POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Inferring Video Quality in Live Streaming Flows Using Network Passive Metrics

Supervisors
Prof. Marco Mellia
Prof. Danilo Giordano

Candidate
Giorgio Daniele Luppina

April 2025

To my father, wherever you are.

Abstract

Video streaming represents a substantial share of internet traffic, driven by the increasing demand for high-quality content and live broadcasts. This trend is particularly evident with the widespread adoption of HTTP-based adaptive bitrate streaming protocols, such as DASH and HLS. Internet Service Providers (ISPs) are often evaluated based on their customers' perceptions of premium services (e.g., video streaming), which are delivered over ISP networks by content providers like DAZN and Amazon Prime. While content providers have direct access to their customers' Quality of Experience (QoE), ISPs must infer this data from key performance indicators (KPIs) such as throughput, packet loss, and latency, especially given the growing prevalence of end-to-end encryption. This highlights the need for models capable of estimating QoE from passive metrics. In this research, we propose a methodology that leverages machine learning algorithms to infer video quality—one of the key QoE factors—using these passive metrics.

List of Contents

List of Figures

1. Introduction

Live streaming services, especially those broadcasting major sports events, have surged in popularity, attracting millions of viewers worldwide. The demand for high-quality live sports streaming has created significant challenges for service providers, as these events generate massive spikes in traffic that strain network infrastructures.

Ensuring a seamless viewing experience is crucial for retaining audiences, making Quality of Experience (**QoE**) a top priority. Achieving optimal QoE requires *minimizing buffering*, delivering *consistent video quality*, and efficiently distributing content, even under fluctuating network conditions. Technologies such as adaptive bit-rate streaming, which adjusts video quality dynamically, and Content Delivery Networks (**CDNs**), which optimize content distribution, play a vital role in meeting these demands.

As live streaming continues to evolve, QoE estimation has become essential for **ISPs** (Internet Service Providers) to ensure a high-quality viewing experience for their customers.

1.1 Motivation

Live streaming events such as sports events (e.g., the Italian Serie A, the UEFA Champions League), are particularly interesting to many users. An ISP is interested in estimating QoE factors – primarily video resolution, or how clearly the user is viewing content – their users experience while watching these events - see Figure 1.1.

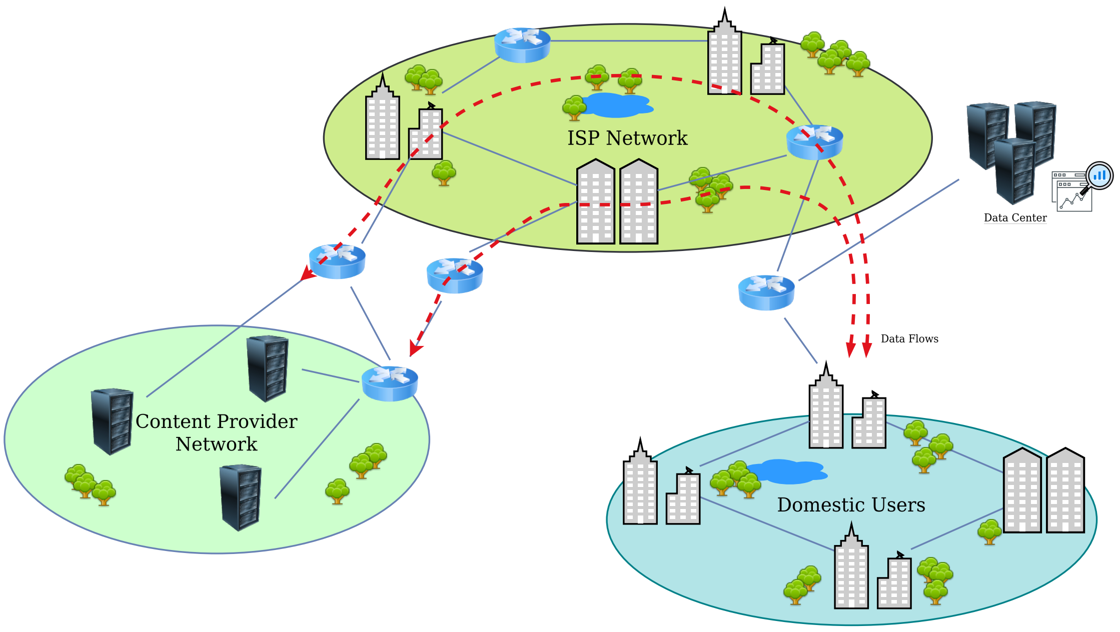


Figure 1.1: Users are interested in consuming popular video streaming services. The ISP is expected to fulfill this demand by deploying a network infrastructure that can actively support both downstream and upstream requirements for their users. For an ISP, assessing whether the infrastructure is sufficient and satisfactory for their users involves estimating the QoE their users experience

QoE encompasses *factors* like *video resolution*[1], *rebuffering frequency*[2], *startup time*[3],

[1]Higher video resolution (e.g., 4K, HD) enhances visual clarity but requires more bandwidth.

[2]Rebuffering occurs when the video pauses to load more data, negatively impacting the viewing experience.

[3]The time it takes for a video to begin playing after being requested. A longer startup time can frustrate users.

which shape the user's perception of service quality.

From a **technological perspective**, estimating and optimizing the QoE enables ISP to efficiently manage network resources. By continuously monitoring QoE, ISP can identify bottlenecks and optimize bandwidth allocation, ensuring that video streaming is delivered with minimal interruptions, even during peak traffic times.

From a **business perspective**, high QoE is essential for customer satisfaction and retention. A positive viewing experience directly impacts user loyalty, as poor streaming quality can lead to customer churn. By ensuring that users consistently enjoy high-quality streaming, ISP can maintain a competitive edge in the market.

Unlike content providers, who offer video streaming content and have direct access to video quality metrics through proprietary analytics, ISP must infer video quality solely from passive network metrics. Due to modern data encryption protocols, ISP are unable to access detailed content data transmitted over the network. This encryption, designed to protect user privacy, prevents ISP from directly observing application data, which could otherwise suggest what content the client is requesting—particularly if the content provider uses a commercial application protocol such as HTTP. As a result, ISP rely on aggregate network traffic metrics, such as *upstream* and *downstream throughput*[4], and the *impulsiveness*[5] of the data stream as their primary inputs for estimating QoE.

With video streaming traffic comprising a significant portion of global Internet traffic, ISPs are increasingly interested in defining models to infer QoE fac-

[4]Upstream and downstream throughput are computed by measuring the amount of data transferred over time, often measured in bytes per second or packets per second.

[5]Impulsiveness refers to sudden bursts or irregular patterns in the data stream, which could indicate periods of rebuffering or network congestion.

tors—particularly, in the case of this research, video resolution—using network passive metrics as input.

1.2 Related Work

Before the prevalent adoption of traffic encryption, network-based QoE monitoring solutions primarily utilized Deep Packet Inspection (DPI) to gather data on video quality metrics such as resolution, codecs, and bit-rate. [6, 13]. The shift towards encryption have forced many of these methods largely to be ineffective, leading to new research challenges concerning the estimation of QoE in the context of encrypted video traffic analysis.

Broadly, two strategies have emerged to address this issue: session-modeling-based (SM) and machine-learning-based (ML) techniques. SM-based methods depend on a thorough understanding of the streaming protocols and utilize session reconstruction to infer key performance indicators (KPIs) that impact QoE. For instance, Mangla et al. describe a solution known as eMIMIC in [9], which demonstrates its effectiveness compared to ML methods. This system reconstructs the chunk-based delivery sequence from packet traces and models a video session to estimate metrics like average bit-rate, re-buffering ratio, bit-rate fluctuations, and startup time.

However, the complexity of this problem is increased by the diversity of devices, platforms, streaming services, and applications involved (including mobile, web, and desktop), as well as variations in network types and protocols. Indeed, adapting this system for QUIC traffic could pose challenges, particularly affecting the performance of chunk detection on which it relies. Additionally, as service providers might alter their streaming protocols—whether regarding adaptation strategies, network protocols, or other factors—network operators may need to adjust their QoE estimation models accordingly. As a result, finding analytical solutions that work for many different use cases is not definitely trivial.

For these reasons, many recent studies have shifted towards using machine learning techniques for QoE estimation, arguing that these approaches may offer greater flexibility and long-term viability [3,5,8,11,15].

In both SM and ML frameworks, collecting application-level ground-truth data is essential for the training phase. This process can be straightforward for certain platforms, such as YouTube, Netflix, and Twicth, especially on browser, but can be significantly more challenging for others, like DAZN or Amazon Prime, whose absence of any APIs make more challenging recording data. This explains why we are interested in developing a system that addresses unpopular streaming services using a general methodology that can be exploited in any case. Indeed, much of the related research has focused on YouTube, partly due to the easier accessibility of application-layer data.

1.3 Problem Statement

ISPs are interested in estimating the quality of experience for their customers, particularly in the case of video streaming services, where network requirements are more restrictive than for other types of services. However, due to the widespread adoption of end-to-end encryption, it is not possible to inspect application data directly. As a result, ISPs can only measure network metrics such as bytes, packets, jitter, and round-trip time, which they can capture on the wire. Based on these metrics, they aim to predict whether the underlying application is performing well enough to ensure a satisfactory experience for customers during video playback.

1.4 Proposed Methodology

We propose a methodology for identifying live streaming data flows associated with a specific content provider and deriving features to be used as input for an ML model, following the current state-of-the-art in estimating QoE from network traffic. Our approach involves extracting relevant network metrics—those that an ISP can also obtain—and training, validating, and testing popular ML models to predict video quality, specifically video resolution.

Specifically, we propose:

(a) **Live Server Name Discovery.** A procedure to identify server names associated with a content provider that actively supports live streaming.

(b) **Inferring Video Quality from Network Passive Metrics.** A procedure for training and testing popular supervised machine learning algorithms to predict video quality (particularly, video resolution) in live streaming flows using passively collected network metrics.

2. Background

HTTP (Hypertext Transfer Protocol) has become the dominant protocol for delivering web content and media over the internet, owing to several key advantages that make it particularly well-suited for modern content delivery.

First, HTTP is highly effective in overcoming common network barriers, such as firewalls and Network Address Translation (NAT). Unlike other protocols, HTTP traffic is typically allowed to pass through firewalls without restriction, as it uses standard web ports (usually port 443 for HTTPS). This characteristic is essential for ensuring that content can be accessed across different network configurations, regardless of network security measures or location.

Additionally, HTTP integrates seamlessly with widely adopted network protocols such as the TCP and QUIC.

Another significant advantage of HTTP is its compatibility with CDNs. CDNs are distributed networks of servers designed to deliver content to end users with high efficiency and low latency. By caching video content closer to end users, CDNs reduce the distance and number of hops required to deliver media, leading to faster load times, reduced buffering, and improved overall video quality. This is especially crucial in the context of video streaming, where large amounts of data need to be transferred in real-time to ensure smooth playback.

As a result, video streaming technologies have increasingly shifted toward HTTP, making specific technologies such as RTP (Real-Time Protocol), SRTP (Secure Real-Time Protocol), RTSP (Real-Time Streaming Protocol), and MMS (Microsoft Media Services) obsolete. These older protocols, which were originally used for live video streaming, have been largely replaced by HTTP-based solutions due to their ability to adapt more effectively to varying network conditions and deliver content through scalable and efficient mechanisms like CDNs.

The widespread adoption of HTTP-based streaming protocols, such as HTTP Live Streaming (**HLS**) and Dynamic Adaptive Streaming over HTTP (**DASH**), has further contributed to the protocol's popularity.

2.1 Progressive Streaming

A preliminary implementation of video streaming over HTTP was **progressive streaming**. It involves transferring video files from a server to a client using the HTTP protocol, typically at a *fixed resolution*. Unlike traditional downloads, playback begins before the entire file is fully downloaded. Indeed, the original video is split into **chunks**, each lasting between 2 to 10 seconds of encoded video, and the client dynamically requests these chunks during playback.

2.1.1 Fetching Chunks

In a progressive streaming setup, the video is divided into small, sequential **chunks**, which are typically 2 to 10 seconds long. The video is stored on the server and divided into these chunks to facilitate easier and faster delivery.

When the client starts the video, it first requests the **manifest** file, which is usually a simple text or XML-based file that provides metadata about the video stream, such as the chunk URLs, their durations, and the sequence in which they should be fetched.

For example, a manifest file (in format .m3u8) might look like this:

```
#EXTM3U
#EXT-X-TARGETDURATION: 10
#EXT-X-MEDIA-SEQUENCE: 0
#EXTINF: 10,
http://server_hostname/live3/chunk_t=121212121.m4s
#EXTINF: 10,
http://server_hostname/live3/chunk_t=121212121.m4s
```

The manifest file is initially requested via an HTTP `GET` request by the client. Once the client receives the manifest, it begins to download the first chunk, which can then be immediately played as playback starts. The client then continues to request subsequent chunks in the correct order to allow continuous playback.

2.1.2 Playback

Once the initial chunk is downloaded, the client begins playback. As playback progresses, the client continuously fetches the next chunk in the sequence, typically requesting them in a buffered manner (ahead of the current playback position) to avoid interruptions. This pre-buffering ensures that the client has enough video data stored to continue playback smoothly, even in the event of minor fluctuations in network speed.

If playback reaches the end of a chunk and the next chunk is unavailable due

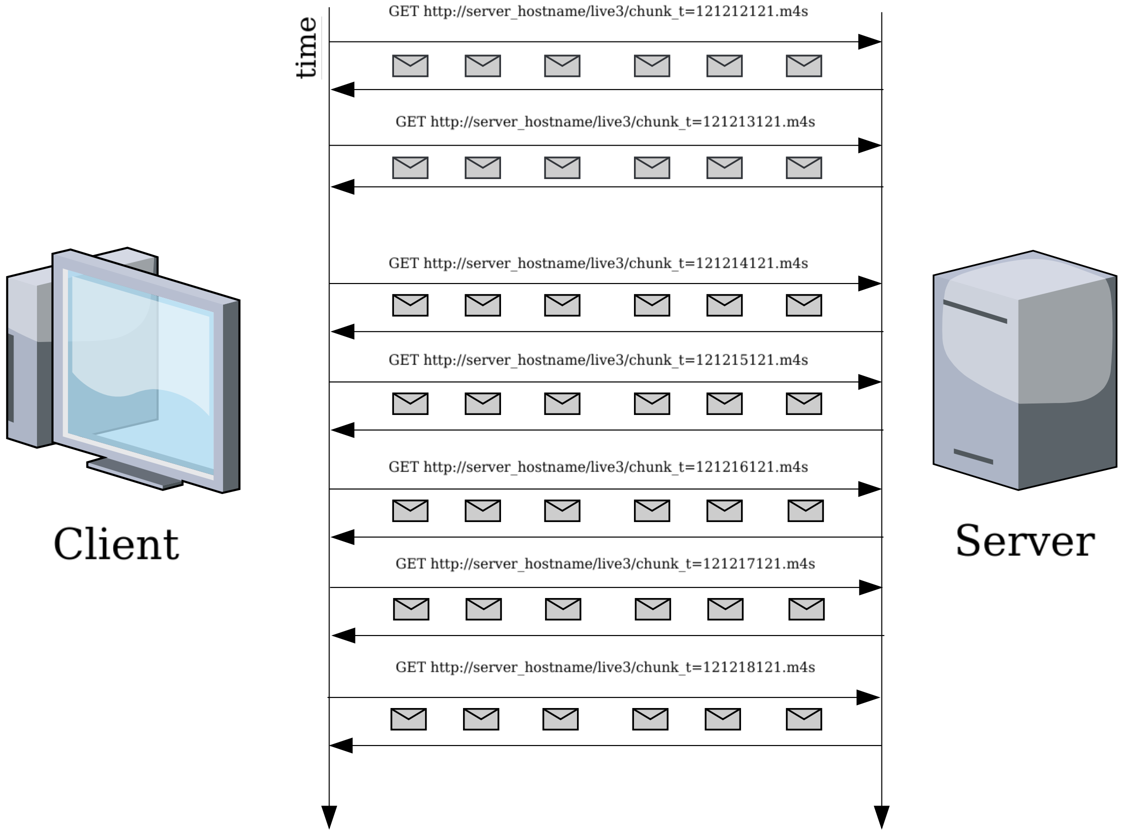

time
GET http://server_hostname/live3/chunk_t=121212121.m4s
GET http://server_hostname/live3/chunk_t=121213121.m4s
GET http://server_hostname/live3/chunk_t=121214121.m4s
GET http://server_hostname/live3/chunk_t=121215121.m4s
GET http://server_hostname/live3/chunk_t=121216121.m4s
GET http://server_hostname/live3/chunk_t=121217121.m4s
GET http://server_hostname/live3/chunk_t=121218121.m4s
Client
Server

Figure 2.1: In progressive streaming, the client selects a fixed video quality during playback and requests only chunks associated with that quality. Once a sufficient number of chunks are collected, the application buffer is filled, and the video starts. If the consumption rate exceeds the network goodput (the rate at which data is delivered to the application), the buffer depletes, causing the playback to stall

to slow network conditions, the client may experience buffering, resulting in a brief pause in playback until more data is available. While progressive streaming can help mitigate this issue by starting playback before the entire file is downloaded, slow or unstable network conditions may still lead to playback interruptions.

2.1.3 Limitations of Progressive Streaming

While progressive streaming offers a simple and efficient way of delivering video content over HTTP, it has several limitations:

1. **Fixed Video Quality:** In progressive streaming, the video is typically delivered at a single, fixed resolution and bit-rate. If the network conditions change, the video quality remains the same, and the client cannot adjust the resolution dynamically. This means that if the network connection slows down or becomes unstable, playback may stutter or pause.

2. **Buffering and Latency:** Although progressive streaming reduces the need for pre-buffering by starting playback before the entire file is downloaded, it still requires sufficient bandwidth to prevent buffering during playback. In the event of a slow or inconsistent network, buffering may occur, which can significantly disrupt the viewing experience. Additionally, since the chunks are typically of a fixed size, delays in fetching the next chunk can result in a noticeable lag between the playback and data retrieval.

3. **Inefficient Network Usage:** Progressive streaming does not make use of the sophisticated techniques available in adaptive streaming, such as the ability to cache and fetch segments at different quality levels. This results

in a less efficient use of the available bandwidth, especially in environments where the network conditions are variable.

4. **No Support for Live Content:** Although progressive streaming can work for pre-recorded content, it is not well-suited for live streaming scenarios where content is generated in real-time.

2.2 Adaptive Streaming

Modern video streaming over HTTP relies on **adaptive streaming** technologies such as HLS [4], HDS [2], and DASH [1], which represent advanced techniques compared to traditional progressive streaming. Unlike progressive streaming, which delivers video at a fixed resolution throughout the session, adaptive streaming provides a more flexible approach by adjusting video quality in real-time based on network conditions. This dynamic adjustment ensures uninterrupted playback even in the presence of fluctuating network bandwidth.

Adaptive streaming leverages multiple video **representations**, also referred to as bit-rate ladders or quality levels. Each representation is encoded at a distinct resolution and bit-rate to accommodate varying network conditions. Similar to progressive streaming, the video is divided into small segments or chunks. However, in adaptive streaming, each chunk is available in multiple quality levels, allowing the client to dynamically select the most appropriate representation based on the current network conditions.

For example, if the client detects a decrease in network bandwidth, it may switch to a lower-resolution stream, thereby reducing the data required for each chunk and ensuring smooth playback. Conversely, as network conditions improve, the client can seamlessly switch to a higher-resolution stream for better

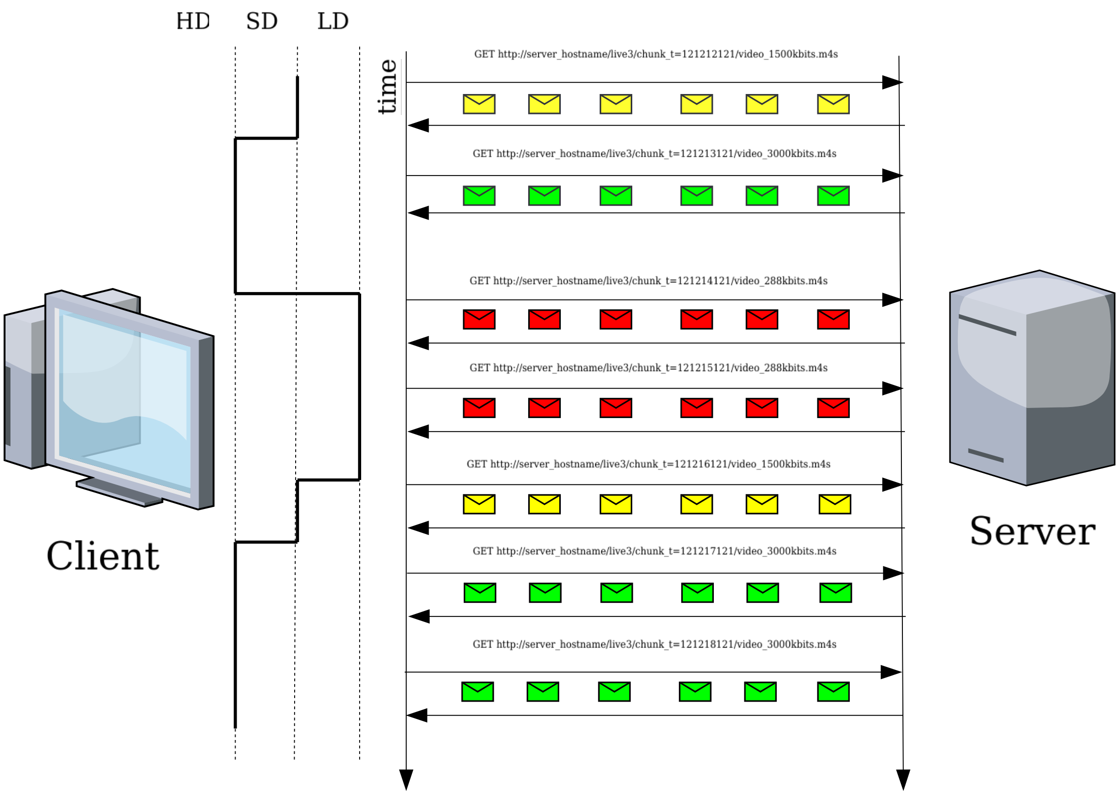

HD
SD
LD
time
GET http://server_hostname/live3/chunk_t=121212121/video_1500kbits.m4s
GET http://server_hostname/live3/chunk_t=121213121/video_3000kbits.m4s
GET http://server_hostname/live3/chunk_t=121214121/video_288kbits.m4s
GET http://server_hostname/live3/chunk_t=121215121/video_288kbits.m4s
GET http://server_hostname/live3/chunk_t=121216121/video_1500kbits.m4s
GET http://server_hostname/live3/chunk_t=121217121/video_3000kbits.m4s
GET http://server_hostname/live3/chunk_t=121218121/video_3000kbits.m4s
Client
Server

Figure 2.2: In adaptive streaming, the client initially selects a video quality but can change it during playback. An engine within the application continuously monitors whether the consumption rate exceeds the network goodput (the rate at which data is delivered to the application). If necessary, the engine prevents buffer stalls by lowering the quality of subsequent chunks, passing from HD (High-Definition) to SD (Standard Definition), and eventually LD (Low-Definition)

video quality.

An example of manifest file (in format .m3u8) for adaptive streaming might look like this:

```
#EXTM3U
#EXT-X-TARGETDURATION: 10
#EXT-X-MEDIA-SEQUENCE: 0
#EXT-X-STREAM-INF: BANDWIDTH=800000,RESOLUTION=640x360
http://server_hostname/live3/chunk_t=121212121/video_800kbits.m4s
#EXT-X-STREAM-INF: BANDWIDTH=1500000,RESOLUTION=1280x720
http://server_hostname/live3/chunk_t=121212221/video_1500kbits.m4s
...
```

The adaptive streaming paradigm is commonly used in the following contexts:

- **On Demand Streaming**: Adaptive streaming enhances the traditional progressive streaming model by allowing the client to select the initial playback quality and adjust it dynamically during the session. This provides better quality in variable network conditions but comes at the cost of a more complex manifest structure and additional overhead in managing multiple video representations.
- **Live Streaming**: While progressive streaming is generally unsuitable for real-time video streaming, adaptive streaming addresses this challenge by allowing the client to periodically request updated manifest files. These manifest files provide URLs for newly generated chunks that are being streamed from the server, enabling real-time video delivery with minimal latency and buffering.

2.3 Network Protocols for Video Streaming

The two primary transport layer protocols used for video streaming are TCP and UDP, each with distinct characteristics in terms of speed, quality, and reliability.

TCP is renowned for its reliability, ensuring that data is delivered in the correct order and without errors. However, its mechanisms for error correction and retransmission can introduce delays, which may not be ideal for real-time applications like streaming.

On the other hand, UDP is faster and more lightweight because it lacks these error-checking mechanisms, but it sacrifices reliability. However, a modern transport protocol called QUIC has been developed to improve upon TCP's performance for internet communications. QUIC, which is based on UDP, provides several advantages, including reduced latency [7], enhanced congestion control, and faster connection establishment. It has been increasingly adopted to improve Quality of Experience (QoE) for end users [10], [16].

2.3.1 HTTP streaming over TCP

In this approach, HTTP messages are encapsuled within a TCP connection, using HTTP/1.1 or HTTP/2 at the application layer. One significant challenge with using TCP for streaming is **head-of-line blocking**, a performance issue where the delivery of subsequent packets is delayed if a single packet is lost or delayed. This issue can be especially problematic for video streaming, where video chunks tend to be larger than audio chunks, causing unexpected delays in playback when packets are lost. A common workaround is to use multiple open connections: one for video and another for audio. This strategy helps avoid interference between the two data streams, allowing more efficient delivery of video and audio content.

2.3.2 HTTP streaming over UDP/QUIC

If the content provider relies on QUIC (which is designed for HTTP/3), the multiplexing feature is inherently supported. QUIC was specifically designed to handle multiplexed streams efficiently, meaning that the client no longer needs to open multiple sockets for video and audio streams. This reduces complexity, improves connection speeds, and minimizes latency, all while maintaining the ability to handle packet loss in a more graceful manner compared to TCP.

3. Tools

This chapter provides an overview of the software, tools, and experimental testbed used to conduct the experiments and collect data for this research. Specifically, in Sections 3.1 and 3.2, we describe the two main software tools utilized for data acquisition. In Section 3.3, we detail the equipment used for gathering content provider data at desktop installation.

3.1 Tstat

`Tstat` [12] processes standard `.pcap` trace files as input and reconstructs network flows for protocols including TCP, UDP, DNS, and HTTP. For the purposes of this research, the focus is primarily on the reconstruction of TCP and UDP flows. These flows are presented in text files, with each file containing statistics for both directions of communication: from client to server and vice versa. Specifically, the following log files are generated:

(a) **`log_tcp_complete`**: This document provides a comprehensive overview of all data exchanges that occurred over TCP. For each entry, `Tstat` details a reconstructed TCP connection, offering primarily volumetric metrics (see Table 7.1) for both client-to-server and server-to-client flows throughout the connection's entire lifespan. Given that TCP is a connection-oriented protocol, the lifespan of each connection is defined by the duration from the initiation of the SYN/SYN-ACK/ACK three-way handshake to the termination of the connection via the FIN/RST sequence.

(b) **`log_tcp_periodic`**: This document decomposes TCP flows into time bins, where each bin represents a short time interval of up to one second. For each bin, `Tstat` associates the same set of features available in the complete mode, thereby offering a more granular overview of flow impulsiveness and the dynamics of upstream and downstream data over time.

(c) **`log_udp_complete`**: This document provides an overview of all data exchanges that occurred over UDP. Each entry records a UDP socket-pair data exchange, presenting volumetric metrics (see Table 7.1) for both client-to-server and server-to-client flows throughout the entire flow's lifespan. Since UDP is a connectionless protocol, the lifespan of each UDP connection is defined as the time range between the first observed packet and the last packet seen before a timeout occurs between the socket pair.

(d) **`log_udp_periodic`**: This document decomposes UDP flows into time bins, similar to TCP.

For each reconstructed couple of flows in TCP and UDP, `Tstat` appends the server name to which the client was connected. As shown in Table 7.2, the server name can be retrieved from various application layer sources.

3.2 Streambot

Considering the following definitions:

Event

The time interval during a video streaming playback. The interval is $[t_{start}, t_{end}]$, where t_{start} is when it starts, while t_{end} when it finishes.

Supervised Experiment

An organized, repeatable, and automated method for observing 1 to N events. During the events, network packets and HTTP messages exchanged are traced, then logged into a file.

we developed a lightweight JavaScript tool called `Streambot` for running supervised experiments.

It automates browser interactions by navigating through a list of provided URLs (list of live streaming events), simulating human watching live events, and recording all HTTP requests made during the browsing session. The tooling relies on the JavaScript library `Puppeteer` for handling browser session life-cycle and URL typing, while the sub-module `puppeteer-har` is used for capturing HTTP messages exchanged by the client during web browsing in HAR format. For packet-level tracing, `Streambot` runs `Wireshark`.

3.2.1 Streambot Runtime

The data acquisition process (Figure 3.1) begins by launching a new `Wireshark` packet capture. Next, a browser instance (e.g., Google Chrome or Firefox) is opened. Once the browser is active, `Streambot` begins tracing HTTP messages at the application layer. It then cycles through a predefined list of URLs, each corresponding to a streaming period. For each URL, `Streambot` waits for a specified duration before proceeding to the next one. This cycle repeats until all URLs in the list have been processed.

Upon completion of a supervised experiment, `Streambot` stops HTTP tracing, terminates the browser session, and finally stops `Wireshark`, generating three output files:

(a) **`log_net_complete`** – a standard `.pcap` packet trace.

(b) **`log_har_complete`** – a standard `.har` trace[1].

(c) **`log_bot_complete`** – a text document that records, row by row, the start time (t_{start}) and end time (t_{end}) of each channel x, using Unix timestamps (Table 3.1).

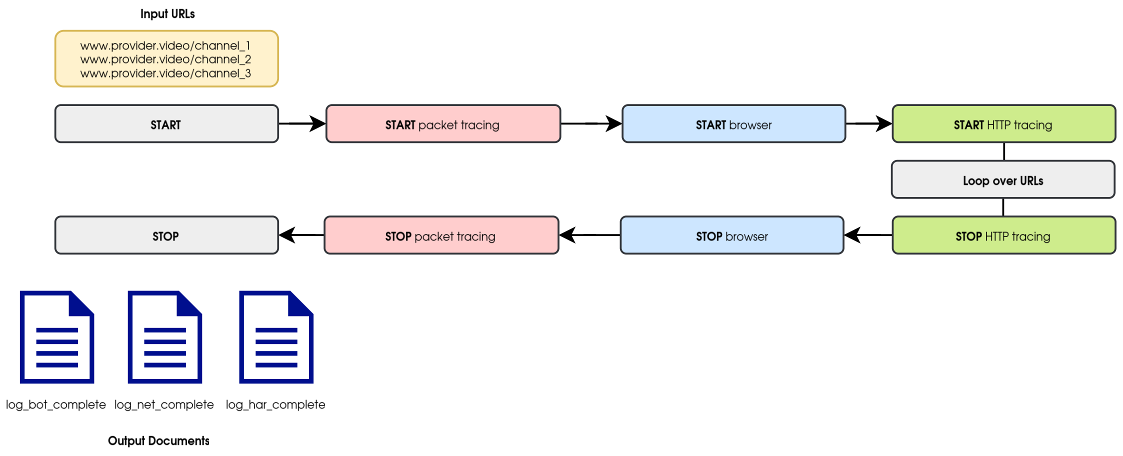


Figure 3.1: Streambot Runtime

[1]The HTTP Archive (`.har`) format is a JSON-based standard for logging interactions between web browsers and websites. It records HTTP request and response headers, metadata, and timing information, enabling performance analysis and debugging.

Channel Name	t_{start}	t_{end}
...	...	..
`golf-tv`	1739522079	1739522259
`mega-calcio`	1739522300	1739522450
`all-serie-a`	1739522600	1739522800
...	...	..

Table 3.1: In a supervised experiment, each `log_bot_complete` file records the start and end time of each channel

3.3 Testbed

The experimental testbed is designed to simulate a real-world scenario, where a domestic user, acting as the client, collects data in the last mile behind the network of the ISP, where application data is accessible. The testbed consists of a Virtual Machine (VM) running a fresh distribution of Debian 12, serving as the client. The client is then equipped with `Streambot`.

As shown in Figure 3.2, the host machine operates within a high-speed domestic LAN (Local Area Network), representing one side of the end-to-end data transfer. The LAN includes an Ethernet connection with a speed of 1 Gbps and a WiFi connection with a speed of 300 Mbps, ensuring high-speed communication for the testbed's operations.

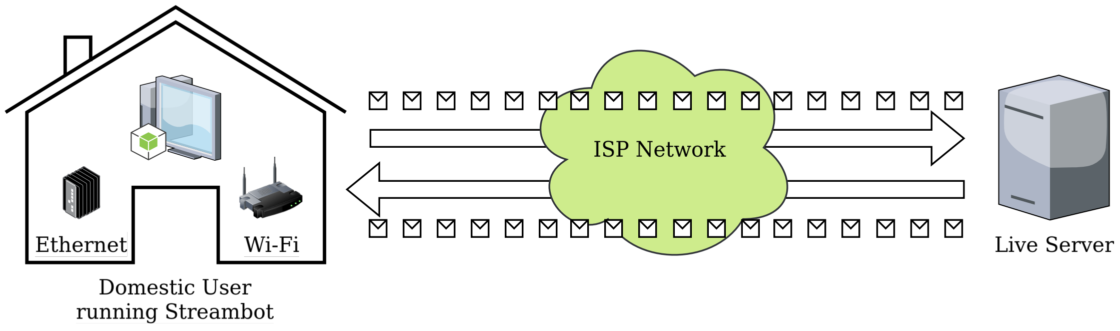


Figure 3.2: Experimental testbed used for data collection

4. Methodology

As mentioned in Section 1, the entire methodology consists of two stages:

1. **Live Server Names Discovery**: Identifying server names responsible for hosting video chunks, and therefore involved in the event, as servers from which the client effectively downloads live streaming data.
2. **Inferring Video Quality from Passive Network Metrics**: Training, validating, and testing off-the-shelf machine learning models using only passive network metrics to infer video quality (i.e., video resolution), in live video streaming flows associated with a content provider.

Tthe chapter is structured as follows:

(1) In Section 4.1, we outline the preliminary process of manifest file retrieval, which is essential for gaining comprehensive insights into the streaming deployment used by the content provider.

(2) In Section 4.2, we explain the method for preparing and consolidating raw `Streambot` data into a single, serializable data structure.

(3) In Section 4.3, we describe the methodology for identifying server names associated with a content provider, a crucial step in recognizing servers actively involved in live streaming.

(4) In Section 4.4, we present the process for training, validating, and testing machine learning classification algorithms to predict video quality using passive network metrics derived from `Tstat` periodic logs.

4.1 Gathering Manifest File

As discussed in Section 2, the manifest file assists the client during playback by helping it discover the URL of the next chunk to be requested, as well as the representation offered by the content provider.

During playback, HTTP requests for video chunks may follow patterns such as:

```
GET server.com/live-show/chunk_t=121212121/video1500kbits.m4s
GET server.com/live-show/chunk_t=121212221/video3000kbits.m4s
...
```

Alternatively, requests may follow this pattern:

```
GET server.com/live-show/chunk_t=121212121/id_profile_0.m4s
GET server.com/live-show/chunk_t=121212221/id_profile_1.m4s
...
```

In both cases, the client is expressing the chunk IDs (i.e., `chunk_t=121212121` and `chunk_t=121212221`) along with the corresponding representation. In the first case, the representation ID directly includes an indicator of video quality—the video bit-rate. In the second case, however, the representation ID is simply an anonymized string without a direct indication of video quality.

With an instance of the manifest file on hand, we can easily look up and retrieve video quality parameters (e.g., video bit-rate, video resolution, video CODEC, and so on) associated with a representation ID.

We obtain the manifest file by observing network requests in the browser's network tab. We manually search through the network requests to identify those related to the manifest, typically labeled with tokens like *manifest*, *mpd*, or *playlist*.

Once we have the manifest file, we save it and create a mapping between the string used by the client in the URL to identify a specific video chunk and its properties, including chunk bit-rate, width, height, and frame rate.

4.2 Streambot Data Consolidation

Based on the output of a supervised experiment conducted with `Streambot` (see Section 3.2), we consolidate the TCP/UDP flows and bins reconstructed by `Tstat` from `log_net_complete`, along with the HTTP requests from `log_har_complete` into JSON files, as shown in Figure 7.1. Specifically, we perform the following steps:

(a) The packet trace `log_net_complete` is input into `Tstat`, resulting in the reconstruction of TCP and UDP flows—`log_tcp_complete` and `log_udp_complete`—as well as the reconstruction of TCP/UDP flow bins—`log_tcp_periodic` and `log_udp_periodic`.

(b) From the trace `log_bot_complete`, we select all events that occurred during the experiment. Each event is represented as $[t_{\text{start}}, t_{\text{end}}]$, where t_{start} denotes the start time of the event and t_{end} denotes the end time. For each i-th event $[t_{\text{start}}, t_{\text{end}}]$, we perform the following stages:

   (i) From the traces `log_tcp_complete` and `log_udp_complete`, we select all TCP and UDP flows that overlap with the event $[t_{\text{start}}, t_{\text{end}}]$.

A TCP/UDP flow is considered overlapping if it satisfies the following condition:

$$t_s \leq t_{\text{end}} \land t_e \geq t_{\text{start}}$$

where:

- t_s is the timestamp when the flow starts
- t_e is the timestamp when the flow ends

Each selected flow (originally a space-separated record) is converted into a JSON object.

(ii) From the traces `log_tcp_periodic` and `log_udp_periodic`, we select the corresponding `Tstat` bins associated with the previously identified TCP/UDP flows in `log_tcp_complete` and `log_udp_complete`. Each selected bin (originally a space-separated record) is converted into a JSON object.

(iii) From the trace `log_har_complete`, we select all HTTP requests falling within the event $[t_{\text{start}}, t_{\text{end}}]$. For each request, we extract the following information: the host name, the URL of the requested resource, the timestamp when the request (t_{req}) was issued, the timestamp when the response (t_{res}) was received, and the MIME type. If the MIME type corresponds to one used by the content provider to encapsulate video chunks (e.g., `video/mp4`), we append the video chunk quality by identifying the bit-rate associated with that chunk using the manifest file as a lookup table.

4.3 Live Server Names Discovery

Based on content provider network, when a client interacts with remote content provider infrastructure, we expect to have:

(1) **Live Servers.** These servers are responsible for delivering video/audio chunks over TCP or QUIC connections, enabling the streaming experience. They are typically part of a CDN, which helps improve scalability, caching efficiency, and low-latency delivery. As a result, these server names often follow patterns that include identifiers like the CDN provider, the content provider's name, and words like `live` or `linear`, which indicate the server's role.[1]

(2) **Supplementary Servers.** These servers play a supporting role, handling tasks like authentication, fetching metadata (such as text, images, and stylesheets), managing DRM, or collecting telemetry data. These servers are usually centralized, managed by the content provider on-premise. Their names tend to be simpler, often just reflecting the name of the content provider as it is in domain part in the FQDN.

In order to identify the server names associated with a content provider, we follow a systematic, incremental approach that involves several key steps, as illustrated in Figure 4.1. In the following, we will outline each stage.

[1]In live streaming, server names usually have a hierarchical structure, including details like geographic location, service type, and content type. For example, a server name might have a region code (e.g., `us-east`), a service type (e.g., `live`), and a content provider tag (e.g., `netflix`). This helps with routing, load balancing, and troubleshooting within CDN systems.

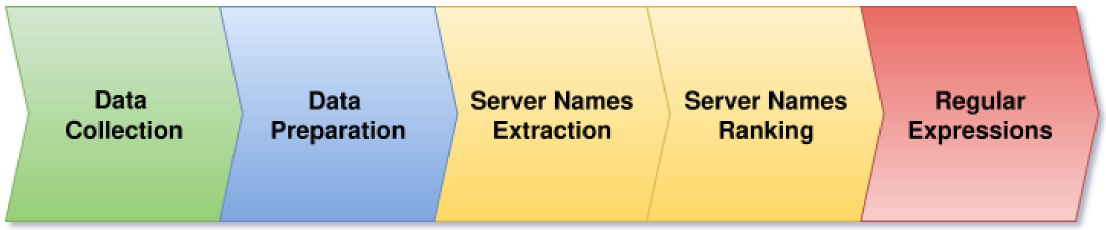


Figure 4.1: Pipeline and operational steps for discovering live streaming server names associated to a content provider

4.3.1 Data Collection

Using `Streambot`, we monitor N events, each lasting a few seconds (up to 60 seconds), covering all live channels offered by the content provider during each supervised experiment. Our primary goal is to detect when the client connects to the server. Since nearly all traffic is protected by TLS (Transport Layer Security), this typically involves intercepting the TLS handshake procedure. During this process, the client authenticates the server and selects the cipher suite to establish a secure and integrity-preserving end-to-end channel.

4.3.2 Data Preparation

Once the raw data is collected, we consolidate the TCP/UDP flows and bins, along with the HTTP requests and responses for each event into a single JSON document, as described in Section 4.2.

4.3.3 Data Processing

Server Names Extraction

For each event, we extract server names associated with TCP and UDP flows, which will serve as keys in two separate maps. The server names are extracted using the following algorithm:

(a) For TLS-based flows, the server name is identified using the Server Name Identifier (SNI) found in the `ClientHello` message, which the client uses to initiate the handshake and request the server's certificate.

(b) For non-TLS flows, the server name is extracted from the host field in HTTP responses, assuming the application layer utilizes HTTP.

(c) If neither of the above methods is applicable, the Fully Qualified Domain Name (FQDN) is used as the server name identifier, which corresponds to the server's name as recorded in the DNS.

(d) In cases where none of the aforementioned methods can be applied, the flow is discarded and not processed further.

Server Names Ranking

The server names extracted from these flows serve as keys in the following maps:

(a) **Server-Request-Frequency-and-Data-Volume**. This map tracks each server name, showing how often it appears in an event and how many bytes the client has downloaded from it.

(b) **Server-to-Video-Request-Count**. This map records how many video requests (e.g., `video/mp4`) the client has made to each server. As a result, it clearly identifies which server names are likely serving multimedia content during the event, making it easy to pinpoint video-serving servers.

4.3.4 Data Post-processing

Trevisan et al. [14] demonstrated that using regular expressions to filter TCP and UDP flows based on well-defined patterns in server names has proven to be an effective approach for identifying network traffic associated with specific public entities, such as Spotify and Facebook. Therefore, the last step involves manually creating **regular expressions** to match server names linked to the content provider, following this procedure:

(a) By analyzing the *Server-Request-Frequency-and-Data-Volume* map, we can easily determine which server names have received at least one request for video, thereby identifying those associated with live streaming.

(b) By sorting the *Server-Request-Frequency-and-Data-Volume* map according to volume data, server names associated with video delivery flows will rank highest.

These two results help us identify an **alphabetical patterns** for:

a) Server names circumstantially linked to a content provider during an event, revealing the event's activity within the web application life-cycle.

b) Server names that are physically associated with streaming activity (the most relevant result).

4.4 Inferring Video Quality from Network Passive Metrics

In this section, we describe the process of inferring video quality from passive network metrics using a classification model that estimates the quality of video requested during an event.

To estimate video quality based on passive network metrics, we rely on two key data sources collected after a supervised experiment: the list of `Tstat` bins associated with live streaming flows and the list of HTTP video requests made by the client.

The data sources—network metrics from `Tstat` bins and video quality indicators derived from HTTP request parsing—are processed using a sliding window approach, as illustrated in Figure 4.2. This approach segments the data into windows of a predefined duration ($winsize$). Each window captures the subset of `Tstat` bins and HTTP requests that fall within it. From the `Tstat` bins, we extract aggregated volumetric and temporal features, while from the HTTP requests, we derive the **ground truth**.

These labeled windows serve as the input data for training and evaluating supervised machine learning models. The model learns to correlate passive network features extracted from the `Tstat` bins with the corresponding video quality values obtained from HTTP tracing. During training, the model identifies patterns within the data, adjusting its internal parameters to minimize the difference between the predicted video quality and the actual ground truth. Once trained, the model can then be applied to unlabeled data, inferring video quality based on the network metrics observed, thus enabling the prediction of video quality in real-world scenarios.

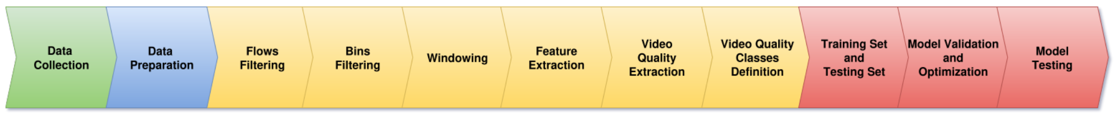


Figure 4.2: Pipeline and operational steps for training and testing a model capable of inferring video quality from network passive metrics

4.4.1 Data Collection

Using `Streambot`, we monitor N events long enough to ensure that the network connections have stabilized, overcoming the initial phase during which transitory effects (e.g., slow-start, startup) may introduce fluctuations. In practice, this means monitoring a live stream for at least 300 seconds.

Since the objective is to infer the video quality (namely the video resolution), we constrain the available bandwidth at the client side. This is approach sufficient to simulate scenarios where the most critical Quality of Service (QoS) parameter—available bandwidth—forces the bitrate adaptation algorithm to maintain video quality below the optimal level.

To impose these bandwidth restrictions, we adopt `wondershaper`[2], a command-line tool designed for network traffic shaping, which interacts at low-level with Linux `tc` network hook.

4.4.2 Data Preparation

Once the raw data is collected, we consolidate the `Tstat` flows and bins, along with the HTTP requests and responses, into a single YAML document for each

[2]`wondershaper` is a Linux command-line utility that enables bandwidth limitation by controlling maximum upload and download rates.

event, utilizing the serialization methodology outlined in Section 4.2.

4.4.3 Data Pre-processing

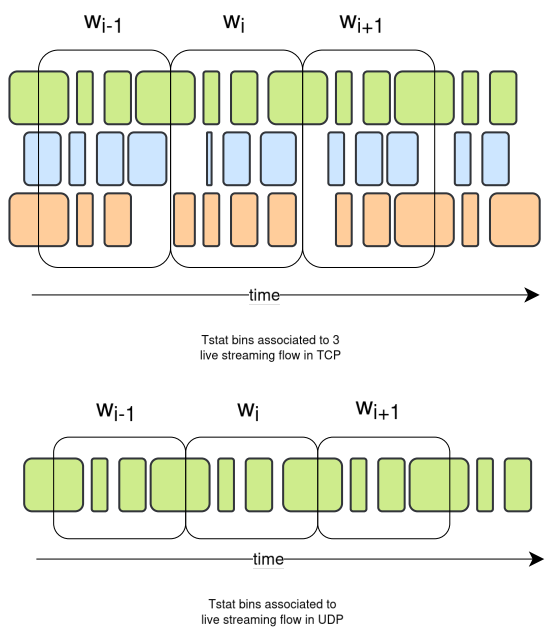

w_{i-1}
w_i
w_{i+1}
time
Tstat bins associated to 3 live streaming flow in TCP
w_{i-1}
w_i
w_{i+1}
time
Tstat bins associated to live streaming flow in UDP

Figure 4.3: Windowing procedure of live streaming flow bins. Upper case is related to TCP live streaming flows, lower case is referred to UDP/QUIC live streaming flows.

Flows Filtering For each event, we select all live streaming flows whose server name matches the regular expression we derived as result from Section 4.3.

Bins Filtering For each event, considering all selected live streaming flows, we select all associated bins.

Protocol Selection We determine the predominant protocol (TCP versus UDP) by counting the number of bins associated with selected live streaming flows. If 90% or more of the bins are in TCP live flows, we conclude that the event has been served over TCP. If 90% or more of the bins are in UDP live flows, we conclude that the event has been served over UDP.

Windowing Considering t_{start} as the timestamp when the event begins and t_{end} as the timestamp when the event ends, live streaming flow bins are segmented using a sliding-window technique, as illustrated in Figure 4.3. For each window [t_i, t_j], all bins falling within the window are considered. These bins may belong to different flows, particularly if the event is delivered over HTTP/1.1 on top of TCP, while another is delivered over HTTP/3 on top of UDP/QUIC. Some bins may be only partially contained within the current window w_i, as well as the adjacent windows w_{i+1} and w_{i-1}. In such cases, the start and end timestamps of each bin, denoted as t_s and t_e, are adjusted as follows:

$$t'_s = \max(t_s, t_i), \quad t'_e = \min(t_e, t_j)$$

where:

- t_s: original timestamp signaling the starting of the bin
- t_e: original timestamp signaling the ending of the bin
- t_i: start time of the window
- t_j: end time of the window

Feature Extraction Considering window w_i, we derive two sets of features: temporal and volumetric.

- **Temporal Features**. These features will represent the impulsiveness in live streaming flows. In particular, we derive:

$$\text{win\_idle} = winsize - \text{merge}\left(|t_{\text{last}} - t_{\text{first}}|\right) \tag{4.1}$$

  where $winsize$ is the window size in seconds, and $|t_{\text{last}} - t_{\text{first}}|$ is the overall duration across bins if they merged within the window $[t_i, t_j]$, which returns an estimation of the period of inactivity within the window.
  Then, we derive

$$\text{max\_span} = \max_{i=1}^{n} x_i \tag{4.2}$$

$$\text{min\_span} = \min_{i=1}^{n} x_i \tag{4.3}$$

$$\text{avg\_span} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{4.4}$$

$$\text{std\_span} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(x_i - \frac{1}{n}\sum_{i=1}^{n} x_i\right)^2} \tag{4.5}$$

  where $x_i = |t'_s - t'_e|$ is the duration of the i-th bin, and n is the total number of bins within the window $[t_i, t_j]$, which returns an estimation of the bin inactivity within the window.

- **Volumetric Features**. These features will represent the downstream and upstream rates observed during an event in live streaming flows. All per-bin metrics outlined in Table 7.1 are aggregated into features considering all n bins falling in the window, using the additive property as mathematical operator. On doing so, we consider proportional distribution of bytes and packets exchanged withing a bin: if the i-th bin is 30% contained within window w_i and 70% within window w_{i+1}, the generic x metric (e.g., packet count) will contribute 30% of their total to w_i and 70% to w_{i+1}, assuming a uniform distribution. In formula:

$$\mathrm{x} = \sum_{i=1}^{n} x_i \cdot \frac{|t_{\mathrm{e}} - t_{\mathrm{s}}|}{|t'_{\mathrm{e}} - t'_{\mathrm{s}}|} \tag{4.6}$$

where x denotes the generic volumetric feature from Table 7.1, x_i denotes such feature in the i-th bin within the window $[t_i, t_j]$, and n is the number of bins in the window $[t_i, t_j]$.

In Table 4.2, we report the complete list of extracted features for each window with a short description, in TCP and UDP cases.

Video Quality Extraction Considering window w_i, we derive the average video quality of the chunks requested during the event by selecting only those that fall within the window. In formula:

$$\text{avg\_video\_rate} = \frac{1}{n} \sum_{i=1}^{n} v_i \tag{4.7}$$

where v_i denotes the video bit-rate quality extracted from the URL in the i-th HTTP request, and n is the number of all HTTP requests for video falling in the window $[t_i, t_j]$.

Video Quality Classes Definition Considering that a content provider can offer several different video qualities, and that a common way to express video resolution (i.e., quality) is by measuring the number of pixels in the height of each video frame (e.g., 480p, 720p), video quality classes are defined based on the ranges of bit-rates that ensure receiving frames at low, medium, and high quality. These classes are determined by their relative distance from 720p (HD) or 1080p (Full HD), which are commercially accepted as high-quality standards. Therefore, based on the average video bit-rate computed in each window, the resulting value is classified into K discrete classes, ranging from label 0 to label $K-1$.

4.4.4 Data Processing

We evaluate and compare off-the-shelf machine learning algorithms to identify the most effective model for predicting video quality using features documented in Table 4.2 of live streaming flows. We utilize well-established implementations from the `scikit-learn` library, applying robust model validation techniques and hyperparameter optimization to achieve optimal performance. The version of `scikit-learn` used in this study is 0.24.2[3].

The classification algorithms selected for this study include **K-Nearest Neighbors (KNN)**, **Decision Tree**, and **Random Forest**. Below is a brief description of each algorithm and its relevance to this study:

- **K-Nearest Neighbors (KNN)**: KNN is a simple, instance-based learning algorithm used for classification and regression tasks. It classifies a data point based on the majority vote of its K nearest neighbors in the feature space. The distance between points is typically calculated using metrics like Euclidean distance. KNN is non-parametric, meaning it does not

[3] `scikit-learn` is a widely used Python library for machine learning, offering various tools for classification, regression, clustering, and dimensionality reduction, among others.

make assumptions about the underlying data distribution. However, it can be computationally expensive as the dataset grows, and its performance is heavily influenced by the choice of K and the distance metric.

- **Decision Tree**: A Decision Tree is a supervised learning model that splits the data into subsets based on feature values, recursively partitioning the data to minimize a certain criterion (e.g., Gini impurity or information gain for classification)
- **Random Forest**: Random Forest is an ensemble method that combines multiple Decision Trees to improve classification or regression accuracy. Each tree is built using a random subset of the data and features, and predictions from all trees are aggregated (usually by majority voting for classification tasks).

Model Validation To ensure the validity and reliability of our model evaluation, we employ the **Leave-One-Group-Out Cross-Validation (LOGO-CV)** technique, available in the `scikit-learn` library.

LOGO-CV is particularly useful when the data is organized into interdependent groups, as in our case, where each window corresponds to a specific event. This cross-validation method divides the dataset into groups, and during each iteration, one group is held out as the validation set while the remaining groups are used for model training. The application of LOGO-CV in this study guarantees that, at each fold, the data windows from a single event are never included in both the training and validation sets. This ensures that the model is evaluated on entirely unseen data, providing a more accurate estimate of its generalization performance in real-world scenarios.

Classifier	**Hyperparameters**
KNN	$n_{\text{neighbors}} \in \{20, 40, 80, 120\}$
	weights $\in$ {uniform, distance}
	metric $\in$ {euclidean, manhattan}
Decision Tree	$\text{max}_{\text{depth}} \in \{2, 3, 4, 5, 6\}$
	$\text{min}_{\text{samples\_leaf}} \in \{20, 40, 80, 120\}$
	criterion $\in$ {gini, entropy}
Random Forest	$n_{\text{estimators}} \in \{5, 10, 15, \ldots, 45\}$
	$\text{max}_{\text{depth}} \in \{2, 3, 4, 5, 6\}$
	$\text{min}_{\text{samples\_leaf}} \in \{20, 40, 80, 120\}$
	criterion $\in$ {gini, entropy}

Table 4.1: Hyperparameters used for the different models. Each row presents a set of hyperparameters specific to each model: Decision Tree, and Random Forest. The table shows the value options for each parameter used in model optimization.

Model Optimization To fine-tune our models, we apply **GridSearch**, a widely-used technique for hyperparameter optimization. For this study, we employed the evaluation metric `scikit-learn`'s default scoring options, selecting the **macro F1-score** to ensure a balanced assessment across all classes. An overview of the hyperparameter configurations tested for each algorithm is presented in Table 4.1.

Name	Description
s_bytes_all	Sum of all bytes observed across all bins in window w_i from server in TCP packets.
c_bytes_all	Sum of all bytes observed across all bins in window w_i from client in TCP packets.
s_pkts_all	Sum of all TCP packets observed across all bins in window w_i from server.
c_pkts_all	Sum of all TCP packets observed across all bins in window w_i from client.
s_ack_cnt	Sum of all TCP packets with ACK = 1 observed across all bins in window w_i from server.
c_ack_cnt	Sum of all TCP packets with ACK = 1 observed across all bins in window w_i from client.
s_ack_cnt_p	Sum of all TCP packets with exclusively ACK = 1 observed across all bins in window w_i from server.
c_ack_cnt_p	Sum of all TCP packets with exclusively ACK = 1 observed across all bins in window w_i from client.
s_pkts_data	Sum of all TCP packets with application data observed across all bins in window w_i from server.
c_pkts_data	Sum of all TCP packets with application data observed across all bins in window w_i from client.

(a) TCP Volumetric Features

Name	Description
s_bytes_all	Sum of all bytes across all bins in window w_i from server in UDP packets.
c_bytes_all	Sum of all bytes across all bins in window w_i from client in UDP packets.
s_pkts_all	Sum of all UDP packets across all bins in window w_i from server.
c_pkts_all	Sum of all UDP packets across all bins in window w_i from client.

(b) UDP Volumetric Features

Name	Description
win_idle	Idle time within the window w_i.
avg_span	Average span across all bins in the window w_i.
max_span	Maximum span across all bins in the window w_i.
std_span	Standard deviation of span across all bins in the window w_i.
min_span	Minimum span across all bins in the window w_i.

(c) TCP and UDP Temporal Features

Table 4.2: Feature set derived within a window w_i, including volumetric and temporal characteristics in live streaming flows

5. Experimental Results

In this chapter, we present the experimental results derived from network passive traces associated with DAZN activity. Specifically, we first discuss the deployment of DAZN inferred from offline observations of their usage. Then, we discuss the server names associated with this content provider. Finally, we provide numerical results on classifying DAZN live streaming flows.

5.1 DAZN Live Streaming Deployment

DAZN utilizes the DASH protocol for video delivery in the `video/mp4` format. The platform categorizes video into nine distinct profiles, as illustrated in Table 5.1. For video streaming, DAZN employs different network protocols based on the type of streaming. For TCP-based streaming, the platform uses HTTP/1.1, where multiple connections are employed to prevent head-of-line blocking and ensure seamless delivery of both audio and video. On the other hand, for UDP streaming, DAZN leverages HTTP/3 with QUIC, a protocol that provides low-latency, efficient transport for real-time data delivery.

Representation ID	Bitrate (kbps)	Width (px)	Height (px)	FPS
`video_288kbps`	288	480	270	25
`video_480kbps`	480	640	360	25
`video_840kbps`	840	640	960	25
`video_1500kbps`	1500	960	540	25
`video_2300kbps`	2300	1280	720	25
`video_3000kbps`	3000	1280	720	25
`video_4400kbps`	4400	1280	720	50
`video_6500kbps`	6500	1280	720	50
`video_8000kbps`	8000	1920	1080	50

Table 5.1: Video qualities derived from DAZN manifest file. According to DAZN's official page, the recommended network bandwidth for different resolutions on TVs, consoles, and computers is as follows: Full HD (1080p) requires at least 16 Mbps, HD (720p) requires at least 9 Mbps, and SD (below 720p) requires at least 3 Mbps

5.2 Live Server Names Discovery in DAZN

5.2.1 Data Collection

We used `Streambot` to monitor all available linear channels offered by DAZN (i.e., Eurosport1, Eurosport2, DAZN TV, Milan TV, Radio DS Serie A Enlive, PGA Tour). Overall, we collected 480 events, each lasting 60 seconds.

5.2.2 Data Processing

Supplementary DAZN servers The server names extracted from all events are summarized in Table 5.4. The server names most consistently associated with DAZN contain the keyword `dazn`, typically within the domain part. Our analysis reveals that DAZN predominantly relies on TCP as the transport layer protocol for its web-related services, while the use of UDP is minimal across the DAZN-associated entries.

Live DAZN servers As shown in Table 5.2, consulting the *Server-Request-Frequency-and-Data-Volume* map reveals clear evidence that these server names are used by the client when requesting video content during all events. More over, they account the largest portion of downloaded data from client - Table 5.3. As an additional demonstration, inspecting the headers of these server names via `curl` reveals the `EDM-Stream-Type` field, which is set to `linear` for live streaming servers. This custom header indicates that the stream is live and continuous, differentiating it from on-demand content and optimizing the delivery of real-time broadcasts. Moreover, the *Server-to-Video-Request-Count* map reveals that these server names have received requests for video.

5.2.3 Data Post-processing

To identify the presence of DAZN-related flows, a regular expression as simple as searching for DAZN is used to match domains containing the keyword, as follows:

```
.*dazn.*
```

This expression matches any string that includes `dazn`, regardless of its position within the domain.

For detecting live streaming server names, the regular expression must capture keyword `live` somewhere in name alongside DAZN's name. A suitable pattern for this task is:

```
(?=.*live)(?=.*dazn)
```

To improve the identification of server names, we use a Go-based tool called `subfinder`. This tool helps find more server names that might be linked to live streaming, allowing us to check how well the proposed regular expressions work.

Server name	**Requests for `mp4`**
`dce-ak-livedazn.akamaized.net`	4351
`dcf-de-livedazn.daznedge.net`	7193
`dcj-ac-live.cdn.indazn.com`	5100
`dce-fs-live-dazn-cdn.dazn.com`	45
`dcj-ak-livedazn.akamaized.net`	17
`dcf-fs-live-dazn-cdn.dazn.com`	2242

Table 5.2: Server names that have received requests for video during events. For each name, there is the number of requested collected across all events in the dataset

Downloaded bytes from servers operating over TCP

Server name	**Volume**	**Ratio**
`dcf-de-livedazn.daznedge.net`	29.17 GiB	86.8%
`dce-ak-livedazn.akamaized.net`	1.72 GiB	5.1%

Downloaded bytes from servers operating over UDP

Server name	**Volume**	**Ratio**
`dce-ak-livedazn.akamaized.net`	19.93 GiB	60.1%
`dcf-fs-live-dazn-cdn.dazn.com`	8.85 GiB	26.6%
`dcj-ac-live.cdn.indazn.com`	4.23 GiB	12.7%

Table 5.3: Decomposition of downloaded bytes across all events in the dataset according to server name

Server names that use TCP

Name	Frequency	Downloaded Bytes
...	...	...
`api.playback.indazn.com`	469	11,419,204
`www.dazn.com`	477	1,443,918,938
`pkg.fe.indazn.com`	470	7,261,977
`rails.discovery.indazn.com`	469	8,395,799
`static.dazndn.com`	477	1,343,934
`resource-strings.acc.indazn.com`	475	17,077,141
`event.discovery.indazn.com`	469	21,376,756
`dcf-de-livedazn.daznedge.net`	282	31,319,597,061
`drm.playback.indazn.com`	466	6,580,945
`onboarding.indazn.com`	469	10,446,504
`images.discovery.indazn.com`	476	906,436
`image.discovery.indazn.com`	478	72,728,477
`search.discovery.indazn.com`	410	551,998
`startup.core.indazn.com`	478	37,919,248
...	...	...

Server names that use UDP

Name	Frequency	Downloaded Bytes
`rs.eu1.fullstory.com`	477	13,919,224
`api.rlcdn.com`	474	3,095,689
`www.google.it`	473	2,695,326
`imasdk.googleapis.com`	46	9,223,711
`safe.dazn.com`	475	3,234,665
`fundingchoicesmessages.google.com`	474	42,467,221
`dce-ak-livedazn.akamaized.net`	119	21,396,600,740
`dcj-ac-live.cdn.indazn.com`	79	4,536,574,423
...	...	...

Table 5.4: Frequent server names identified over TCP and UDP flows during DAZN streaming events. For each name, the table presents the frequency of its occurrence during the streaming events (Frequency) and the total volume of data downloaded from that source (Volume).

Provider	Server name
Akamai	`dce-ak-livedazn.akamaized.net`
	`dcf-ak-livedazn.akamaized.net`
	...
Amazon	`dcf-ac-live.cdn.indazn.com`
	`ac-live.cdn.indazn.com`
	`dc1-ac-live2.cdn.indazn.com`
	`1535932-ac-live.cdn.indazn.com`
	`mss-ac-live.cdn.indazn.com`
	`live.dca.cdn.indazn.com`
	`dcf-ac-live.cdn.indazn.com`
	`913712-ac-live.cdn.indazn.com`
	`dcp1-ac-live.cdn.indazn.com`
	`1402144-ac-live.cdn.indazn.com`
	`dcl-ac-live.cdn.indazn.com`
	`befake1402144-ac-live.cdn.indazn.com`
	`twfake1402144-ac-live.cdn.indazn.com`
	`dc1-ac-live3.cdn.indazn.com`
...	
Google	`dcb-gc-live.cdngc.dazn.com`
	`dca-gc-livefree.cdngc.dazn.com`
	...
Fastly	`dcf-fs-live-dazn-cdn.dazn.com`
	`dce-fs-live-dazn-cdn.dazn.com`
	...
Daznedge	`dce-de-livedazn.daznedge.net`
	`dcf-de-livedazn.daznedge.net`
	`dcj-de-livedazn.daznedge.net`
	`dcaos-de-livedazn.daznedge.net`
	...

Table 5.5: Server names associated with streaming activity discovered using `subfinder`

5.3 Inferring Video Quality from Network Passive Metrics in DAZN

5.3.1 Data Collection

To generate a dataset of realistic events that are sufficiently long to be considered non-transitory, we used `Streambot` to record 300-second-long playback sessions. Since DAZN relies on both TCP and UDP, we created two separate datasets: the **TCP-dataset**, which contains only events over TCP, and the **UDP-dataset**, which contains only events over UDP. Each dataset includes the same number of events, as shown in Table 5.6.

Bandwidth (Kbit/s)	#Events
1500	25
3000	25
4500	25
6000	25
7500	25
50000	25

Table 5.6: Number of events collected for each network rate in both the TCP-dataset and the UDP-dataset

5.3.2 Data Pre-processing

We apply the operational steps discussed in Section 4.4, setting the parameter $winsize = 10$ (10 seconds) for both the TCP-dataset and the UDP-dataset.

Each event lasts 300 seconds, so with $winsize = 10$, each event is divided into 30 windows, resulting in a total of 4,500 windows. In the following, we provide a graphical walkthrough of the dataset's features, considering both the TCP-dataset and the UDP-dataset.

Features Set Visualization

In Figures 5.1 and 5.2, we observe that a reduction in available bandwidth shifts the CDF of application bytes sent from the server (`s_bytes_all`) leftward along the x-axis, for both TCP and UDP. Notably, in TCP, the application bytes represent only streaming data, while in UDP, they also include flow-control data managed by QUIC at the application layer. This shift indicates that, under reduced bandwidth, the amount of application data downloaded during streaming decreases.

We also observe a strong correlation between application bytes and application packets (`s_pkts_data` for TCP and `s_pkts_all` for UDP), likely explained by the constancy of the Maximum Segment Size (MSS).

Additionally, for TCP (Figure 5.3), there is a perfect correlation between `s_pkt_data` and `s_ack_cnt` (the number of packets with the ACK flag set to 1), suggesting that every data packet sent from the server contains an ACK, making it a *piggybacking* packet. Conversely, the number of pure ACK packets (TCP packets with only the ACK flag set) remains constant, regardless of the tested bitrate.

In Figures 5.4 and 5.5, the patterns observed for TCP and UDP differ. Application bytes sent from the client are used primarily for requesting the MPD and multimedia chunks, consisting of HTTP GET requests. The size of these requests is independent of the video quality. While the CDFs of `s_bytes_all` for TCP overlap, the presence of ACK packets in UDP (managed by QUIC and resulting in application data at the UDP layer) causes a slight separation in the CDFs.

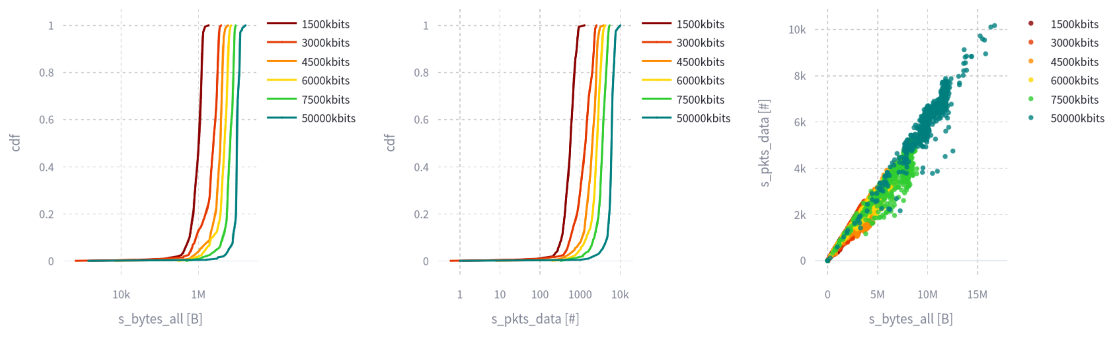


Figure 5.1: Considering all events collected in TCP dataset, from the left to right it is reported: the CDF of server bytes observed within all 10-seconds windows at different network bit-rates; the CDF of TCP packets with payload; the correlation between the two variables

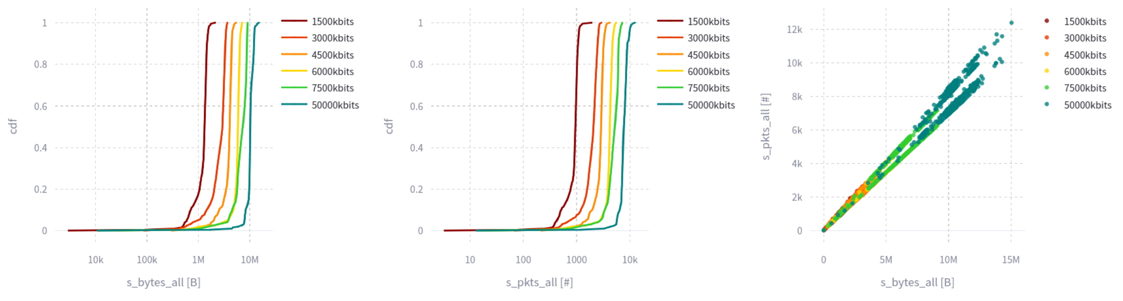


Figure 5.2: Considering all events collected in UDP dataset, from the left to right it is reported: the CDF of server bytes observed within all 10-seconds windows at different network bit-rates; the CDF of UDP packets; the correlation between the two variables

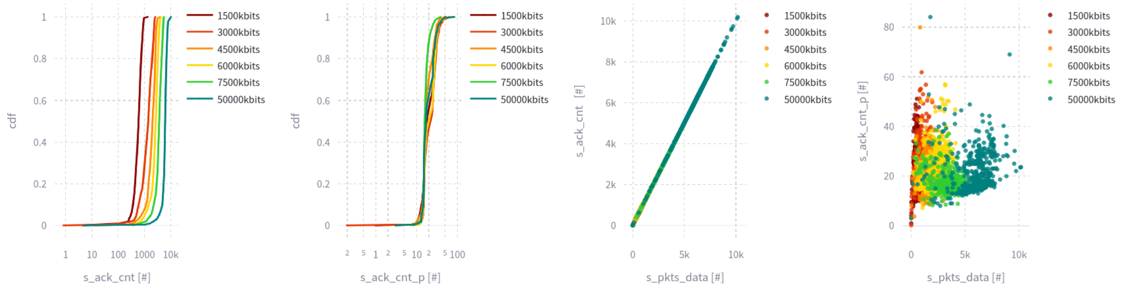


Figure 5.3: Considering all events collected in TCP dataset, from the left to right it is reported: the CDF of TCP packet with ACK = 1; the CDF of TCP packets with exclusively ACK = 1 and no data; the relationship between TCP packets with data and packets with ACK = 1

For TCP (Figure 5.6), there is no correlation between `c_pkt_data` and `c_ack_cnt`. We also observe that the CDFs of `c_ack_cnt` and `c_ack_cnt_p` are nearly identical, suggesting that most ACK packets sent by the server are pure acknowledgments for the application bytes sent.

In Figures 5.7 and 5.8, we observe that as available bandwidth increases, the CDF of `avg_idle` shifts left and the CDF of `avg_span` shifts right along the x-axis for both TCP and UDP. This shift occurs only when the bandwidth constraint is set to 50 Mbps. This behavior aligns with expectations: when the client's bandwidth is insufficient to sustain playback at the highest available quality in DAZN (or even beyond, given that DAZN requires only about 16 Mbps for Full HD playback), the ratio of utilized to idle bandwidth remains stable. The bitrate adaptation algorithm adjusts video quality based on network conditions. Only when the client's bandwidth exceeds the maximum requirement (50 Mbps) does the client experience more free bandwidth, leading to more impulsive transmission.

Feature Set Definition for DAZN

Since the feature sets of TCP and UDP do not completely overlap (e.g., ACKs are unavailable in UDP, and UDP packets carrying application data cannot be distinguished from those transporting QUIC ACKs), we focus on a shared set of features. By removing redundant features and selecting only those common to both protocols, we ensure consistency and coherence while facilitating the training and testing of a unified model applicable to both scenarios.

In Table 5.7, we present the final set of features selected for model training and evaluation.

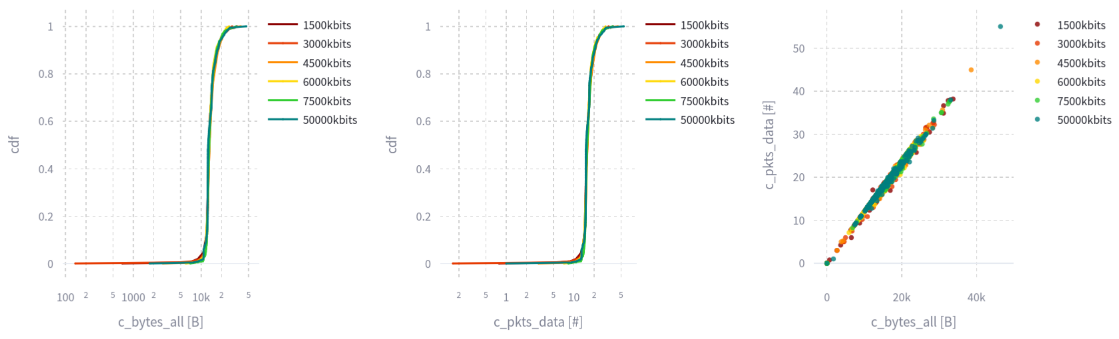


Figure 5.4: Considering all events collected in TCP dataset, from the left to right it is reported: the CDF of client bytes observed within all 10-seconds windows at different network bitrates; the CDF of TCP packets with payload; the correlation between the two variables

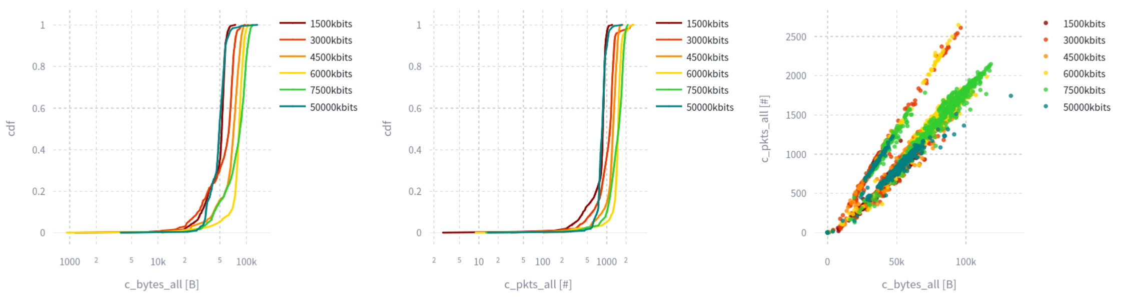


Figure 5.5: Considering all events collected in UDP dataset, from the left to right it is reported: the CDF of client bytes observed within all 10-seconds windows at different network bit-rates; the CDF of UDP packets; the correlation between the two variables

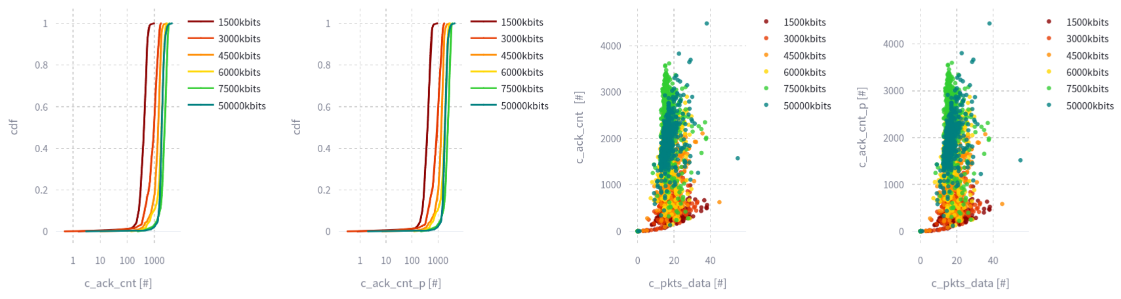


Figure 5.6: Considering all events collected in TCP dataset, from the left to right it is reported: the CDF of TCP packet with ACK = 1; the CDF of TCP packets with exclusively ACK = 1 and no data; the relationship between TCP packets with data and packets with ACK = 1

Feature	**Description**
s_bytes_all	Total number of server bytes (in TCP/UDP packets) observed in all bins within the window w_i
c_bytes_all	Total number of client bytes (in TCP/UDP packets) observed in all bins within the window w_i
win_idle	Idle time within the window w_i
avg_span	Average span across all bins in the window w_i
max_span	Maximum span across all bins in the window w_i
std_span	Standard deviation of span across all bins in the window w_i
min_span	Minimum span across all bins in the window w_i

Table 5.7: Feature Set in DAZN for Training and Testing. Volumetric features have been unified to support seamlessly both TCP and UDP scenarios

Video Quality Classes Definition

Since it is quite common to categorize the quality of something into three levels—low, medium, and high—we define three classes by setting $nclasses = 3$. This three-tier classification is widely used because it provides a balance between simplicity and granularity, making it easy to interpret while still capturing meaningful differences in quality. These classes are determined based on the average bit-rate within each window w_i, divided into three distinct ranges. Each range corresponds to a subset of video quality levels, as illustrated in Figure 5.1, and is defined according to video resolution (i.e., the number of pixels in height).

Class	Range (Kbit/s)
0	$0 \leq \texttt{video\_rate} < 1500$
1	$1500 \leq \texttt{video\_rate} < 6500$
2	$6500 \leq \texttt{video\_rate} \leq 8000$

Table 5.8: Considering video bit-rate ranges, class 0 corresponds to low-quality definition (ranging from 280p to 480p), class 1 corresponds to medium quality (ranging from 480p to 720p), and class 2 corresponds to high quality (ranging from 720p to 1080p)

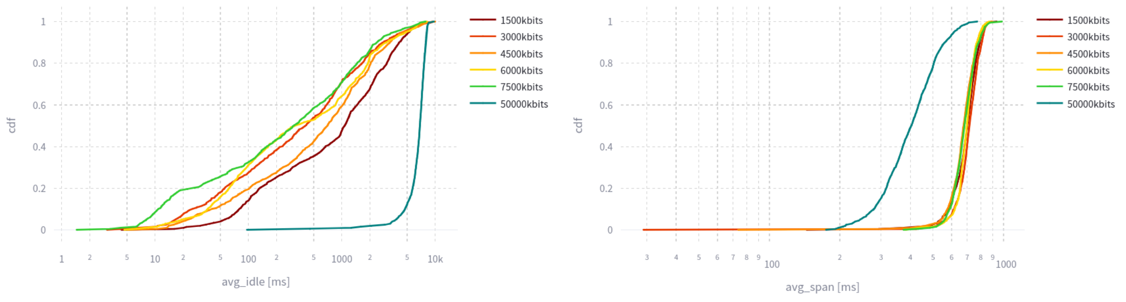


Figure 5.7: Considering all events collected in TCP dataset, from the left to right it is reported: the CDF of idless in all windows; the CDF of the average bin duration

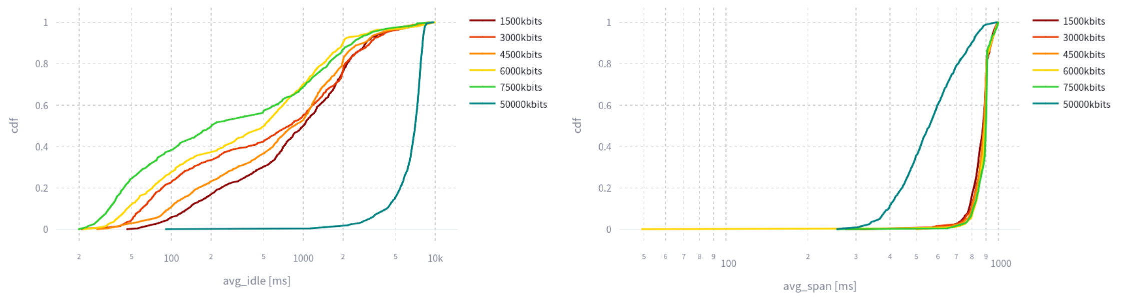


Figure 5.8: Considering all events collected in UDP dataset, from the left to right it is reported: the CDF of idless in all windows; the CDF of the average bin duration

Training and Testing Sets

For both the TCP-dataset and the UDP-dataset, we split each dataset into a training set and a testing set as follows:

- **Training Set:** The training set consists of data from 20 events per network rate. Since data is collected across multiple network rates, this results in a total of 120 events, capturing a diverse range of network conditions for model training.
- **Testing Set:** The testing set includes 5 events per network rate, totaling 30 events. This set is used to evaluate the model's performance and assess its

generalization ability on previously unseen data.

5.3.3 Data Processing

Considering $winsize = 10$ and $nclasses = 3$, we present the numerical results obtained using KNN, Decision Tree, and Random Forest. The **TCP model** refers to a model trained exclusively on the TCP-dataset, while the **UDP model** is trained solely on the UDP-dataset. Additionally, we also introduce the **MIX model**. This model is trained on the combined dataset, formed by merging the TCP-dataset and UDP-dataset.

KNN

In Table 5.12, we present the optimal hyperparameters obtained via GridSearch for each model. The classification reports for the TCP, UDP, and MIX models are shown in Tables 5.9, 5.10, and 5.11, respectively. Additionally, Figure 5.9 displays the confusion matrices for all three models.

Class	Precision	Recall	F1-Score	Support
0	0.95	0.98	0.96	222
1	0.98	0.94	0.96	495
2	0.91	0.97	0.94	179
Accuracy		0.96		896
Macro Avg	0.95	0.96	0.95	896
Weighted Avg	0.96	0.96	0.96	896

Table 5.9: Classification Report for TCP-model with KNN

Class	Precision	Recall	F1-Score	Support
0	0.92	0.94	0.93	185
1	0.97	0.94	0.96	508
2	0.94	0.98	0.96	206
Accuracy		0.95		899
Macro Avg	0.95	0.96	0.95	899
Weighted Avg	0.95	0.95	0.95	899

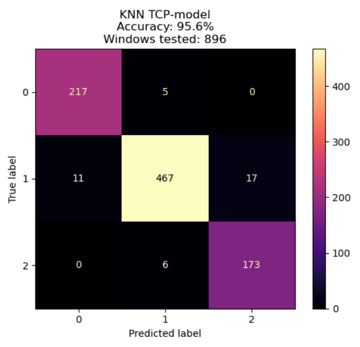
Table 5.10: Classification Report for UDP-model with KNN

Class	Precision	Recall	F1-Score	Support
0	0.91	0.91	0.91	378
1	0.95	0.94	0.94	997
2	0.93	0.97	0.95	421
Accuracy		0.94		1796
Macro Avg	0.93	0.94	0.93	1796
Weighted Avg	0.94	0.94	0.94	1796

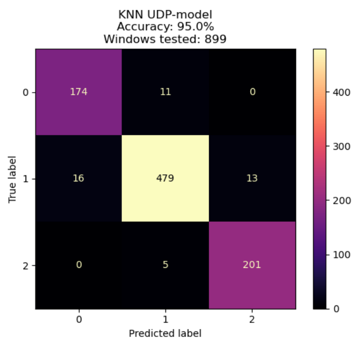
Table 5.11: Classification Report for MIX-model with KNN

Model Name	Metric	Num. Neighbors	Weights
TCP-model	manhattan	40	uniform
UDP-model	euclidean	40	uniform
MIX-model	manhattan	80	distance

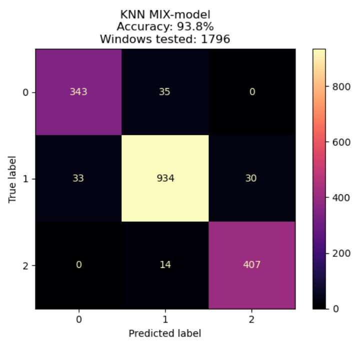
Table 5.12: Optimal Hyperparameters for KNN



(a) TCP model: Accuracy = 95.6%



(b) UDP model: Accuracy = 95.0%



(c) MIX model: Accuracy = 93.8%

Figure 5.9: Confusion matrices for KNN: TCP (left), UDP (middle), and MIX (right) models.

Decision Tree

In Table 5.16, we present the optimal hyperparameters obtained via GridSearch for each model. The classification reports for the TCP, UDP, and MIX models are shown in Tables 5.13, 5.14, and 5.15, respectively. Additionally, Figure 5.10 displays the confusion matrices for all three models.

Class	Precision	Recall	F1-Score	Support
0	0.94	0.99	0.96	222
1	0.98	0.96	0.97	495
2	0.96	0.96	0.96	179
Accuracy		0.97		896
Macro Avg	0.96	0.97	0.96	896
Weighted Avg	0.97	0.97	0.97	896

Table 5.13: Classification Report TCP-model with Decision Tree

Class	Precision	Recall	F1-Score	Support
0	0.92	0.94	0.93	185
1	0.97	0.94	0.95	508
2	0.93	0.98	0.95	206
Accuracy		0.95		899
Macro Avg	0.94	0.95	0.94	899
Weighted Avg	0.95	0.95	0.95	899

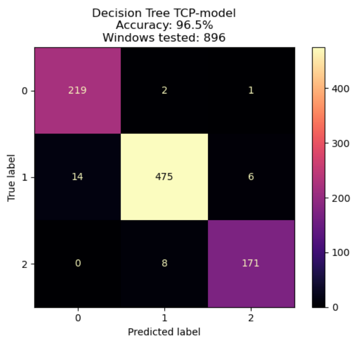
Table 5.14: Classification Report UDP-model with Decision Tree

Class	Precision	Recall	F1-Score	Support
0	0.82	0.94	0.88	378
1	0.96	0.90	0.93	997
2	0.95	0.95	0.95	421
Accuracy		0.92		1796
Macro Avg	0.91	0.93	0.92	1796
Weighted Avg	0.93	0.92	0.92	1796

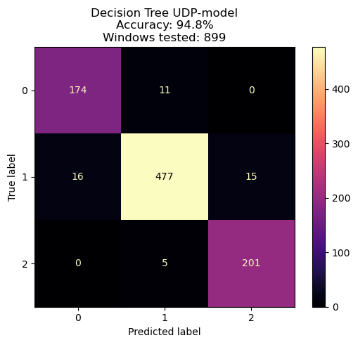
Table 5.15: Classification Report MIX-model with Decision Tree

Model Name	Max. Depth	Min. Leaves	Criterion
TCP-model	6	20	gini
UDP-model	6	20	entropy
MIX-model	6	20	entropy

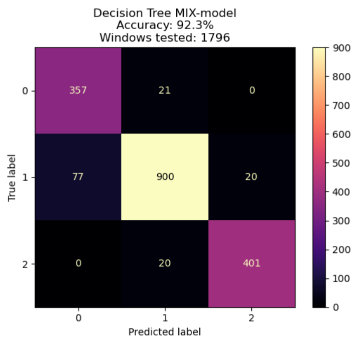
Table 5.16: Optimal hyperparameters with Decision Tree



(a) TCP model: Accuracy on testing set = 96.5%



(b) UDP model: Accuracy on testing set = 94.8%



(c) MIX model: Accuracy on testing set = 92.8%

Figure 5.10: For the Decision Tree, the confusion matrix for the TCP model is shown on the left, the confusion matrix for the UDP model is shown in the middle, and the confusion matrix for the MIX model is shown on the right.

Random Forest

In Table 5.20, we present the optimal hyperparameters obtained via GridSearch for each model. The classification reports for the TCP, UDP, and MIX models are shown in Tables 5.17, 5.18, and 5.19, respectively. Additionally, Figure 5.11 displays the confusion matrices for all three models.

Class	Precision	Recall	F1-Score	Support
0	0.95	0.97	0.96	222
1	0.95	0.96	0.96	495
2	0.96	0.89	0.93	179
Accuracy		0.95		896
Macro Avg	0.95	0.94	0.95	896
Weighted Avg	0.95	0.95	0.95	896

Table 5.17: Classification Report TCP-model with Random Forest

Class	Precision	Recall	F1-Score	Support
0	0.93	0.93	0.93	185
1	0.97	0.94	0.95	508
2	0.91	0.99	0.95	206
Accuracy		0.95		899
Macro Avg	0.94	0.95	0.94	899
Weighted Avg	0.95	0.95	0.95	899

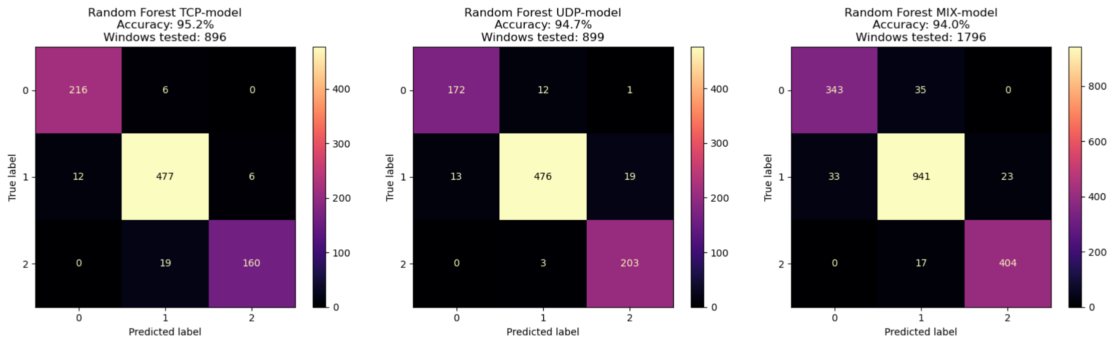
Table 5.18: Classification Report UDP-model with Random Forest

Class	Precision	Recall	F1-Score	Support
0	0.91	0.91	0.91	378
1	0.95	0.94	0.95	997
2	0.95	0.96	0.95	421
Accuracy		0.94		1796
Macro Avg	0.94	0.94	0.94	1796
Weighted Avg	0.94	0.94	0.94	1796

Table 5.19: Classification Report MIX-model with Random Forest

Model Name	Num. Trees	Max. Depth	Min. Leaves	Criterion
TCP-model	15	6	20	entropy
UDP-model	45	6	80	entropy
MIX-model	15	6	20	entropy

Table 5.20: Optimal hyperparameters with Random Forest



(a) TCP model: Accuracy on testing set = 95.2%

(b) UDP model: Accuracy on testing set = 94.7%

(c) MIX model: Accuracy on testing set = 94.0%

Figure 5.11: For the Random Forest, the confusion matrix for the TCP model is shown on the left, the confusion matrix for the UDP model is shown in the middle, and the confusion matrix for the MIX model is shown on the right.

5.3.4 Model Evaluation

Overall, we conclude that:

- **KNN** demonstrated solid performance across all models. The TCP-model achieved the highest accuracy at 95.6%, followed by the UDP-model at 95.0%, and the MIX-model at 93.8%. The confusion matrices indicate that the KNN model is generally effective, with higher recall values for the majority class (1) in each model, reflecting its sensitivity to the dominant class.

- **Decision Tree** models offered competitive accuracy, with the TCP-model performing best at 96.5%, followed by the UDP-model at 94.8%, and the MIX-model at 92.8%. The confusion matrices for the Decision Tree models indicated a strong performance, particularly for the TCP and UDP models, with high precision and recall across all classes. Additionally, the Decision Tree's interpretability provides valuable insights into feature importance, which could be leveraged for further model refinement.

- **Random Forest**, a more complex ensemble model, showed similar performance to the Decision Tree in terms of classification accuracy, with the TCP-model achieving 95.2%, the UDP-model at 94.7%, and the MIX-model at 94.0%. The Random Forest model demonstrated stability and robustness, with balanced precision and recall across the models. However, despite its higher complexity, the Random Forest model did not dramatically outperform the simpler Decision Tree, suggesting that the dataset may not require the extra complexity offered by ensemble methods.

In general, all models performed well on the task, but a key observation in the feature importance profiling, for models like Decision Tree and Random Forest, is that the feature `s_bytes_all` was identified as the most predictive feature across all models. In the case of the Decision Tree, `s_bytes_all` accounted for nearly 92% of the importance. For Random Forest, there was a more balanced distribution of importance among features, particularly `s_bytes_all`, `avg_span`, and `win_idle`. This finding aligns with the expected outcome, as reducing video quality primarily affects downstream performance, which is directly linked to the reduced amount of data that needs to be downloaded at runtime.

6. Conclusion

Video streaming has emerged as the dominant form of Internet traffic, with its share of global bandwidth usage steadily increasing in recent years. As the demand for high-quality streaming content grows, the need for effective management and optimization of streaming services becomes increasingly critical. Platforms delivering live content, particularly sports events, cause significant traffic spikes, which place considerable strain on network infrastructures. This has prompted the adoption of advanced technologies and strategies aimed at enhancing the QoE for end users.

This research experimentally demonstrates the feasibility of using regular expressions for filtering relevant network flows and passive metrics to infer video quality, with a specific focus on determining video resolution. By applying supervised machine learning algorithms, we show how passive network metrics can be leveraged for video streaming future estimations.

The results obtained from the DAZN datasets highlighted the effectiveness of this approach in classifying and predicting streaming behavior, validating the potential of this method for content providers in the streaming industry.

This research methodology, initially applied to DAZN, is also extended to Amazon Prime Video. Below, we provide an early overview of the findings.

6.1 Early Findings with Amazon Prime

Amazon utilizes the DASH protocol for video delivery in the `video/mp4` format. The platform categorizes videos into ten distinct profiles, as illustrated in Table 6.1. Amazon uses the classical TCP with HTTP 1.1 messages exchanged on top of it during the streaming process.

Representation ID	Bitrate (kbps)	Width (px)	Height (px)	FPS
`video_1_0`	1800	960	540	25
`video_2_0`	200	512	288	25
`video_3_0`	500	704	396	25
`video_4_0`	800	704	396	25
`video_5_0`	1200	960	540	25
`video_6_0`	2000	1280	720	25
`video_7_0`	4000	1280	720	25
`video_8_0`	5000	1280	720	25
`video_9_0`	6500	1280	720	25
`video_10_0`	8000	1280	720	25

Table 6.1: Video qualities derived from Amazon manifest file

For detecting live streaming server names, the regular expression must capture the keywords `live` or `linear` somewhere in the name, alongside Amazon Prime's acronyms (i.e., `aiv`, `pv`). A suitable pattern for this task is:

```
(linear|live).*(aiv|pv).*cdn.*|a.akamaihd.net
```

We applied this pattern to intercept live streaming flows in a dataset of similar size to the one used for DAZN and analyzed temporal and volumetric patterns in the live streaming traffic related to Amazon Prime's streaming service. Figures 6.1 and 6.2 present some of the early findings obtained from the data

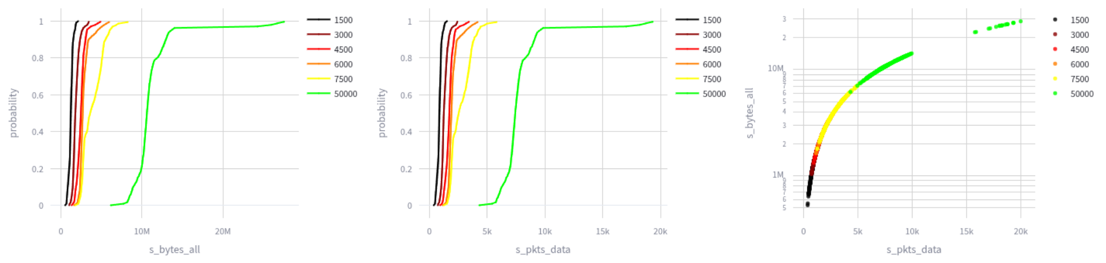


Figure 6.1: Considering all collected events, from left to right, the following is reported: the CDF of server bytes observed within all 10-second windows at different network bitrates, the CDF of TCP packets with payload, and the correlation between the two variables

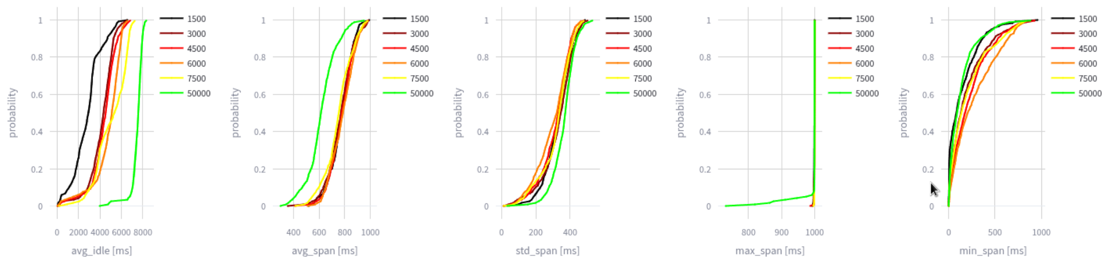


Figure 6.2: Considering all collected events from left to right, the following are reported: the CDF of idleness across all windows, the CDF of the average bin duration, the CDF of the standard deviation in bin duration, the CDF of the maximum bin duration, and the CDF of the minimum bin duration

We experimentally find similarities between features extracted from DAZN and Amazon Prime live streaming flow dynamics, suggesting that the methodology can be fully applied to Amazon Prime in the same way.

6.2 Future Work

In future work, we plan to explore the QoE factor of **rebuffering** frequency. To achieve this, we will introduce Streambot tracing activity, a functionality that is already supported but not fully tested. This will allow us to monitor the minplayer and log every instance when the video pauses. By doing so, we can define periods where the user experiences such poor network conditions that even the lowest video resolution is being used, thereby directly impacting the overall QoE.

Additionally, we intend to train and test models for Amazon Prime Video quality inference, applying the same methodology used for DAZN. Furthermore, we plan to evaluate the performance of these models using real-world network traces from an ISP, specifically Fastweb, to assess how well the models generalize to real-world conditions.

7. Appendix

Name	Description
`s_bytes_all`	Bytes from server within TCP packets.
`c_bytes_all`	Bytes from client within TCP packets.
`s_pkts_all`	TCP packets from server.
`c_pkts_all`	TCP packets from client.
`s_ack_cnt`	TCP packets from server with ACK = 1.
`c_ack_cnt`	TCP packets from client with ACK = 1.
`s_ack_cnt_p`	TCP packets from server with ACK = 1 with no payload.
`c_ack_cnt_p`	TCP packets from client with ACK = 1 witg no payload.
`s_pkts_data`	TCP packets from server with payload.
`c_pkts_data`	TCP packets from client with payload.

(a) TCP Volumetric Metrics

Name	Description
`s_bytes_all`	Bytes from server within UDP packets.
`c_bytes_all`	Bytes from client within UDP packets.
`s_pkts_all`	UDP packets from server.
`c_pkts_all`	UDP packets from client.

(b) UDP Volumetric Metrics

Table 7.1: TCP and UDP volumetric metrics available in Tstat in `log_tcp_complete` and `log_udp_complete` documents

Name	**Description**
`fqdn`	Fully Qualified Domain Name of the server, extracted from DNS request/response messages before connecting to the remote IP.
`c_SNI`	Specifies the server name with which the client is communicating, extracted from the Server Name Indication field during the TLS handshake (if the connection is TLS-protected or if UDP transports QUIC).
`http_hostname*`	Specifies the server name from HTTP headers (if communication occurs without encryption).

* HTTP hostname is available only if TCP flow reconstruction occurs.

(a) Server names in Tstat `log_tcp_complete` and `log_udp_complete`

Name	**Description**
`first`	Unix timestamp when the first TCP packet (SYN packet) is sent from client to server in a TCP connection.
`last`	Unix timestamp when the last TCP packet (RST or FIN packet) is sent from client to server (or vice-versa) in a TCP connection.
`s_first`	Unix timestamp when the first UDP packet is sent from server to client over UDP from a new socket.
`s_duration`	Duration of of the server-to-client data stream over UDP expressed in seconds.

(b) Timestamps in Tstat `log_tcp_complete` and `log_udp_complete` for time location of flows

Name	**Description**
`first`	Unix timestamp when the a TCP/UDP bins starts.
`duration`	TCP/UDP bins duration expressed in milliseconds.

(c) Timestamps in Tstat `log_tcp_periodic` and `log_udp_periodic` for time location of bins

Table 7.2: TCP and UDP metadata reconstructed by Tstat for server name identification and timestamping

```
{
  "ts": 1739522079,
  "te": 1739522259,
  "tcp_flows": [
    {
      "s_ip": "12.23.34.34",
      "c_bytes_all": 17391,
      "s_bytes_all": 172239,
      "ta": 1739522080,
      "tb": 1739522212,
      "sni": "www.playback.content-provider.com"
    }
  ],
  "udp_flows": [
    {
      "s_ip": "98.76.54.32",
      "c_bytes_all": 8291,
      "s_bytes_all": 98127,
      "ta": 1739522095,
      "tb": 1739522200,
      "sni": "www.playback.content-provider.com"
    }
  ],
  "tcp_flow_bins": [
    {
      "start": 1739522080,
      "end": 1739522090,
      "c_bytes": 4000,
      "s_bytes": 45000
    },
    ...
  ],
  "udp_flow_bins": [
    ...
  ],
  "http_reqs": [
    {
      "server": "server.com",
      "url": "/live-show/chunk_t=121212121/video1500kbits.m4s",
      "mime": "video/mp4",
      "rate": "1500kbps",
      "treq": 1739522090,
      "tres": 1739522120
    },
    ...
  ]
}
```

Figure 7.1: An instance of a JSON document containing TCP/UDP flows, flow bins, and HTTP requests for an event

Bibliography

[1] 3GPP and MPEG. Dynamic adaptive streaming over http (dash), 2012.

[2] Adobe Systems Inc. Adobe http dynamic streaming (hds), 2010.

[3] V. Aggarwal, E. Halepovic, J. Pang, S. Venkataraman, and H. Yan. Prometheus: Toward quality-of-experience estimation for mobile apps from passive network measurements. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications (HotMobile)*, page 18, 2014.

[4] Apple Inc. Http live streaming (hls), 2009.

[5] F. Bronzino, P. Schmitt, S. Ayoubi, G. Martins, R. Teixeira, and N. Feamster. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3):1–25, Dec. 2019.

[6] P. Casas, R. Schatz, and T. Hossfeld. Monitoring YouTube QoE: Is your mobile network delivering the right experience to your customers? In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1609–1614, Apr. 2013.

[7] Z. Gurel, T. E. Civelek, and A. C. Begen. Need for Low Latency: Media Over QUIC. In *Proceedings of the 2nd Mile-High Video Conference*, pages 139–140, May 2023.

[8] C. Gutterman, K. Guo, S. Arora, X. Wang, L. Wu, E. Katz-Bassett, and G. Zussman. Requet: Real-time QoE detection for encrypted YouTube traffic. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems (Internet-QoE)*, 2019.

[9] T. Mangla, E. Halepovic, M. Ammar, and E. Zegura. Using session modeling to estimate HTTP-based video QoE metrics from encrypted network traffic. *IEEE Transactions on Network and Service Management*, 16(3):1086–1099, Sept. 2019.

[10] M. Nguyen, H. Amirpour, C. Timmerer, and H. Hellwagner. Scalable High Efficiency Video Coding Based HTTP Adaptive Streaming Over QUIC. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, pages 28–34, Aug. 2020.

[11] I. Orsolic, D. Pevec, M. Suznjevic, and L. Skorin-Kapov. A machine learning approach to classifying youtube QoE based on encrypted network traffic. *Multimedia Tools and Applications*, 76(21):22267–22301, Nov. 2017.

[12] PoliTO. Tstat: Network monitoring tool. `http://tstat.polito.it`, 2009.

[13] R. Schatz, T. Hossfeld, and P. Casas. Passive YouTube QoE monitoring for ISPs. In *Proceedings of the 6th International Conference on Innovative Mobile Internet Services and Ubiquitous Computing*, pages 358–364, July 2012.

[14] M. Trevisan, D. Giordano, I. Drago, M. Mellia, and M. Munafò. Five years at the edge: Watching internet from the isp network. *IEEE/ACM Transactions on Networking*, 28(2):561–574, Apr. 2020.

[15] S. Wassermann, M. Seufert, P. Casas, L. Gang, and K. Li. I see what you see: Real-time prediction of video quality from encrypted streaming traffic. In *Proceedings of the 4th Internet-QoE Workshop on QoE-Based Analysis and Management of Data Communication Networks (Internet-QoE)*, pages 1–6, 2019.

[16] S. Xu, S. Sen, and Z. M. Mao. CSI: Inferring Mobile ABR Video Adaptation Behavior Under HTTPS and QUIC. In *Proceedings of the 15th European Conference on Computer Systems*, pages 1–16, Apr. 2020.