise, it defaults to *main_job*. This binding logic resides in `http_stream_factory_job_controller.cc`.

B. Implementation Challenges for DASH Player Modifications

While the standard `dash.js` player is agnostic to the underlying transport protocol, adapting it for protocol-aware streaming required several non-trivial changes. Extending `dash.js` to support per-request protocol control and QoE-driven feedback introduced four main challenges. **(1) Separation of functional layers:** The player's modular design separates metadata parsing (`FragmentRequest.js`), request orchestration (`HTTPLoader.js`), and transport execution (`XHRLoader.js`). Protocol-awareness required consistent injection of transport-selection decisions across these layers without breaking modularity, as inconsistencies could cause protocol intent to be lost before reaching Chromium. **(2) Maintaining ABR independence:** The Adaptive Bitrate (ABR) logic operates independently of the transport layer, making it critical to integrate protocol switching without entangling the two; otherwise, coupling could destabilize adaptation and degrade QoE. **(3) Feedback-loop integration:** Since `dash.js` does not natively expose fine-grained QoE metrics, the system had to be instrumented to capture parameters such as startup delay, rebuffering events, and segment download times, while keeping the reporting path lightweight to avoid disrupting playback. **(4) Overhead minimization:** Additional metadata handling and metric reporting had to be implemented asynchronously to prevent playback delays or CPU overhead, ensuring that protocol-awareness did not compromise real-time performance.

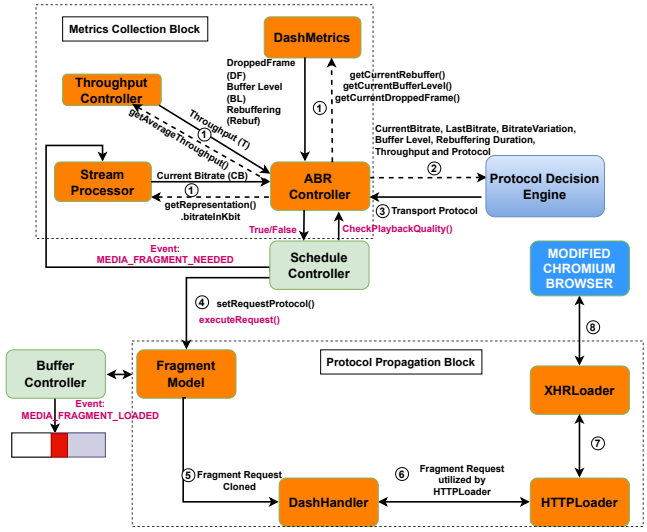


Fig. 5: Workflow of Modified DASH, an orange block indicates modified code blocks. The pink color text indicates the dash.js built-in functions and events.

C. Modifications to DASH Player

To address the identified challenges, we extended `dash.js` to support protocol-aware playback through two primary design enhancements and their seamless integration into the player workflow. Specifically, a **Protocol Propagation Block** embeds protocol intent as metadata at the point of request creation and propagates it consistently across `FragmentRequest.js`, `HTTPLoader.js`, and `XHRLoader.js`. This ensures that each segment request carries explicit transport information interpretable by Chromium without breaking modularity. Complementarily, a **Metrics Collection Block** leverages existing monitoring hooks (e.g., buffer events, throughput estimators) to collect QoE metrics such as bitrate, buffer level, rebuffering time, dropped frames, and throughput. These metrics are exported asynchronously to the decision engine, avoiding interference with playback or ABR logic.

The overall workflow of the modified player, illustrated in Fig. 5, operates as follows: (1) The ABR controller retrieves QoE and network metrics from `DashMetrics`, `StreamProcessor`, and `ThroughputController`. (2) Before each segment request, the DASH player queries the decision engine. (3) The engine returns the preferred protocol (0 = HTTP/2, 1 = HTTP/3). (4) This protocol is attached as metadata to the `FragmentRequest` and propagated through `DASHHandler`, `HTTPLoader`, and `XHRLoader`, ensuring Chromium receives the correct per-request directive. This architecture enables DASH to coordinate bitrate adaptation,

per-request transport control, and QoE feedback in a modular and non-intrusive manner.

Implementation-wise, we added a new `protocol` field in `FragmentRequest.js` metadata (alongside `media_type`, `quality`, `bandwidth`, and `URL`), incorporated it in `HTTPLoader.js`, and ensured it defaults to HTTP/3 if unspecified. For audio, the protocol is automatically set as the alternate of the corresponding video protocol. The modified requests are processed by `XHRLoader.js` to issue the actual HTTP transactions. We also extended `ABRController.js` (via the `_onFragmentLoadProgress` function) to invoke `getNextSegmentProtocol`, which gathers and transmits the collected metrics asynchronously to the local decision engine. Finally, the decision engine's response updates the protocol field of the in-progress fragment via `FragmentModel.getRequest`, with `DashHandler.js` ensuring propagation before forwarding to `HTTPLoader.js`. This integration allows protocol switching decisions to occur transparently within the existing DASH workflow, maintaining compatibility with ABR and throughput estimation modules. Overall, the implementation required modifications to eight source files, totaling approximately 140 lines of code.

D. Implementation Challenges for Browser Modifications

Integrating application-driven protocol selection into Chromium was equally challenging, as its networking stack is heavily optimized for opportunistic connection establishment and speculative racing. Several architectural constraints complicated this process. **(1) Per-request protocol signaling:** Chromium's default protocol negotiation occurs during connection setup via ALPN or cached Alternative Services, functioning at the session level rather than per HTTP request. Supporting per-segment transport selection required introducing request-level flags while maintaining thread safety and avoiding concurrency issues. **(2) Deep cross-layer propagation:** Since each HTTP request traverses multiple layer: Blink (renderer), Network Service (broker), and Net stack (transport), the new `preferredProtocol` field had to be propagated reliably through these components and IPC boundaries without affecting stability or backward compatibility. **(3) Job orchestration complexity:** The `HttpStreamFactory::JobController` manages multiple concurrent jobs and cancels redundant ones once a connection succeeds. While efficient for static protocols, this conflicted with explicit transport directives, necessitating a restructuring of the orchestration logic to prevent premature cancellation while retaining fallback support. **(4) Job binding consistency:** Chromium's single-job binding model introduced risks of unbound or reused jobs when explicit protocol preferences were applied, potentially causing orphaned connections or inconsistent states in the event-driven net stack. These challenges required a systematic redesign to enable deterministic, per-request protocol selection while preserving Chromium's robustness and efficiency.

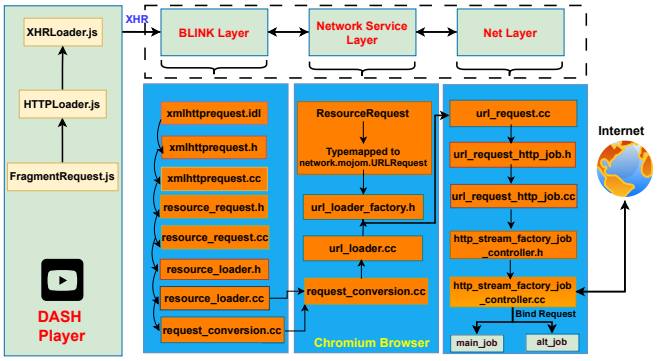


Fig. 6: Workflow of Modified Chromium, an orange bar indicates a modified code block.

E. Modifications to Chromium Browser

To enable per-request transport selection, we restructured Chromium's request pipeline to provide explicit and reliable control over the underlying transport protocol. Our design ensures that the DASH player's protocol intent is propagated end-to-end from the player through Blink, the Network Service, and into the Net stack without compromising Chromium's robustness or maintainability. Specifically, we ensure **end-to-end visibility of protocol preference** by allowing each DASH segment request to carry an explicit protocol field (TCP or QUIC), treated as primary request metadata rather than inferred through cached hints or ALPN negotiation. To achieve **deterministic yet fallback-safe orchestration**, the DASH-provided preference drives connection establishment directly, while Chromium's built-in fallback (e.g., retrying TCP if QUIC fails) remains intact for reliability. Further, our design maintains **minimal intrusion into abstractions** by confining changes to interface boundaries between major components (Blink → Network Service → Net Stack), thereby preserving backward compatibility and simplifying upstream integration. Finally, we retain a strict **separation of concerns**, where the DASH player defines policy through its decision engine, and Chromium enforces this preference within its connection management logic, maintaining modularity and avoiding cross-layer coupling.

As illustrated in Fig. 6, the modified pipeline operates as follows: (1) the DASH player annotates each segment request with a protocol hint; (2) Blink forwards this metadata transparently; (3) the Network Service Layer transmits the hint into the Net stack; and (4) within the HTTP layer (`HttpStreamFactory`), jobs are instantiated and bound deterministically according to the preference; (5) the final HTTP request is then issued over the selected transport protocol. Through this design, protocol selection becomes explicit, per-request, and application-aware, while preserving Chromium's event-driven connection management and robust failure handling.

For implementation, we extended the **Blink layer** by adding a new `protocol` attribute to `xmlhttprequest.idl` and updating `xmlhttprequest.h` and `xmlhttprequest.cc` to set the `protocolMode_`,

defined as an enum in `network.mojom`. This mode represents the preferred transport protocol and is serialized via Mojo IPC from renderer → browser → network service. In the **Network Service layer**, we modified `resource_request.h` and `HttpRequestInfo` to include a `preferredProtocol_` flag, ensuring the preference persists through inter-process boundaries into the Net stack. For `protocolMode_` = *khttp3*, this flag is set to true; for *khttp2* or *khttp1*, it is false. Within the **Net stack**, `HttpStreamFactory::JobController` was updated with an `activeProtocol` variable referencing `preferredProtocol_`, enabling the orchestration logic to honor the chosen protocol. The job creation process (`DoCreateJobs`) was modified to always create both `main_job_` (HTTP/2 over TCP) and `alternative_job_` (QUIC), removing the call to `ClearInappropriateJobs()` to prevent premature cancellation. The flag `main_job_is_blocked_` is now determined by `activeProtocol`, enabling deterministic job creation driven by DASH's decision engine. Job binding semantics (`BindJob`) were extended to enforce consistency by discarding mismatched jobs and preventing fallback to unintended protocols. Finally, Chromium's fallback mechanism remains intact, if QUIC fails (e.g., handshake timeout), the HTTP/2 job is automatically resumed, preserving robustness under dynamic network conditions.

Overall, the implementation modified 29 files across Blink, the Network Service, and the Net stack, amounting to approximately 503 lines of code. This redesign provides a stable and extensible foundation for adaptive, per-request protocol selection, ensuring reliable coordination between the DASH player and Chromium's networking architecture.

F. Implementation Details of Protocol Decision Engine

INTELSWITCH neural network architecture was implemented in Python 3.10.12 with TensorFlow 1.15.0 and TFLearn 0.3.2, running on an NVIDIA L40S GPU with CUDA 12.2, cuDNN 8, and driver version 535.247.01. For protocol decision engine training, we write 1146 lines of code. Since training and testing parameters are crucial for optimizing any neural network architecture, the parameters are summarized in Table II.

TABLE II: Decision Engine Train/Test Parameters

Parameter	Value
Discount factor	$\gamma = 0.99$
Advantage estimator	$\lambda = 0.95$
Actor and critic learning rate	$\alpha_\theta = 0.0003$, $\alpha_\phi = 0.001$
PPO clip parameter	$\epsilon = 0.2$
Batch size	$B_s = 100$
Number of epochs	4000
Number of agents	5

VII. EVALUATION SETUP

In this section, we first discuss our experimental setup in detail. We describe the network traces used for training and testing of INTELSWITCH. Next, we describe the baselines for comparison with INTELSWITCH. Finally, we present the evaluation metrics.

A. Experimental Setup:

We chose one sample video (*testsrc2*) from FFmpeg [18] originally 120 seconds long, and extended it to a total duration of 300 seconds. The video is encoded at a wide range of bitrates with 1-second segment durations: 700, 600, 500, 400, 300, 200, 100, 80, 40, 20, 10, 8, 4, 2, 1, and 0.5 Mbps. We adopt this high-bitrate encoding and 1-second chunk size based on the motivation from [19], which demonstrates the effectiveness of such settings for 5G networks. The encoded bitrate files are packaged into a DASH manifest using GPAC MP4Box, with the video framerate fixed at 24 fps throughout the process. We develop a custom video streaming client-server setup using OpenLiteSpeed server, which supports both TCP (via HTTP/2) and QUIC (via HTTP/3), utilizing the lsquic implementation of QUIC [25]. The server machine is equipped with an AMD EPYC 7543 32-core CPU and an NVIDIA L40S GPU, running Ubuntu 20.04. The client is a desktop system with an Intel Core i5-9400 6-core CPU, also running Ubuntu 20.04. All video segments, along with the MPD file, are hosted on the server. The client runs a modified `dash.js` (version 5.0.0) player with a modified Chromium browser (version 126.0.1666.0) for protocol-aware playback. We perform both replay and real-time evaluation of INTELSWITCH, described as follows:

1) Replay-Based Evaluation: For this, INTELSWITCH begins with QUIC as the default protocol for the first segment. For subsequent segments, the protocol is determined by the model's predicted action: if the model selects TCP, the corresponding sample is taken from the TCP dataset; if it selects QUIC, the sample is taken from the QUIC dataset.

2) Real-Time Adaptive Evaluation: For this, we perform live video streaming while emulating network traces using Mahimahi's `linkshell`. The DASH player issues segment requests for specific video segments and simultaneously reports metrics such as network throughput, rebuffering duration, and downloaded bitrate to the INTELSWITCH protocol decision engine. Based on these inputs, the decision engine uses the trained model to determine the next action.

B. Network Traces:

We now discuss the network traces we have used for training the INTELSWITCH protocol decision engine and for testing it in offline and online setups. We categorize our traces into controlled traces and real network traces. Real traces are obtained from publicly available datasets, whereas controlled traces are generated either by combining the real traces or by using `tc-netem` to introduce packet loss and delay.

Real-world Traces: The real-world traces include high bandwidth traces of 5G mid-band and 5G mmWave datasets from [19], [26] collected across walking and driving scenarios. The aggregate throughput for walking vs. driving is 1.6 Gbps vs. 3.2 Gbps for 5G mid-band, and 935 Mbps vs. 1.1 Gbps for 5G mmWave. For low and variable bandwidth traces, we use a low bandwidth walking trace of 4G [20], [27].

Controlled Traces: We create controlled network traces to evaluate how INTELSWITCH enhances QoE under sudden network changes. The following controlled traces are used

for this purpose: (1) High→Low (H→L): In this, it starts with around 700 Mbps bandwidth (5G mid-band equivalent) and drops to 40 Mbps and below (4G equivalent). (2) High→Low→High (H→L→H): This trace is an extension of the H→L trace, by restoring bandwidth to the 5G mid-band level after 4G, (3) High→Poor→High (H→P→H): This trace is a further extension of H→L→H, by adding 3% packet loss and 50 ms latency during the low-bandwidth phase, emulating degraded 4G walking conditions using tc netem [21]. (4) WiFi→High→Low→High (W→H→L→H): This trace depicts a scenario in which a device initially connected to WiFi, later due to mobility switches to 5G mmWave. Due to handover events, the device subsequently switches to 4G and later returns to 5G mid-band after some time when again in well-connected region.

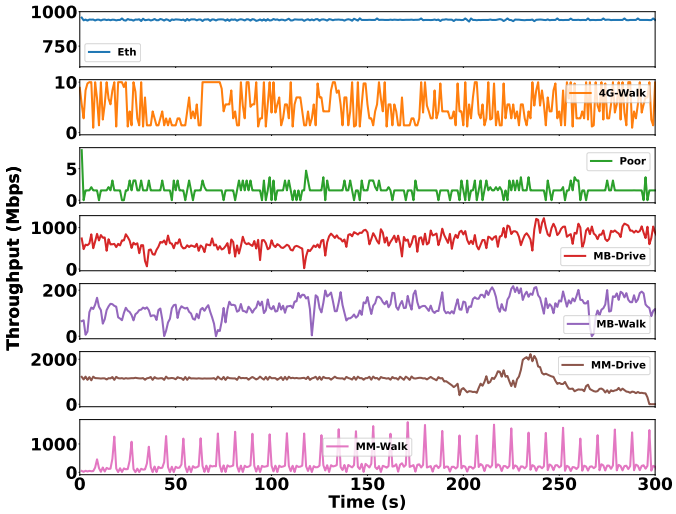


Fig. 7: Training Network Traces Throughput

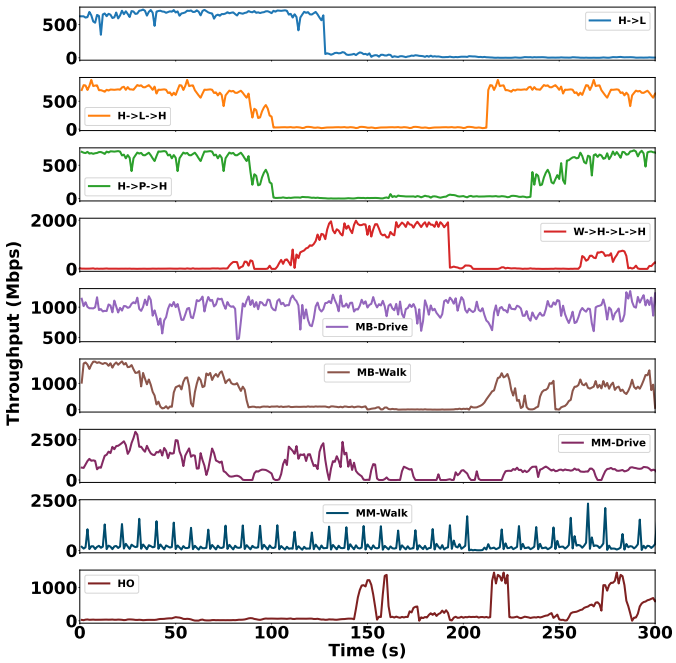


Fig. 8: Testing Network Traces Throughput

TABLE III: Network Traces

	Network Traces	Abbreviation	Network Traces	Abbreviation
Controlled	High-Low	H→L	High-low-High	H→L→H
	High-poor-High	H→P→H	WiFi-5G-4G-5G	W→H→L→H
Real	Ethernet (1 Gbps)	Eth	4G Walking	4G-Walk
	Poor & Lossy network	Poor	Handover	HO
	5G-mid-band	MB-Drive	5G-mmWave	MM-Drive
		MB-Walk		MM-Walk

Training datasets: Fig. 7 presents throughput–time plots of our training datasets. We train INTELSWITCH's protocol decision engine on real traces. Since the original 5G dataset from [19] is large in size, we partitioned it into separate files, each covering 320 seconds of data. One of the partitioned files was used for training, and the remaining files were reserved for testing. We converted each of the real traces into the Mahimahi format. The training traces are: MM-Drive, MM-Walk, MB-Drive, MB-Walk, Ethernet, and Poor. The Ethernet trace represents a 1Gbps link, whereas the Poor trace is emulated using `tc-netem` with a fixed bandwidth of 5Mbps, 3% packet loss, and 50 ms latency.

Testing methodology: To evaluate INTELSWITCH, we employ two complementary approaches: replay-based evaluation and real-time evaluation.

- **Replay-based evaluation:** In this approach, we use data collected from prior streaming sessions conducted over both TCP and QUIC under controlled network conditions. For example, to evaluate a scenario where the network transitions from high to low bandwidth (H → L), we first collect two separate traces: one with the video streamed over QUIC and another over TCP for the same network trace. During the replay phase, the decision engine selects the protocol (TCP or QUIC) for each video segment, and the corresponding segment information is retrieved from the appropriate trace. This allows us to simulate the adaptive behavior of INTELSWITCH in a reproducible and controlled environment while eliminating variability from live network fluctuations.
- **Real-time evaluation:** In this mode, the video is streamed live through a real network environment. The system continuously collects QoE and network metrics, which are fed into the decision engine. Based on the real-time decision, the next video segment request is issued using either TCP or QUIC. This method validates the system's ability to adapt dynamically and react to real-world network conditions in an end-to-end deployment.

Testing Datasets: We use 5G real traces (MM-Drive, MM-Walk, MB-Drive, and MB-Walk) not used for training, along with the controlled traces discussed earlier. Fig. 8 presents the throughput–time plots of them.

C. Baselines and ABR Algorithm:

We compare IntelDASH with the following baselines:

1) **Static Protocols:** We use fixed transport protocols, namely TCP and QUIC, throughout the video session and compare their performance with INTELSWITCH.
2) **HEURISTIC Solution:** We design a solution named, HEURISTIC. It checks if the application throughput is higher than $250Mbps$, then it chooses TCP as the transport protocol; otherwise, it chooses choose QUIC.

We use *BolaRule* (BOLA [28]) as the primary ABR algorithm for evaluating INTELSWITCH. Additionally, we employ *ThroughputRule* and *Dynamic* to assess INTELSWITCH's performance across different adaptation strategies.

D. Evaluation Metrics

(1) Quality of Experience (QoE): We use a popular video streaming metric used by MPC [29] and Pensieve [23], which is defined as:

$$QoE = B_t - \gamma * R_t - \beta * BV_t \quad (2)$$

where B_i is the bitrate of the segment requested at time t, R_t is the rebuffering time experienced at time t that results from downloading $segment_i$ at bitrate B, which is penalised by the maximum encoded bitrate (700Mbps) and BV_i is the bitrate variation from $segment_{i-1}$ to $segment_i$ experienced at time t and penalised with β = 1. In addition to combined QoE, we also show individual QoE metrics. For evaluation, we report the mean values computed over 10 independent runs.

(2) CPU Pressure: We use CPU pressure to monitor the overhead of our implementation. CPU pressure, as defined in the Linux PSI [22], is the proportion of time during which tasks are stalled while waiting for CPU access, reflecting contention for CPU resources. PSI distinguishes between partial stalls ("some"), when at least one task is stalled, and complete stalls ("full"), when all non-idle tasks are stalled simultaneously. Each of these is further tracked using moving averages over 10, 60, and 300 seconds ($avg10$, $avg60$, and $avg300$), which provide short, medium, and long-term views of CPU contention. We have used $avg60$ for comparison.

VIII. EVALUATION RESULTS

In this section, we present the evaluation results. First, we perform a benchmarking of INTELSWITCH to investigate if any overheads are introduced by INTELSWITCH. We then present offline and online testing evaluation results of INTELSWITCH. Finally, we present microbenchmarking of INTELSWITCH to understand the internal working of the same.

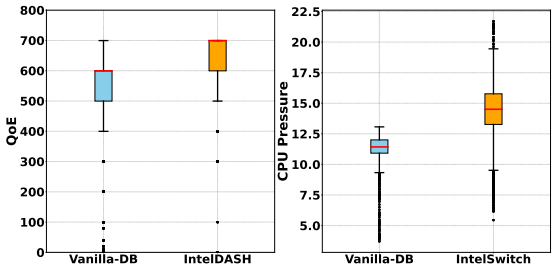


Fig. 9: INTELSWITCH implementation effect on QoE

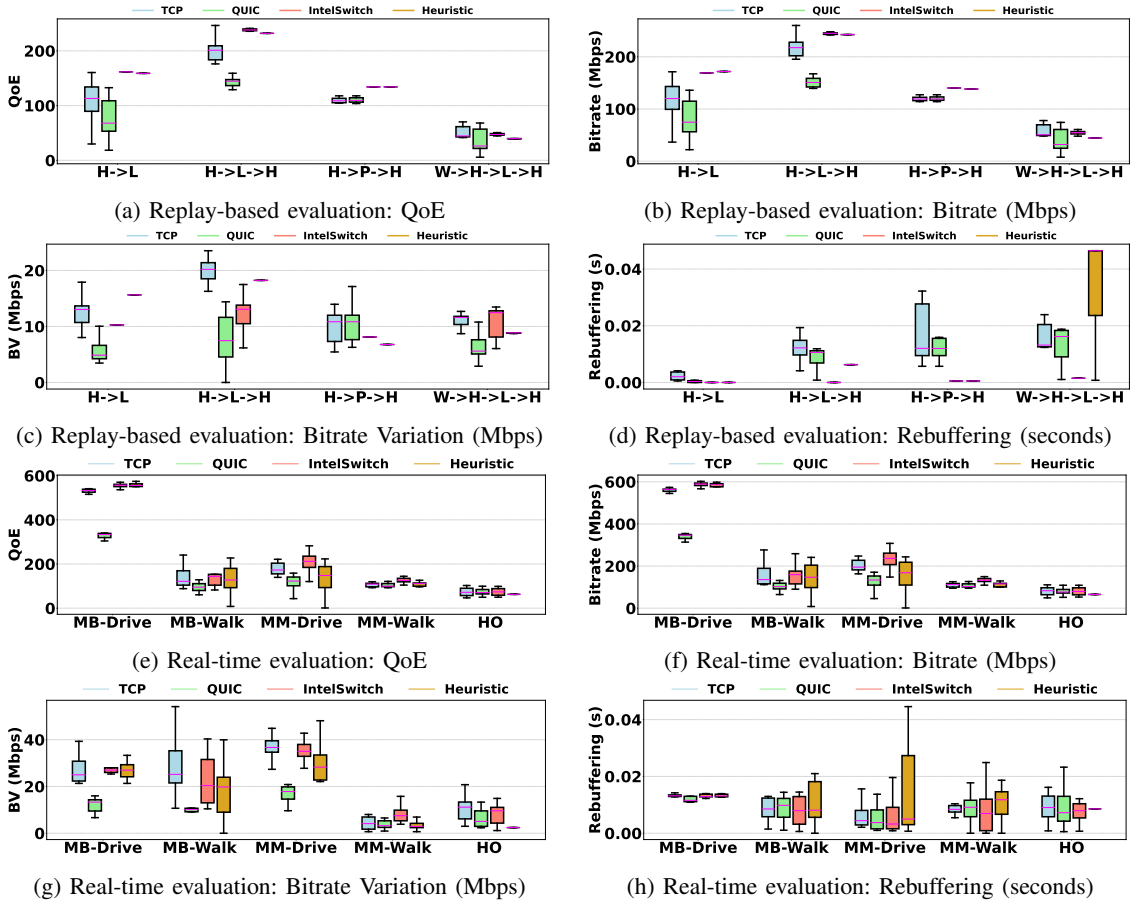
A. Benchmarking of INTELSWITCH

We compare the performance of the unmodified DASH player and browser (*Vanilla-DB*) with INTELSWITCH, which includes all enhancements described in § IV. The evaluation is performed over a stable 1 Gbps Ethernet link using TCP, chosen to isolate potential overheads introduced by INTELSWITCH without the additional processing costs of QUIC. Each setup is tested for ten iterations, and the QoE distribution is shown in Fig. 9. Even under constant TCP operation, INTELSWITCH achieves higher QoE than *Vanilla-DB*, primarily due to its separation of audio and video channels over complementary connections, which improves overall throughput and playback smoothness.

We further analyze CPU pressure to assess computational overhead. INTELSWITCH sustains a higher bitrate, slightly increasing CPU pressure from 12 to 15, owing to additional decoding and data handling. This modest increase is acceptable, as it corresponds to consistently higher video quality. Overall, these results confirm that INTELSWITCH introduces negligible processing overhead while enhancing QoE through efficient connection management and resource utilization.

B. Replay-Based Evaluation of IntelDASH

We first evaluate INTELSWITCH in replay-based experiments using the network traces listed in Table III. Each experiment uses ten iterations of TCP and QUIC video streaming data collected under controlled conditions (H→L, H→L→H, H→P→H, and W→H→L→H). As shown in Fig. 10(a), INTELSWITCH consistently achieves higher QoE than static TCP, static QUIC, and the HEURISTIC baseline. For the H→L trace, it improves QoE by 43.5%, 114.3%, and 1.7% over TCP, QUIC, and HEURISTIC, respectively. During high-bandwidth periods, both INTELSWITCH and HEURISTIC correctly select TCP, switching to QUIC as bandwidth drops. Due to the simple nature of this trace, their performance remains comparable, with INTELSWITCH making two adaptive switches. For H→L→H, INTELSWITCH provides 10.8%, 41.9%, and 2.6% higher QoE than TCP, QUIC, and HEURISTIC. Since this trace predominantly maintains high bandwidth, INTELSWITCH and HEURISTIC mostly prefer TCP, with INTELSWITCH offering a slight advantage due to its faster and more frequent switching. In the H→P→H case, which includes additional packet loss and delay, INTELSWITCH outperforms TCP and QUIC by 8.0% and 10.7%, respectively, performing as par with HEURISTIC. Both methods switch to QUIC during lossy conditions, benefiting equally from QUIC's resilience to Head-of-Line blocking. When bandwidth rises again, they return to TCP, leading to similar performance. In the W→H→L→H scenario, INTELSWITCH achieves 6.17%, 80.70%, and 19.38% gains over TCP, QUIC, and HEURISTIC, respectively. This trace highlights strong variability from WiFi to 5G to 4G and back to 5G where INTELSWITCH adapts faster and more accurately to bandwidth shifts. Unlike HEURISTIC, which reacts only when throughput crosses a fixed threshold (250 Mbps), INTELSWITCH detects and responds earlier, switching to TCP during high-bandwidth phases and to QUIC during low or lossy conditions. Consequently, it maintains smoother adaptation and higher overall QoE. As shown in Fig. 10(b)–(d), INTELSWITCH achieves higher average bitrate with fewer rebuffering and bitrate variation events compared to all baselines. These replay-based results demonstrate that adaptive protocol switching enables INTELSWITCH to sustain higher QoE across diverse and dynamic network conditions.



(a) Replay-based evaluation: QoE

(b) Replay-based evaluation: Bitrate (Mbps)

(c) Replay-based evaluation: Bitrate Variation (Mbps)

(d) Replay-based evaluation: Rebuffering (seconds)

(e) Real-time evaluation: QoE

(f) Real-time evaluation: Bitrate (Mbps)

(g) Real-time evaluation: Bitrate Variation (Mbps)

(h) Real-time evaluation: Rebuffering (seconds)

Fig. 10: Comparison of QoE and individual metrics amongst INTELSWITCH, only TCP, only QUIC and HEURISTIC for Replay and Real-based Evaluation.

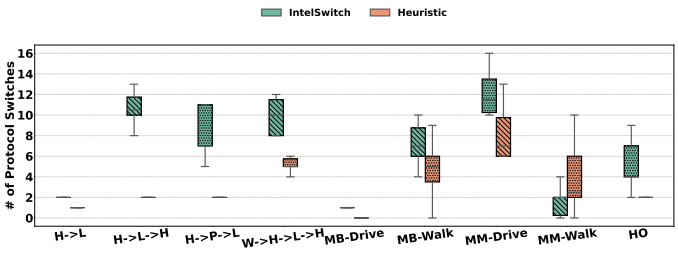


Fig. 11: INTELSWITCH and HEURISTIC number of protocol switch count for Replay and Real-based Evaluation.

C. Real-Time Evaluation of INTELSWITCH

We next evaluate INTELSWITCH in real-time across five real-world network traces, MB-Drive, MB-Walk, MM-Drive, MM-Walk, and HO. Each experiment is repeated ten times, and the results are summarized below. Overall, INTELSWITCH consistently achieves higher QoE than static TCP, static QUIC, and the HEURISTIC baseline by adapting more responsively to network variations.

For MB-Drive, where throughput remains near 1 Gbps, both INTELSWITCH and HEURISTIC stream entirely over TCP, yielding large gains over QUIC and comparable performance to each other. INTELSWITCH achieves a slight 4% improvement over TCP, reflecting minor ABR-level optimizations. In the more variable MB-Walk trace, INTELSWITCH outperforms TCP, QUIC, and HEURISTIC by 18.45%, 51.31%, and 13%, respectively. It reacts faster to bandwidth drops and recoveries, switching between protocols more effectively than HEURISTIC, which often delays switching due to fixed thresholds. This agility reduces rebuffering and stabilizes playback bitrate, resulting in smoother QoE.

In MM-Drive, which exhibits high fluctuation and frequent mmWave blockages, INTELSWITCH improves QoE by 22.55% over TCP, 72.62% over QUIC, and 43.07% over HEURISTIC. Its higher switching frequency (11 vs. 7 for HEURISTIC) allows smoother adaptation during sudden throughput drops. For MM-Walk, where average throughput stays below 250 Mbps with short spikes, INTELSWITCH surpasses TCP, QUIC, and HEURISTIC by 19.52%, 26.59%, and 13.1%, respectively. Unlike HEURISTIC, which relies on static thresholds, INTELSWITCH intelligently ignores transient spikes and makes minimal but timely switches (fewer than four), maintaining higher bitrates with comparable rebuffering. This demonstrates its robustness to short-lived fluctuations and its ability to sustain stable QoE in highly variable links.

In the HO trace, representing a 4G–5G handover, INTELSWITCH achieves modest yet consistent QoE gains: 2.8% over TCP, 3.9% over QUIC, and 5.7% over HEURISTIC. During low-bandwidth, it correctly favors QUIC, and during short

high-bandwidth spikes, it switches to TCP. With a median of four switches (vs. two for HEURISTIC), INTELSWITCH adapts faster to sudden bandwidth transitions, avoiding quality drops. Across all real-world scenarios, these results highlight that INTELSWITCH delivers smoother playback, higher bitrate, and fewer rebuffering events by dynamically selecting the optimal protocol in response to network variability.

D. Microbenchmarking of INTELSWITCH

We now perform microbenchmarking of INTELSWITCH to understand the internal working of INTELSWITCH. We first investigate when and how it decides to switch protocol. For the same, we take one of the live testing samples, i.e., we take HO (Hand Over) trace. Next, we analyze the the convergence of our protocol decision engine training. Finally, we evaluate INTELSWITCH with other ABR algorithms.

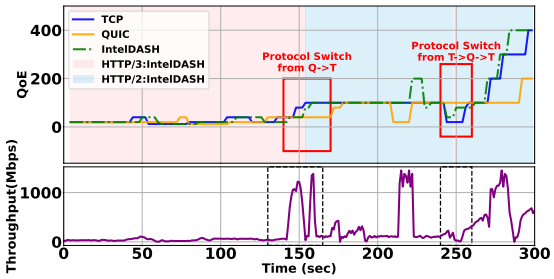


Fig. 12: INTELSWITCH switching example in HO scenario

1) Working of INTELSWITCH*:* Fig. 12 illustrates how INTELSWITCH dynamically switches between TCP and QUIC in response to changing network conditions. From $t = 0$ to $t = 130$ s, the throughput remained low but stable, leading INTELSWITCH to stream over QUIC since both protocols performed similarly. When a handover at $t = 130$ s increased bandwidth to about 900 Mbps, INTELSWITCH detected the change and switched to TCP after one segment, achieving higher bitrates (up to 400 Mbps) compared to QUIC (limited to 200 Mbps). At $t = 245$ s, the network briefly dropped to zero throughput; INTELSWITCH promptly switched to QUIC and then back to TCP as bandwidth recovered. These adaptive transitions maintained seamless playback and improved overall QoE compared to static QUIC streaming.

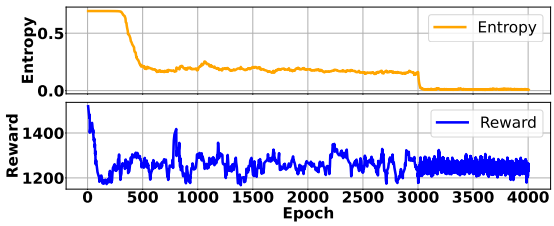


Fig. 13: Convergence time of INTELSWITCH protocol decision engine to provide optimal protocol for video streaming

2) Convergence of INTELSWITCH *Protocol Decision Engine:* Fig. 13 plots the entropy and reward with the number of epochs. One epoch consists of 100 time-steps. Initially, the reward fluctuates significantly as the agent explores different actions. Over time, the fluctuations decrease, and the reward gradually stabilizes after 3000 epochs, indicating that the agent has learned an effective policy for protocol selection. Total duration for training is 20 hours. This convergence behavior demonstrates that INTELSWITCH successfully adapts its decision-making to maximize QoE in diverse network conditions.

3) INTELSWITCH *vs. Pensieve:* Since Pensieve is primarily trained in trace-driven simulated environments, it does not generalize well to real-world DASH players. We initially integrated the pre-trained Pensieve model [27], [30] by modifying our DASH client to provide its required inputs (e.g., rebuffering, buffer level, and chunk size). However, the model consistently produced unstable and incorrect bitrate decisions. Retraining Pensieve with our own bitrate ladder under 5G, 4G, and WiFi conditions yielded similar results. These findings align with prior observations from Pensieve5G [31], which report that Pensieve performs reasonably under 3G/4G but fails in 5G due to mis-scaled normalization (e.g., higher Mbps ranges) and mismatched download-time modeling that ignores RTT effects. Pensieve's simulation-based training also assumes idealized buffer dynamics and synthetic traces, making it unsuitable for high-bandwidth, real-world networks such as 5G mid-band, mmWave, or Ethernet. Additionally, replay-based evaluation is impractical since it would require exhaustive data across all bitrate levels and protocols. Despite retraining and input adjustments, Pensieve remained unstable in our 5G environment and thus could not be reliably integrated or tested with INTELSWITCH in live experiments.

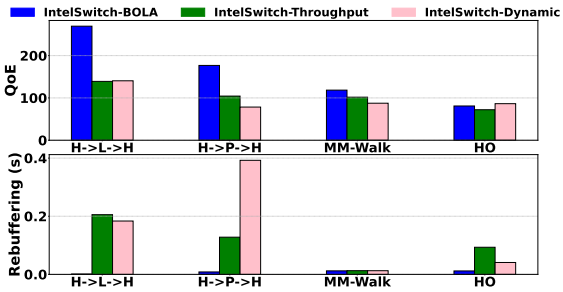


Fig. 14: INTELSWITCH comparison across different ABR

4) INTELSWITCH *with Different ABR Algorithms:* We compared the performance of INTELSWITCH using the three built-in ABR algorithms in `dash.js`: *ThroughputRule*, *BolaRule*, and *Dynamic* under real and replay-based network traces. Since each ABR responds differently to bandwidth changes, this comparison reveals which strategy best complements INTELSWITCH. As shown in Fig. 14, *ThroughputRule* and *Dynamic* struggle to maintain stability when bandwidth drops sharply, leading to frequent rebuffering. In contrast, *BolaRule* uses buffer occupancy for decisions, enabling smoother adaptation, better QoE and lower rebuffering.

IX. CONCLUSION AND FUTURE WORK

We found that the assumption that a single transport protocol can consistently outperform others across all scenarios

does not hold true. To address this limitation, we propose **INTELSWITCH**, a reinforcement learning–based adaptive transport protocol selection framework. Through an extensive evaluation involving 9 different types of networks, we demonstrate that **INTELSWITCH** significantly enhances QoE, where available bandwidth fluctuates between high and low levels. To enable this capability, we implemented modifications in both the DASH player and the Chromium browser. In our experiments, **INTELSWITCH** achieves up to 47% higher QoE compared to streaming solely over TCP, and up to 114% higher QoE compared to streaming solely over QUIC. These results suggest that widespread adoption of **INTELSWITCH** could substantially improve QoE at scale. However, real-world deployment may pose challenges, particularly in ensuring interoperability with diverse congestion control algorithms and ABR schemes. Notably, this work demonstrates adaptive transport protocol switching for video streaming using the BOLA ABR and CUBIC congestion control algorithms, with future efforts aimed at training the protocol decision engine across a wider spectrum of network conditions, congestion control mechanisms, and ABR algorithms.

ol selection framework.

REFERENCES

[1] Sandvine Incorporated, "Global internet phenomena report: 2023," Sandvine, Tech. Rep., 2023, accessed: 2025-07-20. [Online]. Available: https://www.sandvine.com/hubfs/Sandvine_Redesign_2019/Downloads/2023/reports/Sandvine%20GIPR%202023.pdf

[2] X. Zhang, S. Jin, Y. He, A. Hassan, Z. M. Mao, F. Qian, and Z.-L. Zhang, "Quic is not quick enough over fast internet," in *Proceedings of the ACM Web Conference 2024*, 2024, pp. 2713–2722.

[3] T. Shreedhar, R. Panda, S. Podanev, and V. Bajpai, "Evaluating QUIC performance over web, cloud storage, and video workloads," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 1366–1381, 2021.

[4] P. Bi, Y. Zou, M. Xiao, D. Yu, Y. Li, Z. Liu, and Q. Xie, "LiteQUIC: Improving QoE of video streams by reducing CPU overhead of QUIC," in *Proceedings of the 32nd ACM International Conference on Multimedia*, 2024, pp. 7918–7927.

[5] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196.

[6] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. Mislove, "Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols," in *proceedings of the 2017 internet measurement conference*, 2017, pp. 290–303.

[7] G. Carlucci, L. De Cicco, and S. Mascolo, "HTTP over UDP: an experimental investigation of QUIC," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015, pp. 609–614.

[8] J. Zhang, E. Dong, Z. Meng, Y. Yang, M. Xu, S. Yang, M. Zhang, and Y. Yue, "WiseTrans: Adaptive transport protocol selection for mobile web service," in *Proceedings of the Web Conference 2021*, 2021.

[9] M. Zhou, Z. Li, S. Lin, X. Wang, and Y. Chen, "FlexHTTP: an intelligent and scalable HTTP version selection system," in *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, 2022.

[10] A. Ganji and M. Shahzad, "Characterizing the performance of QUIC on android and wear OS devices," in *2021 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2021.

[11] J. Zhang, S. Ren, E. Dong, Z. Meng, Y. Yang, M. Xu, S. Yang, M. Zhang, and Y. Yue, "Reducing mobile web latency through adaptively selecting transport protocol," *IEEE/ACM Transactions on Networking*, 2023.

[12] S. Arisu and A. C. Begen, "Quickly starting media streams using QUIC," in *Proceedings of the 23rd Packet Video Workshop*, 2018, pp. 1–6.

[13] G. Szabó, S. Rácz, D. Bezzera, I. Nogueira, and D. Sadok, "Media QoE enhancement with QUIC," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2016, pp. 219–220.

[14] D. Bhat, A. Rizk, and M. Zink, "Not so QUIC: A performance study of DASH over QUIC," in *Proceedings of the 27th workshop on network and operating systems support for digital audio and video*, 2017, pp. 13–18.

[15] T. Zinner, S. Geissler, F. Helmschrott, and V. Burger, "Comparison of the initial delay for video playout start for different HTTP-based transport protocols," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 1027–1030.

[16] M. Kempf, B. Jaeger, J. Zirngibl, K. Ploch, and G. Carle, "QUIC on the fast lane: Extending performance evaluations on high-rate links," *Computer Communications*, vol. 223, pp. 90–100, 2024.

[17] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: accurate record-and-replay for HTTP," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 417–429.

[18] FFmpeg, "FFmpeg," https://ffmpeg.org/, accessed: January 14, 2026.

[19] R. A. K. Fezeu, C. Fiandrino, E. Ramadan, J. Carpenter, L. C. De Freitas, F. Bilal, W. Ye, J. Widmer, F. Qian, and Z.-L. Zhang, "Unveiling the 5G mid-band landscape: From network deployment to performance and application QoE," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 358–372.

[20] S. Chaudhary, N. Mishra, K. Gambhir, T. Rajore, A. Bhattacharya, and M. Maity, "COMPACT: Content-aware multipath live video streaming for online classes using video tiles," in *Proceedings of the 16th ACM Multimedia Systems Conference*, 2025, pp. 201–213.

[21] man7.org Linux man-pages, "tc-netem(8) — linux traffic control network emulator," https://man7.org/linux/man-pages/man8/tc-netem.8.html, 2024, accessed: 2025-07-09.

[22] J. Weiner, "PSI – Pressure Stall Information," https://docs.kernel.org/accounting/psi.html, Apr. 2018, linux Kernel Documentation.

[23] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 197–210.

[24] Z. Wu, C. Yu, D. Ye, J. Zhang, H. H. Zhuo *et al.*, "Coordinated proximal policy optimization," *Advances in Neural Information Processing Systems*, vol. 34, pp. 26 437–26 448, 2021.

[25] LiteSpeed Technologies Inc., "Openlitespeed web server," https://openlitespeed.org/, 2013, accessed: 2025-08-03.

[26] A. Narayanan, E. Ramadan, R. Mehta, X. Hu, Q. Liu, R. A. Fezeu, U. K. Dayalan, S. Verma, P. Ji, T. Li *et al.*, "Lumos5G: Mapping and predicting commercial mmwave 5G throughput," in *Proceedings of the ACM internet measurement conference*, 2020, pp. 176–193.

[27] A. Narayanan, X. Zhang, R. Zhu, A. Hassan, S. Jin, X. Zhu, X. Zhang, D. Rybkin, Z. Yang, Z. M. Mao *et al.*, "A variegated look at 5G in the wild: performance, power, and QoE implications," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 610–625.

[28] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman, "BOLA: Near-optimal bitrate adaptation for online videos," *IEEE/ACM transactions on networking*, vol. 28, no. 4, pp. 1698–1711, 2020.

[29] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over http," in *Proceedings of the 2015 ACM conference on special interest group on data communication*, 2015, pp. 325–338.

[30] A. Narayanan, X. Zhang, R. Zhu, A. Hassan, S. Jin, X. Zhu, X. Zhang, D. Rybkin, Z. Yang, Z. M. Mao, F. Qian, and Z. L. Zhang, "A variegated look at 5G in the wild: Performance, power, and QoE implications," in *Proceedings of the 2021 ACM SIGCOMM Conference (SIGCOMM '21)*, 2021, pp. 610–625, artifact repository: https://github.com/SIGCOMM21-5G/artifact.

[31] T. H. (godka), "pensieve-5G: Pensieve adaptation for 5G networks," https://github.com/godka/pensieve-5G, 2021, gitHub repository; accessed 2025-09-26.