Stream-Level Routing over QUIC: A WebTransport Proxy Architecture for Kubernetes

Sibin George, Msc

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

Supervisor: Dr. Stefan Weber

August 2025

Stream-Level Routing over QUIC: A WebTransport Proxy Architecture for Kubernetes

Sibin George, Master of Science in Computer Science

University of Dublin, Trinity College, 2025

Supervisor: Dr. Stefan Weber

Modern web applications increasingly demand ultra-low-latency, multiplexed, and secure transport between browsers and cloud-native back-ends. While the QUIC transport protocol [1] and its application-layer companion HTTP/3 [2] eliminate head-of-line blocking and reduce handshake latency, current Kubernetes ingress controllers and reverse proxies only provide coarse-grained, connection-level load balancing. They terminate QUIC early and forward traffic over HTTP/1.1, losing the ability to route or observe individual WebTransport streams—a limitation that severely hampers real-time use cases such as interactive video, gaming, and telemetry.

This dissertation presents Custom Router Proxy, a novel WebTransport-aware proxy that operates natively inside Kubernetes [3]. By extending the Aioquic library [4], the custom router intercepts incoming QUIC packets, demultiplexes WebTransport streams containing audio, video, chat, or file data, and forwards each stream—via configuration-driven rules—to dedicated micro-services that publish processed results into Apache Pulsar topics [5]. The architecture leverages a 32-byte application header embedded in each stream to enable content-based routing at line rate while preserving end-to-end encryption.

Evaluation on a single-node Minikube cluster [6] demonstrates that the Router sustains sub-millisecond latency, linear throughput scaling up to five concurrent clients, and zero packet loss even under 90% fragmentation. The system is fully declarative: routing rules and broker endpoints are supplied through Kubernetes ConfigMaps that support hot-reloading without service restarts. These results validate the feasibility of stream-level, QUIC-native ingress and establish a reusable foundation for deploying next-generation, real-time applications on Kubernetes.

Acknowledgments

Firstly, I would like to take this opportunity to thank my supervisor, Prof. Stefan Weber, for providing me with the right guidance as I navigated and explored this vast domain. I thank you again for sparking my curiosity during each meeting we had. To my parents and elder sister, thank you for the love and support from afar; your quiet encouragement and belief in me were a constant source of strength that carried me through this master's journey.

Sibin George

University of Dublin, Trinity College
August 2025

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The world of web communication has changed tremendously over the last couple of decades particularly because of growing demand for instance access to information. This results in need for ultra low-latency, high-throughput and resilient data-exchange mechanisms. The history of internet started with web traffic being based on HyperText Transfer Protocol (HTTP)/0.9, HTTP/1.0, each successive version, including HTTP/1.1 and HTTP/2, has managed to increase efficacy in the form of persistent connections and multiplexing [7]. These enhancements have however been limited by their dependence on the Transmission Control Protocol (TCP), a transport layer protocol specifically designed for exchanging reliable information. TCP Head-of-Line (HOL) blocking has been one of the most persistent challenges that occurs in this scenario, as loss of packets in one stream blocks packets in other streams until TCP has successfully retransmitted the lost packet.

To overcome such inherent limitations, the research community and Internet Engineering Task Force (IETF) have introduced Quick UDP Internet Connections (QUIC) [1], a modern transport layer protocol that is built on top of User Datagram Protocol (UDP). QUIC, and by extension HTTP/3 [2], its application-layer protocol, introduces several features such as stream-level multiplexing, reduced handshake latency via 0-Round Trip Time (RTT) connections, connection migration, and integrated encryption leveraging Transport Layer Security (TLS) 1.3 protocol. These features significantly increase performance particularly in environments which require critical real-time responsiveness such as video conferencing, online gaming, financial trading platforms, and interactive media.

Another protocol building on QUIC like HTTP/3 is WebTransport [8]. WebTransport is a web Application Programming Interface (API) designed to support unidirectional and bidirectional, multiplexed communication between web clients and servers, allowing developers to build applications that require granular, low-latency communication without

resorting to workarounds like WebSockets, Polling, multipath TCP (MTCP), etc. By exposing the different capabilities of HTTP/3 and QUIC through a secure and efficient JavaScript interface, WebTransport stands as a fundamental building block towards a next generation protocol suitable for real-time web applications.

However, even with these developments in the protocol, WebTransport adoption in the real world of the modern web and commonly deployed cloud-native, especially in Kubernetes [3], remains a significant challenge. Kubernetes has emerged as the dominant container orchestration platform because of its robust support for scalability, self-healing, and infrastructure management using declarative syntax. Yet, within Kubernetes the networking abstractions and ingress controllers which allow the traffic into the environment are primarily designed for HTTP over TCP and UDP, offering limited to no support for the semantics of HTTP/3 or QUIC beyond generic UDP passthrough.

Current networking solutions for kubernetes such as ingress controllers (e.g., NGINX [9], Traefik) and load balancers lack visibility into the multiplexed nature of QUIC streams. As a result, all streams within a QUIC connection are routed uniformly to a single backend service, regardless of their function or content type. This places limits on the potential benefits of WebTransport in microservice architectures, where different stream types—such as audio, video and chat could be directly proxied and sent to microservices that are optimized for their respective workloads.

1.1 Motivation

The shift to real-time web communication is currently the next big thing. The rise of streaming platforms such as Twitch, YouTube, and Kick demonstrates the growing demand for delivering real-time streams of data to millions of users across the world [10]. With this surge in real-time big data, it is interesting to explore the potential advancements that could be achieved through lower-latency and more efficient performance outcomes enabled by the adoption of the WebTransport protocol.

During my time as a Development Operations (DevOps) Engineer, I personally observed the heavy dependency on the TCP networking protocol when working with Kubernetes-based solutions. My initial motivation stemmed from asking myself a few key questions about this dependency.

The core motivation discussed in this dissertation is the absence of native, stream-conscious routing support for WebTransport traffic in the Kubernetes environment. Although QUIC and HTTP/3 address many protocol-level performance issues associated with TCP, they do not yet scale effectively for microservice ecosystems. This is due to the current inability of Kubernetes' networking and ingress infrastructure to support

granular stream-level routing and application-aware processing.

Specifically, the limitations of existing Kubernetes networking and ingress infrastructure are outlined in the following ways, which serve as the primary motivation for this dissertation

- **Lack of Stream-Level Visibility:** QUIC streams within a single connection are opaque to existing Layer 4 load balancers. Without visibility into individual streams, ingress controllers cannot differentiate between types of data or implement tailored routing logic.
- **Monolithic Connection Handling:** The entire QUIC connection is treated as a single UDP session and directed to one backend, regardless of the number or nature of streams it carries. This eliminates the possibility of decomposing responsibilities across multiple microservices, leading to tightly coupled designs and inefficient resource utilization.
- **Absence of Application-Layer Intelligence:** Ingress controllers lack insight into the semantics of WebTransport streams, making it impossible to apply policies such as stream-based Quality of Service (QoS), content-based routing, or differentiated handling of critical and non-critical data flows.
- **Limited Protocol Translation Capabilities:** Many backend services still rely on HTTP/1.1 for communication. The absence of mechanisms to terminate HTTP/3 and translate individual streams into HTTP/1.1 requests further hinders interoperability and adoption in legacy-compatible environments.

This set of challenges are a major motivation to attempt productionizing WebTransport, especially when working with streams and microservices within Kubernetes-powered architecture, where one of the key architectural advantage is modularity, observability, and dynamic routing.

1.2 Research Objectives

The overarching aim of this dissertation is to bridge the aforementioned gap by introducing a WebTransport-aware routing system that seamlessly integrates into Kubernetes environments. The system enables application-layer routing decisions based on individual stream semantics, transforming the way real-time data is managed in cloud-native platforms.

1.2.1 Primary Objective

This dissertation aims to fill the gap described above with the development of a WebTransport stream-aware routing system that will integrate easily within the Kubernetes environment. Application-layer routing decisions are facilitated using individual stream semantics that strives to change the management of real-time information within cloud-native platforms.

1.2.2 Secondary Objectives

1. **Protocol Translation:** Support translation of HTTP/3/WebTransport to HTTP/1.1 and achieve the compatibility of the technology with current microservice backends that have still not implemented newer protocol types.

2. **Stream Demultiplexing:** Establish effective systems to understand QUIC streams on the basis of a fixed packet structure, thus logically isolating the various data data like audio, video and control messages.

3. **Dynamic Configuration:** Develop a flexible, configuration-driven routing mechanism that supports hot-reloading and dynamic updates via Kubernetes-native objects such as ConfigMaps and Secrets.

4. **Real-time Processing Use Case:** Demonstrate to the practical feasibility by building a scenario with live streaming of different data such as audio, video, and chat, to process and integrate with streaming backend systems like Apache Pulsar.

5. **Performance Evaluation:** To evaluate the system wiht metrics such as backend performance buffer utilization, cpu and memory.

1.3 Methodology and Approach

This research follows a task decomposition approach by breaking down the large problem into manageable components and addressing each one iteratively and progressing towards a complete solution. The approach starts with dividing overall challenge of WebTransport stream routing into distinct phases that brings us closer to the goal. At a highlevel the divided tasks include configuring Kubernetes to accept incoming UDP traffic using tools, extracting individual WebTransport streams based on custom application-layer defined headers. Deploying each component in a configuration-driven mechanisms to classify and map stream types to corresponding backend services deployed within the Kubernetes

cluster. Demultiplexing streams and forwarding to its respective microservice. Finally establishing proper monitoring mechanisms for evaluation.

1.4 Contributions

This dissertation makes notable contributions to the field of cloud-native networking by building a WebTransport-aware proxy system for Kubernetes. The system built aims to route individual WebTransport streams using a custom QUIC demultiplexer which contributes to open-source development by providing a Proof of Concept, building upon the aioquic library [4]. The solution supports dynamic routing through configuration files like ConfigMaps and Secrets, which can be updated without restarting the service providing easy deployable solution in kubernetes.

1.5 Structure & Contents

The structure of this dissertation will contain seven chapters, progressing from theoretical considerations to execution, review and conclusions. The chapters are independent of each other and are organized based on logical flow.

Chapter 1: Introduction outlines the motivation, research goals, and scope of WebTransport routing in Kubernetes.

Chapter 2: State of the Art reviews relevant literature on QUIC, HTTP/3, Kubernetes networking, and identifies existing gaps.

Chapter 3: Problem Formulation defines the core challenges, research questions, and introduces the proposed WebTransport proxy.

Chapter 4: Design presents the high-level architecture, stream structure, routing logic, and Kubernetes integration patterns.

Chapter 5: Implementation details the development setup, proxy and microservice code, deployment with ConfigMaps and Secrets, and Apache Pulsar integration.

Chapter 6: Evaluation benchmarks system performance in terms of latency, throughput, and scalability under real-world scenarios.

Chapter 7: Conclusions & Future Work summarizes findings, reflects on limitations, and suggests future improvements such as HTTP/3 pass-through and distributed scaling.

Chapter 2

State of the Art

This state-of-the-art section investigates the emerging status of real-time data communication in modern data systems. It examines three interrelated domains viz- WebTransport streams, Kubernetes and its support for emerging web protocols, and modern streaming platforms. WebTransport [8] built on top of the QUIC transport protocol [1] and HTTP/3 [2] introduces new features which enables low-latency, bidirectional communication over the internet. Its ability to support both reliable and unreliable data transfers makes it suitable for latency-sensitive applications such as live video streaming. By avoiding head-of-line blocking and supporting QUIC enabled features such as connection migration, WebTransport addresses several limitations existed in other technologies like HTTP/2, positioning itself as a next-generation protocol for web applications.

In conjunction, we will look at Kubernetes [3], which is one of the most widely used systems for container orchestration, which can handle large applications deployed across different machines. Additional support for newer generation protocols like QUIC, HTTP/3, and WebTransport in Kubernetes brings its own set of challenges while we explore the state-of-the-art solutions. This also covers several tools and components and their explanations, providing better context. Several Ingress solutions, the Gateway API, and QUIC library implementations are explored. Finally, we investigate the web streaming use case and technologies that work together to support real-time systems to process big data.

2.1 Networking Concepts

2.1.1 OSI Model

The Open Systems Interconnection (OSI) Model [7] essentially breaks down communication networks into 7 layers, viz., Physical, Data Link, Network, Transport, Session, Presentation, and Application Layer. Each of these layers handles specific tasks and ab-

stracts its functionality from the other layers. The physical layer is responsible for moving raw bits wired using cables or wireless using links. The data link layer deals with the management of data and ensures it is error-free as it travels to other devices. The network layer is responsible for handling routing decisions and deals with the transmission and reception of messages. The session, presentation, and application layers deal with managing sessions, formatting data, and running and creating application-formatted data. It's where the application services run. With respect to our dissertation topic, QUIC is a transport layer protocol, whereas HTTP/3 and WebTransport are application layer protocols.

2.1.2 IP Frames

Internet Protocol (IP) Frames, also called Packets, are a protocol structure that operates at the networking layer, which encapsulates the data. In its structure there are source and destination IP addresses, which are responsible for inter-network routing. An IP packet consists of a header that contains the IP address and protocol type, and a payload which contains the data from the underlying protocol that was used. IP packets are most critical for routing TCP, UDP, or QUIC transport data across the internet.

2.1.3 Network Interface

Network interfaces operate between the Physical and Data Link layers,. It acts as an interface between the hardware and software for network communication. Examples are Ethernet Network Interface Cards (NICs) (e.g., eth0) and wireless adapters (e.g., wlan0), which is responsible for converting raw bits to signals and for processing Ethernet frames. Any form of communication passes through a network interface. Heres a snapshot of some network interfaces on my device. Note that we have an Maximum Transmission Unit (MTU) parameter equal to 1500.

2.1.4 MTU (Maximum Transmission Unit)

The MTU decides the maximum packet size that can be transmitted over a network without being subject to fragmentation (splitting of packets). In the images above it is seen that all the network interfaces have an MTU of 1500 Bytes. In order to work with larger sized data, modiications to the protocols MTU must be done to accomadate the changes

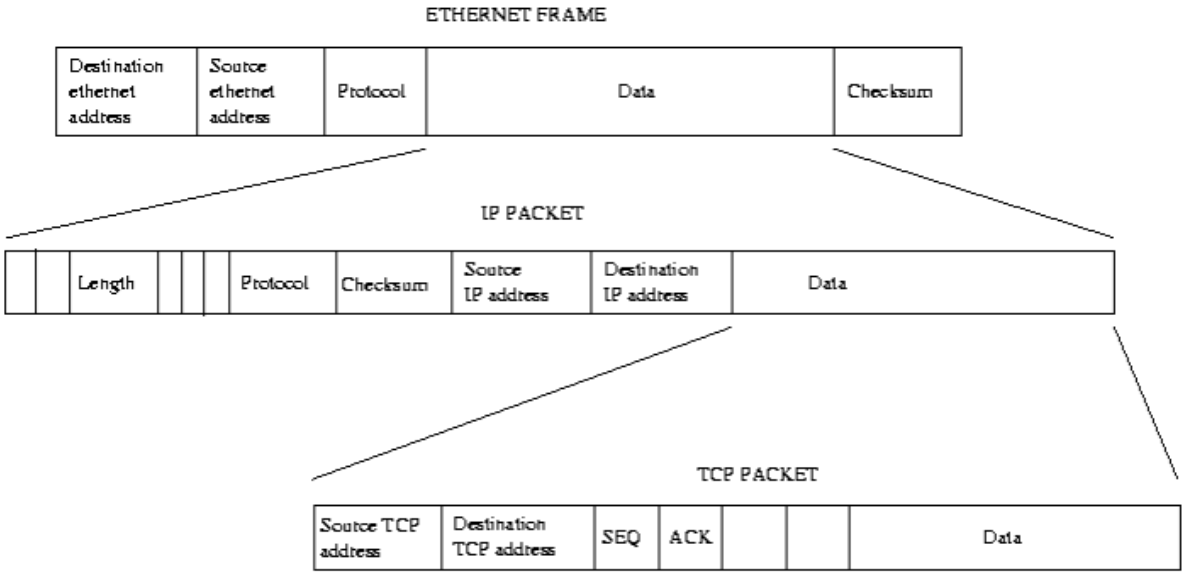
Figure 2.1: Network Interfaces

```
sibin@sibin-ThinkPad-T495:~$ ifconfig
br-5d3c7d61609f: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.49.1  netmask 255.255.255.0  broadcast 192.168.49.255
        inet6 fe80::42:53ff:fed9:de1d  prefixlen 64  scopeid 0x20<link>
        ether 02:42:53:d9:de:1d  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:6d:a5:82:af  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp2s0f0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 00:2b:67:00:4b:f1  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

2.1.5 Packets

Packets are bundled units of data that is transmitted across network interfaces. It is idenitified as: bits for Physical Layer, frames for Data Link, packets for Network, and segments or datagrams for Transport Layer. For HTTP/3 App data, Ethernet frames encapsulates IP packets which contain UDP datagrams which carry QUIC Pakcets with multiple streams enabling multiplexing.

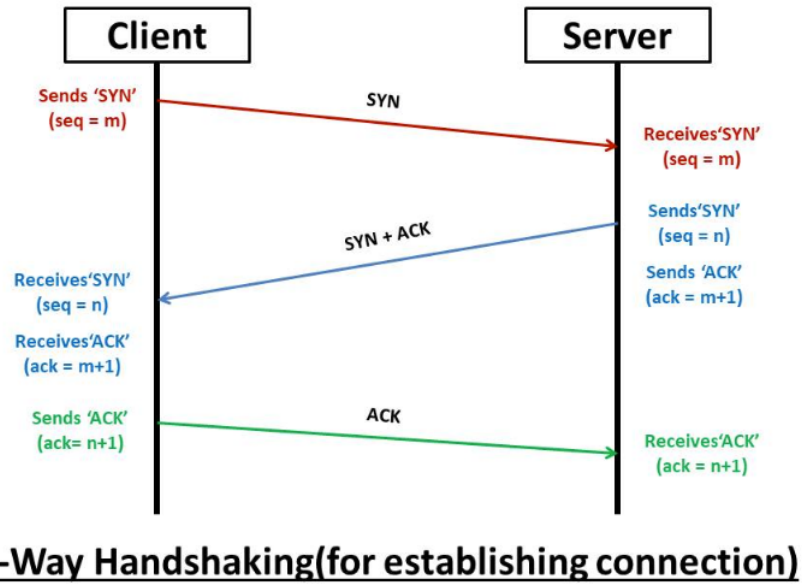
Figure 2.2: Flow of Packets



2.1.6 TCP (Transmission Control Protocol)

TCP is a connection oriented protcol which ensures data reliability with several mechanisms such as Acknowledgments (Acks), Retransmissions and in-order delivery [7]. TCP fucntions by establishing a three-way handshake between client and server. The handshakes can be seen in the figure below. TCP requires roundtrups to initiate communication and when it works with higher level application protocols which require TLS, it impacts performance introducing latency particularly during the connection setup.

Figure 2.3: TCP Handshake



TCP's design introduced challenges such as Head of Line Blocking. This is an issue which impacts performance for applications at scale. This HoL problem basically means that when a packet is lost, all subsequent packets are waited until the packet reaches the destination. Several approaches are followed to mitigate this issue such one such approach is multi-path TCP where data is sent over multiple paths avoiding this issue but the underlying protocol still suffers from this issue and the middlebox configuration is a concern.

TCP guarantees reliability making it ideal for application which require stringent ordered delivery. However, due to middle box configurations and TCPs reliance on outdated implementations restricts it from being highly performative.

2.1.6.1 Operating System (OS) Configurations for TCP

In linux machines, TCP's state is exposed through the /proc/next/tcp path which is a virtual filesystem part of the procfs interface. This provides dynamically generated real time information about the active TCP connections which includes local addresses, remote addresses, port numbers, connection states and sequence number. Each entry persists for 60s in the TIME_WAIT state. (Protocol Compliance Requirement). The confgurations for handling concurrent tcp connections are monitored via parameters such as tcp_max_orphans, fs.file-max, and ulimit -n. Modification to these parameter increases the set limits to increase performance.

2.1.6.2 TCP Congestion and Flow Control

TCP's congestion control is managed by the sender with the congestion window ('cwnd'), its initialized with a small value at the start (e.g., 10 segments) and adjusted based on network feedback from the server. During the slow start phase, 'cwnd' grows exponentially with reception of acknowledgments (ACKs), when a packet loss is detected via duplicate ACKs or timeouts, it triggers a reduction in 'cwnd' to mitigate network congestion with the help of flow control which is managed by the receiver's advertised window ('rwnd'), it prevents buffer overflow by limiting the sender's transmission rate.

2.1.7 Middle-Box Ossification

Middleboxes become bottlenecks because of its old implementation and inability to update widely adopted middleboxes. In context of tcp it means the network devices like firewalls, routers have tcp implementation at their kernel level, which becomes an issue when updates to the core implementation of protocol are required.

2.1.8 UDP

User Datagram Protocol (UDP), is a connectionless protocol that focuses low latency than reliability [7]. This is due to the fact that it lacks the built-in mechanisms for retransmission, ordering, or congestion control and hence makes it suitable for applications where speed is critical, and occasional packet loss is tolerated. Due to this it enables one-RTT communication which is ideal for Domain Name System (DNS) queries and real-time media streaming like online gaming, video streaming, etc. But, its lack of reliability, security, and congestion control makes it unsuitable for use in cases requiring guaranteed delivery.

2.1.8.1 UDP Packet Structure

UDP operates within a simple packet structure encapsulated as follows:

- **Ethernet Frame**: It contains the Media Access Control (MAC) addresses(source and destination) and the control information for physical transmission.
- **IP Packet**: It encapsulates the source and destination IP addresses for routing.
- **UDP Datagram**: It includes the source and destination ports and the data(payload), with no reliability or ordering mechanisms.

UDP's minimalist design enables low-latency communication but it lacks the robust features of TCP, and hence it requires application-level handling of reliability and ordering where needed.

UDP, lacking connection state, does not maintain persistent entries in /proc(the virtual filesystem used to expose kernel and process information). Both protocols face challenges with middleboxes(see section 2.1.7), such as firewalls, which often rely on outdated TCP configurations, complicating updates for advanced features like TCP Fast Open.

2.1.9 Limitations of Traditional Transport Layer Protocols

The foundational transport protocols of the internet, TCP and UDP, face well-documented limitations that hinder performance, scalability, and adaptability in modern network environments.

TCP suffers mostly from *head-of-line (HOL) blocking*, a condition where the loss of a single packet holds up the delivery of all subsequent packets in that stream, thereby leading to increase in latency and degrading user experience. In addition to internal limitations, TCP evolution is impeded by *middlebox ossification* see section 2.1.7. Network devices like legacy firewalls and Network Address Translation (NAT) gateways often do not recognize newer TCP features, such as, TCP Fast Open or Multipath TCP, blocking their deployment and stalling protocol innovation across the wider internet.

On the other hand, UDP, designed as a minimal transport layer, circumvents many of TCP's internal restrictions but introduces its own set of challenges. It lacks built-in mechanisms for reliability, packet ordering, and congestion control, forcing developers to reimplement these features at the application level. Moreover, UDP is susceptible to *NAT rebinding*, where NAT devices revoke and reassign port mappings after idle periods, leading to unexpected connection drops.

Both TCP and UDP also has host-level limitations. Important among these is the finite ephemeral port range, typically around 65,000 ports per IP address which restricts the number of simultaneous connections to the same destination. In high-throughput systems, the availability of file descriptors and system memory further limits the scale of concurrent network connections, imposing additional resource constraints.

These protocol-level and systemic challenges have led to the development of next-generation transport protocols like QUIC, that aim to address these shortcomings in a modern, application-aware manner.

2.1.10 QUIC Protocol

The Quick UDP Internet Connections (QUIC) protocol represents an innovative transport layer solution designed to address the limitations of traditional TCP-based protocols, particularly in the context of modern web applications. Unlike TCP, which relies on sequential TLS and TCP handshakes, QUIC leverages UDP to combine transport and security handshakes into a single step, significantly reducing connection setup latency.

QUIC's design aims to provide TCP-like reliability, stream multiplexing, and congestion control (see section 2.1.10.2) while mitigating TCP's drawbacks, such as HOL blocking and the need for kernel-level updates across network infrastructure. By operating over UDP, QUIC enables deployment without requiring modifications to middleboxes like firewalls or routers, which often struggle with outdated TCP configurations.

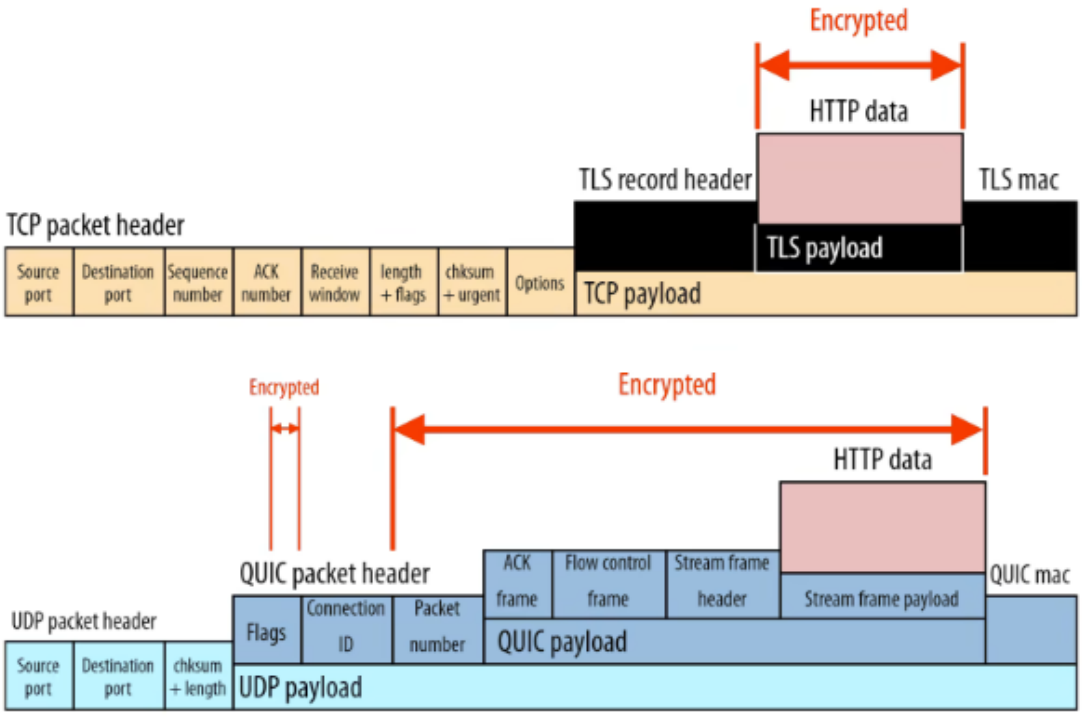
QUIC is a transport protocol that integrates the TLS 1.3 handshake with its initial transport handshake, achieving connection setup in a single RTT. QUIC also incorporates reliability, in-order delivery, and congestion control, addressing UDP's limitations while retaining its low-latency characteristics. The protocol supports multiplexing through independent streams within a single connection, eliminating inter-stream HOL blocking, a significant issue in TCP-based HTTP/2.

2.1.10.1 QUIC Packet Structure

QUIC operates within a layered architecture, encapsulated as follows:

- **Ethernet Frame**: Contains MAC addresses and control information for physical transmission.
- **IP Packet**: Encapsulates source and destination IP addresses for network routing.
- **UDP Datagram**: Provides a lightweight, connectionless transport layer for QUIC packets.
- **QUIC Packet**: Includes a header (e.g., Short Header with Destination Connection ID and Packet Number) and an encrypted payload containing frames such as ACK, STREAM, and datagram frames.

Figure 2.4: QUIC vs TCP packet Structure



For example, in above figure [11] depicting QUIC Packet structure we can see that quic packets are sent as a udp packet in its payload

2.1.10.2 Congestion and Flow Control

QUIC implements congestion control at the connection level, using a sender-managed congestion window (`cwnd`) that adjusts based on network conditions, similar to TCP. The sender initializes `cwnd` conservatively (e.g., 10 segments) and increases it exponentially during the slow-start phase or linearly during congestion avoidance, reducing it upon detecting packet loss. Flow control, managed by the receiver's advertised window (`rwnd`), operates at the stream level, ensuring that senders do not overwhelm receiver buffers.

Stream prioritization, supported by QUIC's APIs, allows developers to assign higher priority to critical streams (e.g., audio over video), optimizing resource allocation. Frame coalescing further enhances efficiency by packing multiple frame types into a single UDP datagram, following the Maximum Transmission Unit (MTU).

2.1.10.3 Implementation Considerations

QUIC's user-space implementation, unlike TCP's kernel-space design, enables rapid iteration but introduces performance overhead. Debugging tools like `qlog` and `qvis` facilitate

debugging and analysis of QUIC's advanced features, such as multiplexing and congestion control. For large file downloads, QUIC supports range-based requests which is similar to HTTP Range headers, allowing recovery of broken connections by requesting only missing byte ranges, provided servers support partial content requests.

QUIC's ability to multiplex streams and datagrams within a single connection enhances its suitability for applications requiring concurrent data transfers, such as web browsing and file downloads. However, intra-stream HOL blocking persists, as data within a single stream is delivered in order, requiring retransmission of lost packets before subsequent data can be processed. In this dissertation implementation, we will be focusing on the feature multiplexing (multiple independent streams per connection).

2.1.11 Comparison of TCP, UDP, and QUIC

The following table summarizes the key features of QUIC compared to TCP and UDP, highlighting its hybrid design:

Feature	**UDP**	**TCP**	**QUIC**
Connectionless	Yes	No	Yes (UDP-based)
Reliable Delivery	No	Yes	Yes
In-Order Delivery	No	Yes	Yes (per stream)
Multiplexing	No	No	Yes (multiple streams per connection)
Built-in TLS	No	No (external TLS)	Yes (TLS 1.3 mandatory)
Head-of-Line Blocking	No	Yes (affected)	No (avoids inter-stream blocking)
Connection Migration	No	No	Yes (survives IP changes)

Table 2.1: Comparison of features across UDP, TCP, and QUIC

2.2 HTTP Protocols

The HyperText Transfer Protocol (HTTP) has undergone multiple changes since its inception, which helped it evolve in order to meet the growing demands of modern web applications. Each version introduces significant changes in how data is structured, delivered, and optimized over networks.

2.2.1 HTTP/0.9

Introduced in 1991, HTTP/0.9 was the earliest version of the protocol, designed for the simple retrieval of HTML documents. It supported only the `GET` method, and responses

consisted of raw HTML without headers. Lacking support for metadata, status codes, or content types, HTTP/0.9 was limited to extremely basic use cases. It operated over a single TCP connection and closed the connection immediately after the response, making it inefficient for handling multiple resources.

2.2.2 HTTP/1.1

Standardized in 1997 (RFC 2068, later updated by RFC 2616 and RFC 7230 series), HTTP/1.1 allowed multiple requests and responses to be sent over a single TCP connection hence introducing persistent connections [7]. It added methods like `PUT`, `DELETE`, and `OPTIONS` and support for chunked transfers, caching mechanisms, virtual hosting. However, HTTP/1.1 had head-of-line blocking at the application layer which severely limit performance, particularly for resource-heavy web pages due to fundamentally being constrained by the serialized nature of TCP streams.

2.2.3 HTTP/2

HTTP/2 was published in 2015 (RFC 7540), in order to fix the performance limitations of HTTP/1.1 [7]. It introduced binary framing, header compression (HPACK), and multiplexing, allowing multiple concurrent streams over a single TCP connection. These changes led to significant reduction in latency and improved page load times. As HTTP/2 still has underlying TCP, it remains susceptible to TCP-level head-of-line blocking. If a packet is lost, the entire TCP connection stalls until retransmission is complete, affecting all multiplexed streams.

2.2.4 HTTP/3

HTTP/3 is an application-layer protocol built on top of QUIC, developed to overcome the limitations of HTTP/2, particularly head-of-line (HOL) blocking caused by its reliance on TCP [2] [11]. By replacing TCP with QUIC (which runs over UDP), HTTP/3 introduces features such as built-in encryption (TLS 1.3), reduced handshake latency, connection migration, and robust stream multiplexing. These capabilities make HTTP/3 ideal for modern web environments, including mobile-first, real-time, and high-throughput applications.

2.2.4.1 Multiplexing, Flow Control, and Congestion Handling

HTTP/3 benefits directly from QUIC's architecture, which provides independent, multiplexed streams within a single connection. This means that packet loss on one stream

doesn't block others, resolving the HOL blocking issue thats present in TCP. It allows more efficient and smoother data delivery, especially over unreliable or variable networks.

QUIC also implements stream-level flow control, ensuring that each stream can progress without interfering with others. Additionally, it features advanced congestion control algorithms (e.g., BBR or CUBIC), similar to TCP, but designed to react faster and recover more effectively in poor network conditions. These mechanisms enable HTTP/3 to maintain low latency and high performance across diverse network environments.

HTTP/3, through QUIC, provides several key enhancements including faster connection establishment via reduced round trips, encryption by default using TLS 1.3, stream multiplexing without HOL blocking, connection migration in case of IP changes (e.g., switching from Wi-Fi to mobile data), and 0-RTT support allowing faster repeat connections.

2.2.4.2 Implementation and Adoption

HTTP/3 is supported by major implementations such as Google's quiche, Cloudflare's quiche, Mozilla's neqo, and Facebook's mvfst, with integration into popular web servers like nginx, LiteSpeed, and Apache. Browser support is strong, with Chrome, Firefox, Safari, and Edge already implementing HTTP/3.

However, widespread adoption is still in progress. Challenges such as middlebox compatibility, UDP tuning limitations, and infrastructure readiness hinders deployment. Nonetheless, as HTTP/3 continues to demonstrate tangible performance and reliability improvements, especially in mobile and high-latency scenarios, its adoption is steadily accelerating.

2.2.5 Summary Table: HTTP Protocol Evolution

Feature	HTTP/0.9	HTTP/1.1	HTTP/2	HTTP/3
Transport Layer	TCP	TCP	TCP	QUIC (UDP-based)
Multiplexing	No	No (pipelining only)	Yes (over TCP)	Yes (parallel over QUIC)
Header Compression	No	No	Yes (HPACK)	Yes (QPACK)
Persistent Connections	No	Yes	Yes	Yes
HOL Blocking (Transport)	N/A	Yes	Yes	No
Encryption	No	Optional (via TLS)	Optional (via TLS)	Yes (via QUIC)

Table 2.2: Evolution of HTTP protocol features across versions

2.3 WebTransport Protocol

WebTransport is a modern protocol built over QUIC, designed to provide developers with flexible transport options for web applications. Unlike WebSockets, which rely on TCP and lack multiplexing, WebTransport leverages QUIC's stream and datagram capabilities to support both reliable and unreliable data transfers within a single connection. WebTransport's design enables developers to choose between streams for ordered, reliable delivery and datagrams for low-latency, loss-tolerant communication, making it suitable for diverse applications, including real-time gaming, audio streaming, and large file transfers.

WebTransport's architecture encapsulates data within QUIC packets, which are transmitted over UDP datagrams, IP packets, and Ethernet frames. This layered approach ensures compatibility with existing network infrastructure while providing advanced features like connection migration and frame coalescing. The framework's APIs, including Streams and Datagram APIs, allow fine-grained control over data transmission, as will be explored in the implementation section with a focus on WebTransport Streams.

2.3.1 WebTransport Streams

WebTransport Streams provide a reliable, ordered, and flow-controlled mechanism for data transfer, analogous to TCP but with QUIC's performance benefits. Each stream operates independently within a QUIC connection, identified by a unique Stream ID (Quic streams in background), allowing concurrent data transfers without inter-stream HOL blocking. For example, a single QUIC packet can carry multiple STREAM frames, such as HTML data and image data, as demonstrated in the provided content.

Streams are particularly suited for applications requiring guaranteed delivery, such as file downloads or structured data exchanges. However, intra-stream HOL blocking persists, as data within a single stream must be delivered in order, requiring retransmission of lost packets. Stream prioritization APIs enable developers to assign higher priority to critical streams (e.g., audio over video), optimizing resource allocation. In the implementation section, WebTransport Streams will be selected as the protocol of choice due to their reliability and suitability for structured web applications.

2.3.2 WebTransport API

The WebTransport API is a modern JavaScript interface that enables web applications to communicate over QUIC, offering both reliable and unreliable transport options [12]. It is designed to support low-latency, multiplexed, and secure communication directly from

the browser, making it ideal for use cases such as real-time gaming, video streaming, and collaborative applications.

WebTransport exposes two main interfaces in javascript implementation.

2.3.3 JavaScript Implementation

2.3.3.1 Streams API – Reliable and Ordered Communication

The Streams API provides bidirectional and unidirectional streams that ensure reliable, ordered data transfer. This makes it suitable for applications requiring robust delivery, such as sending structured HTML or JSON between a client and server.

Bidirectional Stream Example A client can open a stream and write data reliably to the server:

```
const transport = new WebTransport('https://example.com:4999');
await transport.ready;

const stream = await transport.createBidirectionalStream();
const writer = stream.writable.getWriter();

await writer.write(new TextEncoder().encode('<html><head>'));
```

Unidirectional Stream Example Used for sending data in one direction only:

```
const stream = await transport.createUnidirectionalStream();
```

These stream methods are part of the WebTransport session object. createBidirectionalStream() and createUnidirectionalStream() return stream instances, which can be accessed using .writable or .readable interfaces for sending and receiving data. Encoders like TextEncoder() convert strings into binary format suitable for transport.

2.3.3.2 Datagram API – Unreliable and Unordered Communication

The Datagram API allows for lightweight, unreliable, and unordered data exchange—perfect for scenarios where speed is prioritized over reliability, such as real-time positional updates in games or telemetry data.

Datagram Example:

```
const transport = new WebTransport('https://example.com:4999');
await transport.ready;

await transport.sendDatagram(new TextEncoder().encode('x:10,y:20'));
```

The sendDatagram() function transmits a raw byte payload over QUIC datagrams. Since datagrams are not guaranteed to arrive or preserve order, developers must handle any necessary retries or loss-tolerance logic in the application layer.

2.3.4 Custom Header Handling

Unlike HTTP-based protocols, WebTransport streams do not inherently support headers for identifying content type or routing information. To overcome this, developers must embed custom headers within the data payload itself.

For example, when sending audio or video data over a stream, a header might be prefixed to indicate the type:

```
const header = 'audio';
const payload = new Uint8Array([...new TextEncoder().encode(header), ...binaryAudio

await writer.write(payload);
```

On the receiving end (e.g., in a server or reverse proxy), the application parses this prefixed header to determine how to route or process the incoming stream—such as directing it to the appropriate microservice (e.g., audio handler vs. video handler). While this provides flexibility, it introduces additional complexity in parsing logic and requires robust message framing techniques. Since we are working with WebTransport Protocol we will be exploring this further in the implementation.

WebTransport is a new method to send data over the internet using modern web technologies. Its main version works over HTTP/3 and is explained in RFC 9297, but it is not yet an official standard. The WebTransport API used in web browsers is also still a Working Draft by the W3C, meaning it may change before becoming a standard. This opens an area if research for this protocol.

2.4 Wireshark (Packet Capturing Tool)

Wireshark is a widely used open-source network protocol analyzer that allows real-time capture and inspection of network traffic across various protocols and interfaces. By selecting a network interface (e.g., Ethernet, Wi-Fi, or loopback), users can monitor all incoming and outgoing packets exchanged between a client and server. This level of visibility makes it an essential tool for network diagnostics, debugging, and protocol analysis, especially in complex distributed environments. Wireshark provides a graphical

interface that breaks down packet details across different OSI layers, offering timestamps, protocol headers, payloads, and flow sequences.

One of Wireshark's key features is its ability to decrypt encrypted traffic, including packets protected by TLS, provided the necessary session keys (e.g., pre-master secrets) are available. This is particularly useful when analyzing secure protocols like HTTPS, HTTP/3, or QUIC, which rely heavily on TLS 1.3 for encryption. By decrypting the payload, Wireshark enables developers and analysts to troubleshoot issues related to application-layer behavior, latency, or data integrity, even within secure connections (See section 5.4.3 in implementation). This powerful functionality makes Wireshark a critical tool for both performance tuning and ensuring secure, reliable communication across networks. We will leverage this tool for our implementation in order to understand what exactly is going on.

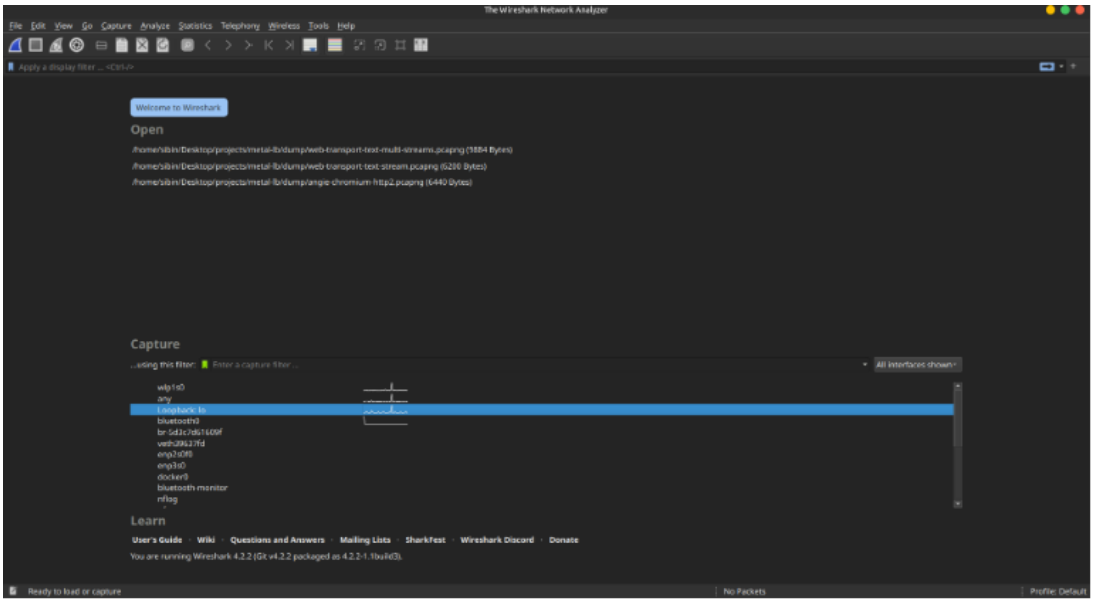


Figure 2.5: WebSocket Interfaces

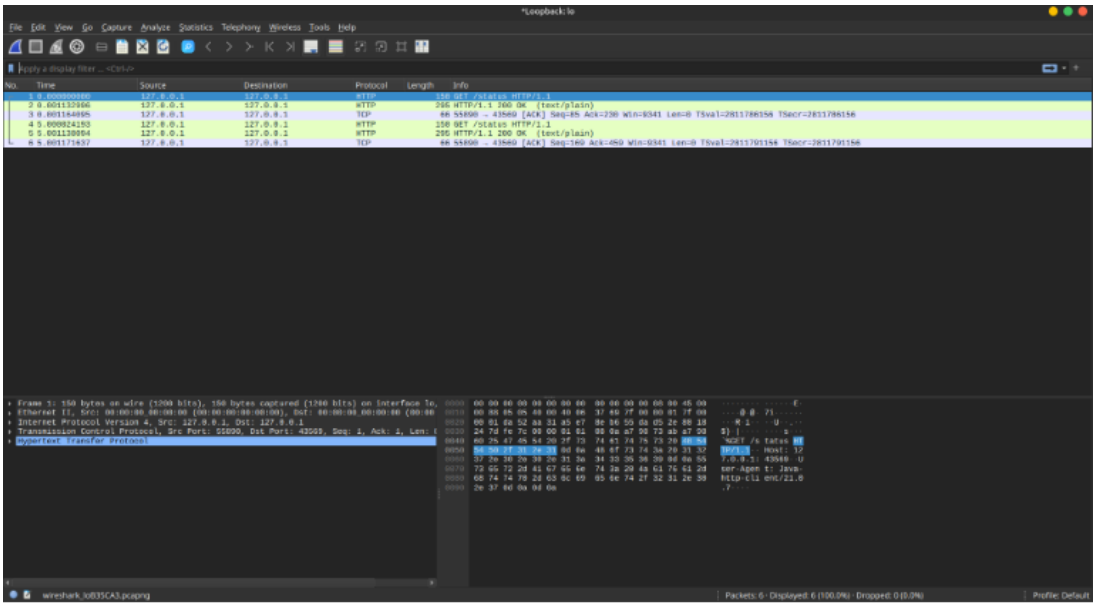


Figure 2.6: WebSocket Packets

2.5 Containers

Containers are lightweight, portable, and consistent runtime environments that package an application along with all its dependencies, configuration files, and system libraries. Unlike traditional virtual machines, containers share the host operating system kernel, making them faster to start and more resource-efficient. Containers are created with a container image and all the information is bundled into an image and stored in a registry. This makes it ideal for deploying and scaling applications in a microservices architecture.

In context for this dissertation, in order to deploy and run applications in Kubernetes, the prerequisite is having a Docker image ready.

2.6 Kubernetes Concepts

2.6.1 Kubernetes Architecture Overview

Kubernetes is built on a distributed architecture consisting of a Control Plane and multiple Worker Nodes. The Control Plane manages the cluster's desired state and handles decisions such as scheduling, scaling, and responding to cluster events. It includes components like the API Server, etcd (a consistent, distributed key-value store), Scheduler, and Controller Manager. The Control Plane acts as the brain of the system, making high-level decisions and continuously monitoring and adjusting the cluster to match the user-defined configuration.

The Worker Nodes run the actual containerized applications. Each node hosts essential services: the kubelet (which ensures containers are running properly as specified by the control plane), the container runtime (e.g., containerd), and the kube-proxy, which handles networking and service discovery. Kubernetes follows a declarative model, where users define the desired state (like the number of replicas, image versions, etc.), and the system uses control loops to match the actual state with the desired state, ensuring continuous compliance and fault tolerance. Various core components within these layers work together to ensure reliability, scalability, and self-healing of applications across the cluster.

2.6.2 Kubernetes Components

2.6.2.1 API Server

The Kubernetes API Server (kube-apiserver) serves as the central management interface, exposing the Kubernetes API for communication between cluster components and external clients. It processes RESTful requests, validates and stores object configurations (e.g., Pods, Services) in the etcd database, and coordinates cluster operation. Any operation in the cluster goes through the API Server.

2.6.2.2 etcd

etcd is a distributed key-value store that maintains the cluster's configuration and state data, ensuring consistency and fault tolerance. It stores all Kubernetes objects, such as Pod definitions and Service configurations, providing a single source of truth for the control plane. The etcd allows to rollback to a previous cluster state.

2.6.2.3 Controller Manager

The Controller Manager (kube-controller-manager) runs control loops that monitor the cluster's state and match it with the desired state defined in manifests. It includes controllers like the Node Controller (managing node lifecycle), ReplicaSet Controller (ensuring pod replicas), and Service Controller (integrating with load balancers).

2.6.2.4 Scheduler

The Scheduler (kube-scheduler) assigns pods to nodes based on resource requirements, constraints, and policies (e.g., affinity, taints). It optimizes resource utilization by evaluating node capacity and workload demands. For streaming use cases, the Scheduler ensures that Pulsar brokers or services are placed on nodes with sufficient resources, enhancing performance.

2.6.2.5 Kubelet

The Kubelet, running on each worker node, manages pod lifecycle by communicating with the API Server and container runtime (e.g., containerd, CRI-O). It ensures containers are running, healthy, and configured per pod specifications. In HTTP/3 deployments, Kubelets manage containers hosting QUIC-based services, as seen in prior Angie configurations.

2.6.2.6 Kube-Proxy

Kube-Proxy, also running on worker nodes, manages network rules to enable communication between pods, services, and external clients. It supports modes like iptables and IPVS, facilitating load balancing for services. In this dissertation, MetalLB setup [13], Kube-Proxy worked alongside MetalLB to expose HTTP/3 services via UDP passthrough.

2.6.2.7 Container Runtime

The container runtime (e.g., containerd, CRI-O) executes containers within pods, handling image pulling, container creation, and lifecycle management. It interfaces with the Kubelet to ensure containers adhere to pod specifications.

2.6.2.8 Storage Provisioner

The Storage Provisioner dynamically allocates persistent storage to pods via PersistentVolumeClaims (PVCs) and PersistentVolumes (PVs) (see section 2.6.2.16). It interfaces with storage backends (ie volumes) to provision volumes based on StorageClass

definitions. In this dissertation, for Apache Pulsar [5], the Storage Provisioner ensures bookies have access to persistent storage for ledgers, supporting fault-tolerant data retention. Also configurations, involving MetalLB and Pulsar, relied on Storage Provisioners to manage storage for stateful workloads.

2.6.2.9 CoreDNS

CoreDNS, the default DNS service in Kubernetes, provides service discovery and name resolution within the cluster. Running as a pod, it resolves service names (e.g., http3-service.default.svc.cluster.local) to Pod IP addresses, enabling communication between pods and services. In HTTP/3 deployments, CoreDNS resolves backend service names for QUIC-based traffic, as seen in prior Angie and MetalLB setups. Its plugin-based architecture supports customization, such as integrating with external DNS providers.

2.6.2.10 Namespaces

Namespaces are logical separation of resources into groups. For example, all of the core architecture components run in a separate namespace kube-system.

2.6.2.11 Pods

A Pod is the smallest and simplest unit in the Kubernetes object model. It represents a single instance of a running process in a cluster. A pod can contain one or more containers that share the same network namespace, IP address, and storage volumes. Containers in the same pod can communicate using localhost and share mounted storage, making them ideal for tightly coupled workloads (e.g., a main app container and a sidecar for logging). Pods are ephemeral—once deleted or crashed, they aren't recreated unless managed by a higher-level controller like a Deployment or StatefulSet.

Figure 2.7: Pods

```
sibin@sibin-ThinkPad-T495:~$ kubectl get pods -n kube-system
NAME                               READY   STATUS    RESTARTS      AGE
coredns-674b8bbfcf-n9lsw           0/1     Running   2 (41s ago)   5d7h
etcd-minikube                      1/1     Running   2 (41s ago)   5d7h
kube-apiserver-minikube            1/1     Running   2 (41s ago)   5d7h
kube-controller-manager-minikube   1/1     Running   2 (41s ago)   5d7h
kube-proxy-xc2s9                   1/1     Running   2 (41s ago)   5d7h
kube-scheduler-minikube            1/1     Running   2 (41s ago)   5d7h
storage-provisioner                1/1     Running   5 (41s ago)   5d7h
```

The above figure shows the pods present in the namespace kube-system

2.6.2.12 Deployments

A Deployment is a controller that manages the lifecycle of pods. It ensures the desired number of pod replicas are running and updates them in a controlled manner. With deployments, users can perform rolling updates, rollbacks, and maintain zero downtime during application changes. When you update a deployment (e.g., to use a new container image), Kubernetes gradually replaces the old pods with new ones, ensuring stability throughout the process. Deployments abstract the complexity of pod creation and replication, offering scalability and resilience.

Figure 2.8: Deployments

```
sibin@sibin-ThinkPad-T495:~$ kubectl get deploy
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
microservice-audio-deployment  1/1     1            1           11d
microservice-chat-deployment   1/1     1            1           11d
microservice-video-deployment  1/1     1            1           11d
pulsar-adminconsole            1/1     1            1           3d
pulsar-autorecovery            1/1     1            1           3d
pulsar-bastion                 1/1     1            1           3d
pulsar-broker                  1/1     1            1           3d
pulsar-proxy                   1/1     1            1           3d
pulsar-pulsarheartbeat         1/1     1            1           3d
webtransport-router-deployment 1/1     1            1           11d
sibin@sibin-ThinkPad-T495:~$
```

The above figure illustrates the different deployment in the cluster's default namespace

2.6.2.13 Services

Kubernetes Services provide a stable networking endpoint to access pods, abstracting away their transient nature. Since pods have dynamic IPs and may be replaced, services ensure that applications or users can access workloads reliably. There are several types of services:

- ClusterIP (default): Exposes the service on a cluster-internal IP. It is only accessible within the cluster, useful for internal communication between services.
- NodePort: Exposes the service on a static port (between 30000–32767) on each worker node's IP. Traffic to this port is forwarded to the underlying pods. It makes services accessible outside the cluster using NodeIP:NodePort.
- LoadBalancer: Provisions an external load balancer (in cloud environments like Amazon Web Services (AWS) or Google Cloud Platform (GCP)) and routes traffic

from the external world to the service. It provides a single external IP and balances traffic to the appropriate pods.

Figure 2.9: Service

```
sibin@sibin-ThinkPad-T495:~$ kubectl get svc
NAME                          TYPE           CLUSTER-IP       EXTERNAL-IP
kubernetes                    ClusterIP      10.96.0.1        <none>
microservice-audio-service    ClusterIP      10.107.53.194    <none>
microservice-chat-service     ClusterIP      10.107.146.245   <none>
microservice-video-service    ClusterIP      10.96.231.67     <none>
pulsar-adminconsole           LoadBalancer   10.106.176.180   192.168.49.1
pulsar-bookkeeper             ClusterIP      None             <none>
```

The above figure shows the services present in our cluster in default namespace.

If we are working with an internal service, go for default ClusterIP, and if you want to expose the service to the outside world, use NodePort or LoadBalancer.

2.6.2.14 ConfigMaps

ConfigMaps allows to externalize configuration from container images, separating application code from configuration data. They store key-value pairs that can be used to inject environment variables, command-line arguments, or configuration files into pods. This decoupling allows the same container image to be reused in different environments by simply changing the ConfigMap. For example, you could store database URLs, feature flags, or app settings without baking them into the container. In implementation microservice container image was deployed with config map.

2.6.2.15 Secrets

Secrets are similar to ConfigMaps but designed specifically for sensitive data like passwords, tokens, SSH keys, or certificates. Secrets are base64-encoded (not encrypted by default) and can be mounted as volumes or exposed as environment variables. They help protect confidential data by controlling access through RBAC and reducing the risk of hardcoding secrets into application images or code.

2.6.2.16 Persistent Volumes (PVs) and Persistent Volume Claims (PVCs)

Kubernetes provides an abstraction layer for storage through Persistent Volumes (PVs) and Persistent Volume Claims (PVCs). A PV is a piece of storage (e.g., a disk in a cloud provider or an NFS mount) provisioned by an admin or dynamically via StorageClasses. A PVC is a user's request for storage with specific requirements (like size, access mode).

This decouples storage provisioning from storage consumption, allowing pods to request storage without knowing the details of the underlying storage infrastructure. PVs can outlive pods and enable stateful applications like databases to persist data even when pods are recreated.

2.6.2.17 StatefulSets

StatefulSets are a Kubernetes controller used to manage stateful applications that require stable network identities and persistent storage. Unlike Deployments, which treat all pods as interchangeable, StatefulSets assign each pod a unique, stable identity (including name and network address) and ensure ordered, graceful deployment, scaling, and deletion.

StatefulSets are ideal for workloads like databases (e.g., MongoDB, PostgreSQL), distributed systems (e.g., Kafka, ZooKeeper [14]), and any application that requires persistent storage tied to a specific pod. Each pod in a StatefulSet can be connected via a predictable DNS name and retains its associated PVC even when restarted.

2.6.3 Cloud Kubernetes Clusters

As Kubernetes adoption continues to rise across industries, cloud providers have responded with fully managed solutions to simplify deployment, scaling, and maintenance. Among the most prominent managed Kubernetes offerings are Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), and Google Kubernetes Engine (GKE). This section reviews these platforms, examining their design principles, cloud integration capabilities, and recent developments, with a focus on their maturity and positioning in the cloud-native ecosystem.

2.6.3.1 EKS

Amazon Elastic Kubernetes Service (EKS), introduced by AWS in 2018, offers a fully managed control plane that spans multiple Availability Zones to ensure high availability and resilience [15]. It leverages the security and scalability of AWS infrastructure while supporting native Kubernetes tooling. EKS integrates tightly with AWS services such as Identity and Access Management (IAM) for access control, CloudWatch for observability, and Elastic Load Balancer (ELB) for load balancing. Recent additions like EKS Anywhere (for on-premises Kubernetes) and Bottlerocket (a container-optimized OS) reflect AWS's commitment to extending Kubernetes beyond cloud boundaries (AWS, 2025). However, EKS has historically been more complex to configure than its competitors, particularly due to its networking model, which tightly couples pods to Virtual Private Cloud (VPC) configurations (Smith Alqahtani, 2024).

2.6.3.2 AKS

Azure Kubernetes Service (AKS) represents Microsoft's managed Kubernetes platform and is particularly favored in enterprise environments with existing Azure investments [16]. AKS manages the Kubernetes control plane free of cost and supports features like node auto-scaling, Azure Active Directory (AD) integration, and Virtual Node support via Azure Container Instances (ACI). As of 2025, Microsoft has enhanced its hybrid cloud capabilities through Azure Arc, enabling consistent Kubernetes management across cloud and on-prem infrastructure (Microsoft Docs, 2025). Though AKS is generally user-friendly and well-integrated with Azure DevOps and Visual Studio, it sometimes lags behind Google Kubernetes Engine (GKE) in offering the latest Kubernetes versions (Nguyen et al., 2023).

2.6.3.3 GKE

GKE is widely considered the most advanced managed Kubernetes platform, owing to Google's foundational role in developing Kubernetes [17]. It offers both a Standard mode (with user-managed nodes) and Autopilot mode, where Google handles infrastructure completely. GKE is known for its rapid adoption of upstream Kubernetes versions, robust autoscaling features (including vertical and horizontal autoscaling), and seamless integration with Istio, Anthos, and Cloud Monitoring. In 2025, GKE continues to be the platform of choice for use cases involving AI/ML and latency-sensitive applications, driven by GCP's high-performance network and container runtime optimizations. However, the opinionated nature of GKE Autopilot may limit low-level customization, which could be a drawback for advanced users.

2.6.4 Comparison of Cloud Clusters

More or less, all Kubernetes services offer similar core functions because they all use the open-source Kubernetes implementation. The main differences come from how well they fit specific use cases and how easily they integrate with other tools and services you use. Another big factor when choosing a service is the cost.

Running Kubernetes on managed cloud platforms like EKS, AKS, and GKE can quickly become expensive. Costs include not only the control plane and worker nodes but also storage, networking, and extra monitoring or logging services. Even running a small cluster can cost several hundred dollars a month, which is often too high for students, researchers, or small projects with limited budgets. These expenses make it difficult to experiment freely or run many tests without worrying about costs. Because of

Aspect	**Amazon EKS**	**Azure AKS**	**Google GKE**
Learning Curve	Steep	Easy for Azure users	Very user-friendly
Flexibility	High	Moderate	Lower in Autopilot mode
Hybrid Cloud	EKS Anywhere	Azure Arc	Anthos
K8s Feature Updates	Slower	Slight delay	Fastest
Control Plane Cost (EU/hour)	$0.10	Free	$0.10 (Standard), Free (Autopilot)
Minimum Monthly Cost (1 node)	~$37 (t3.small + control plane)	~$27 (B2s, free control plane)	~$30 (e2-small, Autopilot includes control plane)

Table 2.3: Simplified Comparison of Kubernetes Cloud Services with Cost Estimates (EU Region)

this, managed cloud Kubernetes may not always be the best choice for those just learning or developing smaller projects.

Instead, leaning towards a local Kubernetes solution would be much more beneficial. It allows users to experiment, learn, and develop without the pressure of high cloud costs. This approach makes it easier to try different configurations and troubleshoot problems without incurring unexpected expenses. For educational or early-stage projects, this flexibility can be a significant advantage.

2.6.5 Local Kubernetes Cluster

Due to the high cost of managed Kubernetes services, especially for smaller research or academic projects, local Kubernetes setups have become the preferred and practical alternative. Running clusters locally avoids the ongoing expenses of cloud infrastructure and gives developers full control over their environments. While exposing services or configuring networking can take more effort, local clusters help in understanding the inner workings of Kubernetes, making them valuable for hands-on learning and experimentation. Today, local clusters are widely used in educational, testing, and CI/CD environments, making them a state-of-the-art solution for development purposes. This section explores three widely adopted tools used for local Kubernetes deployments: Minikube, kind, and K3s.

2.6.5.1 Minikube

Minikube is one of the most widely used tools for spinning up a local Kubernetes cluster [6]. It supports running a single-node cluster inside a VM or container and works on Win-

dows, macOS, and Linux. Minikube offers a rich set of built-in features like a Kubernetes dashboard, Ingress controllers, metrics-server, and support for multiple drivers (VirtualBox, Docker, etc.). It also allows testing across different Kubernetes versions. The documentation is well maintained, and the active user community makes troubleshooting and learning much easier. These characteristics make Minikube a solid choice for learners, developers, and small project environments.

2.6.5.2 kind

Kind (Kubernetes IN Docker) is a fast and lightweight tool that runs Kubernetes clusters inside Docker containers [18]. It's designed mainly for testing Kubernetes itself or running automated CI/CD pipelines. Kind is quick to start and destroy, making it ideal for scenarios where clusters need to be recreated often. However, it doesn't provide many built-in add-ons, and setting up networking features like Ingress or LoadBalancer typically needs more manual configuration. Kind is more suitable for automated testing than hands-on experimentation with services or full-featured deployments.

2.6.5.3 K3s

K3s is a lightweight Kubernetes distribution developed by Rancher and optimized for low-resource environments such as edge devices and IoT systems [19]. Despite being compact, K3s remains fully Kubernetes-compliant and supports modern features like Helm charts and CRDs. It is often used in production for small clusters, thanks to its low memory usage and simplified installation. While K3s is powerful, it may not be the best starting point for new users due to its minimal abstraction and less emphasis on beginner documentation.

2.6.6 Local Clusters Comparison

Aspect	**Minikube**	**kind**	**K3s**
Setup	Easy	Very Easy	Moderate
Use Case	Learning / Dev	CI/CD Testing	Edge / IoT
Add-ons	Built-in	Manual setup	Limited
Performance	Moderate	Fast	Very Lightweight
Community	Large	Niche	Growing

Table 2.4: Simplified Comparison of Local Kubernetes Clusters

After comparing all three tools, it was decided to proceed with Minikube for this project. It offers the best balance between usability, features, and support. The wide range of add-ons made it easier to test application deployments, networking policies,

and configuration setups. Its clear documentation and active support community also reduces the time spent on troubleshooting. Although kind and K3s are also capable tools, Minikube gave us the flexibility needed to experiment freely in a cost-effective and controlled environment.

ALl of the internal component communication in the local clusters are based in TCP and HTTP 1.1. Neither of the Kubernetes solutions support forwarding QUIC connections natively hence, alternative workarounds are needed.

2.6.7 Custom Resource Definitions (CRDs)

Custom Resource Definitions (CRDs) let users add new types of objects to Kubernetes, such as Gateway API (see section 2.6.10). These custom resources behave like built-in ones (e.g., Pods, Services), allowing teams to manage them using tools like kubectl. A CRD only defines the structure of a custom resource. To make it work, a separate program called a controller watches for changes and performs actions to match the desired state. Controllers are often built using tools like Kubebuilder or Operator SDK.

CRDs are widely used in tools like Prometheus Operator to automate tasks and manage infrastructure. While building controllers can be complex, they allow teams to create powerful, reusable, and automated workflows inside Kubernetes. In this project, CRDs help manage Apache Pulsar clusters automatically within Kubernetes, making deployment and scaling much easier.

2.6.8 Ingress Controllers in Kubernetes

An Ingress Controller is a critical component in Kubernetes used to expose services, APIs, or web applications running inside the cluster to external users over HTTP or HTTPS. It works alongside Ingress resources, which define routing rules, to manage how external requests reach internal services. While Kubernetes provides the Ingress resource as part of its API, it does not ship with a default controller—users must install and configure one manually. Ingress controllers interpret these rules and handle tasks such as URL-based routing, TLS termination, and load balancing, enabling scalable, declarative web and API access. The different types of ingress controllers are as follows:

For platform-independent or more advanced use cases, several third-party ingress controllers are widely adopted. The NGINX Ingress Controller is the most common, known for its flexibility and ease of use across any Kubernetes setup. Traefik is a modern alternative that auto-discovers services and integrates well into dynamic or CI/CD-heavy environments. For API-first architectures, the Kong Ingress Controller doubles as an API gateway, offering authentication, rate limiting, and monitoring features. Istio's Ingress

Gateway, designed for service meshes, provides advanced capabilities like mTLS, circuit breaking, and traffic splitting—ideal for secure, production-grade microservices deployments.

The following is the syntax for ingress controllers

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example-ingress
  annotations:                          # (Controller-specific customizations go here
    <controller-specific-annotations>
spec:
  ingressClassName: <controller-name>  # (Name of the ingress class, like nginx, al
  rules:
  - host: <your-domain.com>
    http:
      paths:
      - path: /                         # (URL path prefix to match)
        pathType: Prefix                # (Usually Prefix or Exact)
        backend:
          service:
            name: <service-name>     # (Your Kubernetes service name)
            port:
              number: <port-number>  # (Service port to route traffic to)
```

Here we would use the custom annotations for the specific ingress we will use.

2.6.9 Ingress Controller Support for HTTP/3 and QUIC

HTTP/3, the latest evolution of the HTTP protocol, brings significant advantages in performance and reliability by using QUIC, a transport protocol that operates over UDP and integrates TLS 1.3. Despite growing interest in HTTP/3 for its benefits—such as reduced latency and improved multiplexing—Kubernetes Ingress Controllers have been slow to adopt it due to technical challenges and architectural limitations.

Most widely-used Ingress Controllers, like NGINX and HAProxy, are based on traditional HTTP/1.1 and HTTP/2 over TCP. Since HTTP/3 relies on QUIC, it requires support for UDP, TLS 1.3, and specialized libraries like BoringSSL. These changes involve substantial rewrites of existing proxying logic and networking layers, making native support difficult. As a result, HTTP/3 functionality in Kubernetes Ingress Controllers

is generally experimental, requiring custom builds, complex configuration, and lack of standardization across implementations.

2.6.9.1 NGINX Ingress Controller and HTTP/3 Support

NGINX Ingress Controller is one of the most widely adopted ingress controllers in Kubernetes [9]. However, its official support for HTTP/3 remains limited and somewhat unclear. The official NGINX Ingress Controller documentation lacks a straightforward guide for enabling HTTP/3, making configuration challenging for users attempting to adopt this protocol.

At the core, HTTP/3 support in the underlying NGINX web server is provided via the `ngx_http_v3_module`, which has been available since NGINX version 1.25.0. However, this module is **not built into the official NGINX package by default**. Enabling HTTP/3 requires compiling NGINX from source with the `--with-http_v3_module` flag and linking it against a QUIC-compatible TLS library such as **BoringSSL** or **QuicTLS**. This setup is more feasible at the web server level than within the Kubernetes-managed ingress controller.

To investigate support at the Kubernetes ingress level, attempts were made to pass relevant parameters via Helm during deployment:

```
helm install ingress-nginx ingress-nginx/ingress-nginx \
  --namespace ingress-nginx --create-namespace
```

This opened TCP ports in the service. To enable QUIC support, UDP traffic must also be allowed to reach the ingress. Therefore, the following patch was applied to the service to expose UDP on port 443:

```
kubectl patch svc ingress-nginx-controller -n ingress-nginx \
  --type='json' -p='[{"op": "add", "path": "/spec/ports/-",
  "value": {"name":"quic","port":443,"protocol":"UDP","targetPort":443}}]'
```

In the latest versions of the NGINX Ingress Controller, the HTTP/3 module appears to be compiled in by default. Additional configurations to expose QUIC via UDP were attempted.

Despite these configurations, the ingress controller does not currently provide sufficient mechanisms to fully inject QUIC and HTTP/3 configuration parameters within a Kubernetes environment, as the NGINX Ingress Controller does not support any specific flags for QUIC behavior. There are **no officially supported parameters** to manage

QUIC-specific settings or negotiate HTTP/3 connections reliably. As a result, while partial support exists at the NGINX core level, the NGINX Ingress Controller's HTTP/3 capabilities remain **experimental and limited** in practical Kubernetes deployments.

2.6.9.2 HAProxy Ingress Controller and HTTP3 Support

HAProxy has introduced experimental HTTP/3 support starting with version 2.6 [20] [21]. Its Ingress Controller can leverage this functionality. This includes enabling UDP listeners, tuning TLS settings, and adjusting protocol stacks manually—none of which are typically available through standard Helm charts.

To test HTTP/3 support in a Kubernetes environment, the following steps were performed:

1. The HAProxy Helm repository was added and updated:

   ```
   helm repo add haproxytech https://haproxytech.github.io/helm-charts
   helm repo update
   ```

2. The HAProxy ingress controller was installed with a custom values file:

   ```
   helm install haproxy-quic haproxytech/kubernetes-ingress -f haproxy-quic-valu
   ```

   In the `haproxy-quic-values.yaml` file, the configuration exposed UDP port 443 and passed the flag `--quic-bind-port=443` to enable QUIC and HTTP/3 support.

3. An `Ingress` resource was applied, using the HAProxy ingress class and pointing to a backend service. A self-signed TLS certificate was used with the hostname `quic-aioquic.com`.

Once deployed, HTTP/3 functionality was tested using `curl`:

```
curl --http3-only --verbose --insecure https://192.168.49.4 \
  -H "Host: quic-aioquic.com"
```

The response confirmed that HTTP/3 was successfully negotiated. However, backend service logs indicated that the request was received using HTTP/1.1:

```
10.244.1.122 - - [07/Aug/2025:12:55:39 +0000] "GET / HTTP/1.1" ...
```

This confirms that HAProxy terminates the HTTP/3 connection and opens a new one to the backend using a lower protocol version.

2.6.9.2.1 Limitations. Although HAProxy can accept HTTP/3 connections from clients, it does not forward them using HTTP/3 to backend services. The ingress controller does not support inspecting or demultiplexing QUIC packets. As a result, HTTP/3 is only supported at the edge, and end-to-end HTTP/3 communication within Kubernetes is not yet feasible using HAProxy.

2.6.9.3 Workarounds

Given the lack of mature, native HTTP/3 support in mainstream Ingress Controllers, it is currently more practical to experiment with HTTP/3 directly at the web server level for Nginx and Traefic which wasnt able to support ingress like ha-proxy. We will look at the webserber Angie which is a fork of nginx especially created for h3 support in section

2.6.10 Gateway API

2.6.10.1 Overview

The Gateway API is a recent advancement in Kubernetes networking designed to address the limitations of the traditional Ingress API. While Ingress has served as the de facto method for managing external traffic into Kubernetes clusters, it suffers from inflexibility, heavy reliance on annotations, and poor support for non-HTTP protocols. The Gateway API introduces a more robust, extensible, and protocol-agnostic model that allows infrastructure operators and application developers to define their concerns independently, thus supporting multi-tenancy and complex networking scenarios more effectively.

One of the defining features of the Gateway API is its emphasis on supporting a wide variety of protocols beyond HTTP and HTTPS, such as TCP and UDP, with emerging support for QUIC and HTTP/3. This makes the API particularly relevant for modern, low-latency, and multiplexed applications that demand richer transport capabilities. The architecture facilitates this through a set of Kubernetes-native Custom Resource Definitions (CRDs), allowing developers and operators to interact with networking policies declaratively and with clear separation of responsibilities.

2.6.10.2 Architecture and Core Components

The Gateway API introduces modular CRDs for fine-grained network traffic control in Kubernetes. GatewayClass defines types of gateways (e.g., Envoy, HAProxy), managed by admins, while Gateway instances configure ports, protocols, and TLS settings. Routing is handled by resources like HTTPRoute, TCPRoute, and UDPRoute, which forward traffic based on specific rules. Additional resources like ReferencePolicy support cross-namespace

and security configurations. This design separates infrastructure and application concerns, improving scalability, security, and maintainability

2.6.10.3 Gateway API Implementation for HTTP3 Support

Since HAProxy successfully worked with Ingress for HTTP/3 forwarding, the next step was to implement the Gateway API using the HAProxy controller.

Installing the HAProxy Gateway Controller

The following steps were performed:

- Added the Helm repository:

```
helm repo add haproxytech https://haproxytech.github.io/helm-charts
helm repo update
```

- Installed the controller with Gateway API support:

```
helm install haproxy-gateway-controller haproxytech/kubernetes-ingress \
  --namespace haproxy-gateway-system --create-namespace \
  --set controller.kubernetesGateway.enabled=true \
  --set controller.kubernetesGateway.gatewayControllerName=haproxy.org/gateway
  --set controller.service.type=LoadBalancer
```

Once deployed, a LoadBalancer service was exposed, including a UDP port for QUIC:

```
NAME                                                TYPE           CLUSTER-IP     EXTER
haproxy-gateway-controller-kubernetes-ingress   LoadBalancer   10.104.20.31   192.1
```

2.6.10.4 Defining the Gateway

The Gateway resource was created to expose HTTPS and QUIC traffic:

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: haproxy-gateway
spec:
  gatewayClassName: haproxy
  listeners:
  - name: https
    protocol: HTTPS
```

```
    port: 443
    tls:
      mode: Terminate
      certificateRefs:
      - kind: Secret
        name: your-tls-secret
  - name: quic
    protocol: HTTPS
    port: 443
    tls:
      mode: Terminate
      certificateRefs:
      - kind: Secret
        name: your-tls-secret
```

2.6.10.5 Defining the HTTPRoute

An HTTPRoute was defined to route requests based on hostname:

```
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: haproxy-http3-route
spec:
  parentRefs:
  - name: haproxy-gateway
    sectionName: https-quic
  hostnames:
  - "gateway-haproxy.com"
  rules:
  - backendRefs:
    - name: http3-test
      port: 80
```

2.6.10.6 Packet Flow Description

The observed packet flow is as follows:

- Client sends request to LoadBalancer service.

- The HAProxy pod receives the request, decrypts it using the TLS certificate.
- It matches the hostname and routes the request based on the Gateway and Route configuration.
- The request is forwarded to the backend service.

2.6.10.7 Operational Separation

The Gateway API introduces a clean separation of responsibilities:

- **Infrastructure providers** manage the GatewayClass resources (e.g., HAProxy, NGINX).
- **Cluster administrators** handle gateway resources (TLS, listeners, ports).
- **Application developers** define HTTPRoute or TCPRoute objects, configuring hostnames and routing rules.

This separation encourages better scalability and maintainability in Kubernetes environments.

2.6.10.8 Observed Limitation in HTTP/3 Stream Handling

While the Gateway API supports HTTP/3 and correctly forwards such requests to backend services, a limitation was observed:

When HTTP/3 requests are made, the Gateway forwards them correctly, showing basic QUIC support. However, similar to the behavior of the HAProxy Ingress Controller, the Gateway API treats the request as a single unit. There is no visibility or control over individual HTTP/3 streams.

This indicates that the Gateway API currently lacks support for stream-level features of HTTP/3. It does not provide insight into or allow manipulation of separate streams within a connection. Therefore, the full capabilities of HTTP/3, such as fine-grained stream-level routing or observation, are not realized.

2.6.10.9 Declarative Configuration vs. Annotation-Driven Models

A major improvement introduced by the Gateway API is its declarative configuration model, which replaces the annotation-heavy approach of the traditional Ingress API. Annotations in Ingress controllers often lack standardization, are implementation-specific, and are difficult to validate, leading to operational ambiguity. In contrast, the Gateway

API uses structured CRDs to define listeners, routes, and policies, making configuration clearer, more consistent, and easier to validate.

Nevertheless, annotations have not been entirely eliminated. They are still used in specific cases, particularly to enable experimental features or for backward compatibility with legacy tooling. For example, enabling QUIC support or selecting specific cipher suites via BoringSSL may still require annotations, depending on the controller. However, these annotations are supplementary rather than central, with the primary configuration expressed through first-class Kubernetes resources. This allows for richer multi-protocol configurations—such as defining a single gateway that supports both TCPRoute and UDPRoute—an arrangement difficult to express using traditional Ingress.

2.6.11 QUIC Library Implementations: Aioquic, Quic-go, and Quiche

The standardization of the QUIC protocol by the IETF has encouraged the development of several independent and open-source implementations to support integration across diverse ecosystems. These implementations serve not only as vehicles for QUIC protocol adoption but also as platforms for experimentation, optimization, and integration into real-world systems. Among the most prominent and actively maintained QUIC libraries are Aioquic, Quic-go, and Quiche, each developed in a distinct programming language and suited to particular performance and deployment goals. This section provides a detailed examination of these libraries, discussing their internal architecture, design philosophies, and practical usage in production or research environments.

2.6.11.1 Aioquic

Aioquic is a Python-based implementation of the QUIC and HTTP/3 protocols developed by aiortc, and is designed to provide asynchronous, non-blocking I/O through Python's asyncio framework. Aioquic is particularly well-suited to academic and research applications due to its readability, high-level abstractions, and ease of integration within Python-based systems. It supports essential QUIC features such as stream multiplexing, TLS 1.3 handshake, connection migration, datagram frames, 0-RTT resumption, and HTTP/3 protocol negotiation.

2.6.11.1.1 Architecture and Design At the core of Aioquic lies an event-driven state machine that responds to received packets and timer expirations. It integrates closely with Python's asyncio event loop, allowing developers to implement clients and servers that handle concurrent connections efficiently using coroutines. The QuicConnection class

encapsulates the transport logic, while the H3Connection class manages the HTTP/3 layer on top of QUIC streams. Developers can subclass the QuicConnectionProtocol to customize behavior for server-side or client-side implementations.

Aioquic also includes built-in support for logging using qlog, a standardized QUIC tracing format. This enhances its suitability for debugging and protocol analysis. In terms of modularity, the library is structured to allow developers to use just the QUIC transport layer or to integrate HTTP/3 as needed, making it flexible for building custom protocols atop QUIC.

2.6.11.1.2 Implementation Example A typical Aioquic application involves initializing a QUIC configuration, creating a connection context, and using asynchronous tasks to send and receive data. Below is a simplified example of a client establishing a secure QUIC connection and sending data over a stream:

```
from aioquic.asyncio import connect
from aioquic.quic.configuration import QuicConfiguration
import asyncio

async def main():
    configuration = QuicConfiguration(is_client=True)
    async with connect("example.com", 4433, configuration=configuration) as protoco
        stream_id = protocol._quic.get_next_available_stream_id()
        protocol._quic.send_stream_data(stream_id, b"Hello, QUIC!", end_stream=True
        await protocol.wait_closed()

asyncio.run(main())
```

This minimal example illustrates Aioquic's ease of use and is indicative of how rapidly developers can prototype with the library.

2.6.11.1.3 Use Cases and Adoption While Aioquic is not yet widely deployed in commercial production environments due to Python's performance limitations, it is frequently used in academic research, protocol testing, and rapid prototyping.

2.6.11.2 Quic-go

Quic-go is a Go-based implementation of the QUIC protocol, maintained by the open-source community and initially developed by lucas-clemente, with substantial contributions from the Chromium networking team. Quic-go is designed for robustness and per-

formance and has been widely integrated into real-world systems, including browsers and cloud-native applications.

2.6.11.2.1 Architecture and Design Quic-go leverages Go's lightweight concurrency model through goroutines and channels, making it highly scalable for managing thousands of concurrent QUIC sessions. The library adheres closely to the IETF QUIC specification and includes a complete implementation of HTTP/3, along with transport features such as packet pacing, congestion control (CUBIC, BBR), retransmissions, connection ID rotation, stateless resets, and forward error correction (FEC).

The internal architecture comprises multiple interacting subsystems: the connection package manages connection-level state, streams handles stream multiplexing, and ackhandler is responsible for packet acknowledgment and retransmission tracking. The integration of Go's cryptographic primitives allows native handling of TLS 1.3 through the crypto/tls library.

2.6.11.2.2 Implementation Example The following snippet illustrates a basic QUIC server using Quic-go, which listens for incoming connections and echoes received data:

```
package main

import (
    "crypto/tls"
    "github.com/quic-go/quic-go"
    "io"
    "log"
)

func main() {
    tlsConf := &tls.Config{
        Certificates: []tls.Certificate{loadTLSCertificate()},
        NextProtos:   []string{"h3"},
    }

    listener, err := quic.ListenAddr(":4242", tlsConf, nil)
    if err != nil {
        log.Fatal(err)
    }
```

```
    for {
        sess, err := listener.Accept(nil)
        if err != nil {
            log.Fatal(err)
        }
        go handleSession(sess)
    }
}

func handleSession(sess quic.Connection) {
    stream, err := sess.AcceptStream(nil)
    if err != nil {
        log.Println(err)
        return
    }
    io.Copy(stream, stream) // Echo
}
```

This example demonstrates how Quic-go abstracts away low-level networking concerns, allowing developers to focus on application logic.

2.6.11.2.3 Use Cases and Adoption Quic-go is embedded in the Chromium browser stack, where it underpins HTTP/3 communication. It has also been adopted in several high-throughput applications, including Caddy, a modern Go-based web server, and FrankenQUIC, a QUIC protocol experimentation platform. Its production-readiness and active maintenance make it an excellent choice for developers building scalable, network-intensive applications in the Go ecosystem.

2.6.11.3 Quiche

Quiche is a QUIC and HTTP/3 implementation written in Rust by Cloudflare, focusing on performance, security, and integration into high-speed, production-grade systems. Quiche is used extensively within Cloudflare's global CDN infrastructure, where it handles millions of QUIC connections across edge locations.

2.6.11.3.1 Architecture and Design Quiche benefits from Rust's safety guarantees and zero-cost abstractions, which make it ideal for low-level network programming. The library is designed to operate as a "no-std" Rust crate, enabling usage in environments

with limited standard library support (e.g., embedded systems or unikernels).

The architecture revolves around a stateful Connection object, which manages packet parsing, stream states, congestion control, and cryptographic negotiation. The library implements advanced features such as 0-RTT data resumption, QPACK header compression, packet coalescing, and stream prioritization. Quiche also offers tight integration with qlog tracing and supports ALPN negotiation for HTTP/3 and custom application protocols.

Quiche is built to be embedded into other C/C++/Rust programs via its Foreign Function Interface (FFI), making it highly versatile.

2.6.11.3.2 Implementation Example A basic HTTP/3 server using Quiche is more complex due to Rust's strict safety model, but the following pseudocode outlines the core flow:

```
let mut conn = quiche::accept(&scid, None, local_addr, peer_addr, &mut config)?;
let (read, _) = socket.recv_from(&mut buf)?;
let recv_info = quiche::RecvInfo { from: peer_addr, to: local_addr };

conn.recv(&mut buf[..read], recv_info)?;

if conn.is_established() {
    if let Ok((stream_id, data)) = conn.readable().next() {
        // Process stream data
    }
}
```

This example shows the control developers have over connection handling and packet flow when using Quiche.

2.6.11.3.3 Use Cases and Adoption Quiche is used in Cloudflare's HTTP/3 infrastructure and has been integrated into major projects such as Nginx, curl, and Wireshark (for traffic analysis). Its adoption by performance-critical applications highlights its production maturity and scalability. Additionally, its inclusion in the QUIC Interop Runner and extensive qlog support make it a valuable reference implementation for QUIC protocol conformance testing.

2.6.11.4 Summary of Libraries

The three libraries—Aioquic [4], Quic-go [22], and Quiche [23] explains the diversity of QUIC implementations, each targeting different domains. Aioquic prioritizes accessibility and research flexibility whereas Quic-go balances performance and developer usability and finally Quiche focuses on high-throughput, secure, production deployments. The choice among them depends on the development environment, performance constraints, and programming language expertise. In the context of this dissertation, Aioquic is selected for the implementation phase because of its easy syntax. ie. Python being easy to understand and configure.

2.6.12 Angie Webserver

Angie is a high-performance, modular web server and reverse proxy developed as a fork of NGINX, with explicit focus on supporting emerging protocols and modern web traffic patterns [24]. One of Angie's key differentiators is its support for HTTP/3 via native QUIC integration. By leveraging QUIC, Angie can handle low-latency, multiplexed connections over UDP, making it suitable for next-generation web workloads. Angie's modular architecture also facilitates extensibility for protocol enhancements and security mechanisms.

In this study, Angie was deployed within a Kubernetes environment using the official container image (docker.angie.software/angie:latest). Configuration was managed via a Kubernetes ConfigMap, specifying server blocks, TLS settings, and QUIC-related directives. Angie successfully forwarded HTTP/3 traffic to backend services, validating its compatibility with QUIC at the connection level. However, its current implementation lacks the ability to perform stream-level inspection or manipulation within QUIC connections. This limits its support for finer-grained traffic control and observability across individual HTTP/3 streams which is feature critical for advanced routing, security policies, and telemetry in production environments.

Despite these limitations, Angie presents a stable and efficient reverse proxy solution for HTTP/3 traffic, suitable for experimental deployments and scenarios where basic QUIC forwarding suffices. Its adherence to evolving protocol standards and active development trajectory make it a compelling candidate for future-ready capable proxy.

2.6.12.1 Angie Configuration for HTTP/3 Forwarding

Angie's architecture supports HTTP/3 by enabling QUIC listeners alongside traditional TCP listeners. The proxy was deployed Angie using the Docker image docker.angie.software/angie:late in a Kubernetes environment, configured via a ConfigMap as shown below:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: angie-config
data:
  angie.conf: |
    worker_processes auto;
    events {}
    http {
      log_format detailed '$remote_addr - $remote_user [$time_local] '
                          '"$request" $status $body_bytes_sent '
                          '"$http_referer" "$http_user_agent" '
                          'protocol=$server_protocol http3=$http3';
      access_log /dev/stdout detailed;
      server {
        listen 443 ssl;
        listen 443 quic reuseport;
        ssl_certificate     /certs/tls.crt;
        ssl_certificate_key /certs/tls.key;
        add_header Alt-Svc 'h3=":443"; ma=86400; persist=1';
        ssl_protocols TLSv1.3;
        ssl_ciphers HIGH:!aNULL:!MD5;
        ssl_prefer_server_ciphers on;
        http2 on;
        location / {
          return 200 "Hello from Angie via HTTP/3!\n";
        }
        location /service {
          proxy_pass https://quic-service.default.svc.cluster.local;
        }
        location /pod {
          proxy_http_version 3;
          proxy_pass https://quic-service.default.svc.cluster.local;
        }
        location /test-h3 {
          add_header X-Test-Protocol $server_protocol;
          add_header X-Test-Http3 $http3;
```

```
        return 200 "This is /test-h3 (HTTP version: $server_protocol, http3: $htt
      }
    }
  }
```

This configuration enables Angie to listen for QUIC traffic on port 443 (listen 443 quic reuseport), advertise HTTP/3 support via the Alt-Svc header, and forward requests to a backend service (quic-service.default.svc.cluster.local) using HTTP/3 (proxy_http_version 3). The Kubernetes Deployment and Service manifests further expose Angie as a Load-Balancer, supporting both TCP and UDP traffic on port 443. This allows it to proxy http3 traffic to backend but faces the same limitation as faced in haproxy ingress controller and gateway api of not having stream level knowledge

2.6.13 Limitation of Modern Webservers for H3 Support

A fundamental limitation observed in both Angie web server is the inability to perform stream-level inspection of QUIC traffic. QUIC, by design, supports stream multiplexing, which allows multiple concurrent and independent streams—such as HTML documents, images, or scripts—to be transmitted over a single connection without head-of-line blocking. However, both Angie and other webservers treat QUIC packets as udp datagrams and do not parse or prioritize individual streams.

This architectural constraint significantly limits their ability to apply intelligent routing, prioritization, or filtering at the stream level. No granular control over the constituent streams and no existing mechanisms existing to allow such modification raises a critical problem.

Consequently, while reverse proxies, ingresses and gateway api support the transport of HTTP/3 traffic, they do not fully exploit QUIC's stream-level features, which restricts their applicability in more advanced use cases such as fine-grained traffic management or application-aware routing.

2.6.14 Load Balancers for HTTP/3 Traffic Management

2.6.14.1 Overview

Load balancers are critical components in modern distributed systems, enabling traffic distribution across backend services to ensure scalability, fault tolerance, and availability. With the rise of HTTP/3, which operates over the QUIC protocol and utilizes UDP, load balancers must evolve to support this shift from traditional TCP-based communication.

Service load balancers distribute incoming network traffic across multiple backend servers, optimizing resource utilization and ensuring fault tolerance. In Kubernetes, the Service resource with a LoadBalancer type exposes applications externally, typically integrating with cloud provider load balancers in cloud environments or requiring alternative solutions in on-premises setups. The adoption of HTTP/3, built atop QUIC, requires load balancers to support UDP traffic, as QUIC's low-latency handshake and stream multiplexing rely on UDP datagrams rather than TCP.

2.6.14.2 MetalLB for On-Premises Load Balancing

MetalLB is a lightweight, open-source load balancer specifically designed for bare-metal Kubernetes clusters that do not rely on cloud providers [13]. It supports both TCP and UDP traffic and integrates seamlessly with Kubernetes via the LoadBalancer service type. In this solution, MetalLB is used to expose HTTP/3 services externally by assigning static IPs from a pre-defined pool. Once configured, it enables Minikube to accept external HTTP/3 connections over port 443, acting as the ingress point to the cluster for QUIC-based traffic. This makes it a practical choice for on-premises testing and development of HTTP/3 applications.

2.6.14.3 Challenges of MetalLB

While MetalLB fulfills the basic requirement of enabling UDP passthrough for HTTP/3 traffic, it does not provide advanced capabilities seen in cloud load balancers. It cannot inspect individual QUIC streams, meaning it treats all QUIC packets as udp datagrams and lacks stream-level routing or prioritization. It also lacks features such as autoscaling, failover, and built-in redundancy, which are critical for production-grade environments. Configuration in metal-lb must be done manually, including the setup of IP address pools and service manifests. MetalLB's compatibility with Minikube makes it suitable for local HTTP/3 testing, where external access and QUIC packet forwarding are necessary but enterprise grade production features are not.

2.6.15 Apache Pulsar for Streaming Use Cases

2.6.15.1 Introduction to Streaming Systems

In order to work with modern transport protocols such as QUIC and WebTransport, which support bi-directional, low-latency, and multiplexed data transmission, a streaming use case is typically required. These protocols are specifically designed to operate within systems that continuously process data in motion, rather than in static batches.

In the context of modern data-driven architectures, streaming systems have thus become foundational to processing high-velocity, continuous data flows in near real-time. Such systems facilitate the ingestion, transformation, and delivery of data as it is generated, supporting critical use cases including real-time analytics, fraud detection, anomaly monitoring, telemetry processing, and event-driven microservices [Kreps et al., 2011]. At the core of these platforms lies a robust messaging infrastructure, which guarantees the reliable delivery and persistence of events between producers and consumers, enabling scalable and responsive data pipelines.

Two prominent distributed messaging systems supporting streaming workloads are Apache Kafka and Apache Pulsar. While Kafka has historically dominated the event streaming landscape, Apache Pulsar presents a rearchitected solution that addresses several of Kafka's limitations, particularly with respect to scalability, multi-tenancy, and storage management.

2.6.15.2 Comparative Evaluation: Apache Kafka vs Apache Pulsar

Apache Kafka and Apache Pulsar are two leading platforms for distributed event streaming, but they differ significantly in architecture and operational capabilities. Kafka adopts a monolithic design where each broker handles both message processing and local storage. This simplifies the system under certain conditions but introduces limitations in scalability, multi-tenancy, and storage flexibility. Kafka's tight coupling of compute and storage means both must be scaled together, increasing operational overhead.

In contrast, Apache Pulsar follows a cloud-native layered architecture, separating compute (stateless brokers) from storage (Apache BookKeeper). This decoupling allows independent scaling, built-in multi-tenancy, native tiered storage, and geo-replication—features that better support modern, dynamic, and multi-tenant workloads. Pulsar's architecture is particularly aligned with containerized environments and streaming use cases involving QUIC/WebTransport, which require persistent, low-latency streams and decoupled resource management.

Thus, while Kafka remains effective for stable, centralized deployments, Pulsar offers greater architectural flexibility and operational efficiency for contemporary streaming systems. The storage-compute separation gives Pulsar an elastic stability and long-term message retention.

2.6.15.3 Apache Pulsar Architecture and Components

Apache Pulsar's design consists of several modular components, each performing a specific function within the messaging and storage lifecycle. These components collaborate to

deliver a scalable, fault-tolerant, and flexible streaming platform.

2.6.15.3.1 Pulsar Brokers Pulsar brokers are stateless services responsible for handling client requests, maintaining topic metadata caches, and managing the coordination of message dispatch between producers and consumers. Brokers do not store message data locally. Instead, upon receiving a message, a broker writes it to BookKeeper and returns acknowledgements once the required replication factor is achieved.

This stateless nature allows brokers to scale horizontally with minimal coordination overhead. It also enhances system reliability, as brokers can fail and recover without requiring data rebalancing. In streaming scenarios—such as telemetry ingestion, application monitoring, and order processing pipelines—Pulsar brokers facilitate consistent and low-latency message routing across topics.

2.6.15.3.2 Apache BookKeeper (Bookies) Apache BookKeeper serves as the storage backend for Pulsar and is comprised of nodes referred to as "bookies." Data in Pulsar is persisted in write-ahead log structures called ledgers, which are divided into segments and distributed across multiple bookies. Each segment is replicated (typically across three nodes) for durability and fault tolerance.

BookKeeper enables Pulsar to support high-throughput writes and provides deterministic data placement. Additionally, tiered storage mechanisms allow the migration of older data to cheaper storage backends, such as Amazon S3 or Hadoop-compatible file systems. This makes Pulsar particularly effective for long-term event retention and replay, common in systems involving historical trend analysis or data backfills.

2.6.15.3.3 Apache ZooKeeper ZooKeeper [14] is used by Pulsar to coordinate metadata operations, including broker and bookie membership, topic partition assignments, and ledger metadata. It ensures consistency in cluster-wide operations and supports automatic leader election and failover scenarios.

Though newer Pulsar versions aim to decouple from ZooKeeper via alternative metadata stores like Oxia, ZooKeeper remains a critical component in most production deployments. In multi-tenant streaming environments, ZooKeeper facilitates dynamic topic creation and fine-grained resource control across tenants and namespaces.

2.6.15.3.4 Pulsar Proxy (Optional) The Pulsar Proxy is an optional stateless component that acts as a reverse proxy between clients and brokers. It is particularly useful in cloud-native environments or scenarios involving network segmentation, where direct broker exposure is undesirable. The proxy simplifies client connectivity by routing traffic

to the correct broker and is commonly integrated with Kubernetes load balancers, such as MetalLB or Ingress controllers, to expose HTTP or gRPC endpoints.

2.6.15.3.5 Pulsar Functions Pulsar Functions is a built-in lightweight serverless framework for stream processing. Developers can implement custom logic in Python, Java, or Go and deploy it directly into the Pulsar ecosystem without managing external processing infrastructure. Use cases include real-time filtering, transformation, enrichment, and aggregation of messages.

Unlike Kafka Streams, which requires additional compute resources and cluster management, Pulsar Functions operate within the broker environment, offering simplified deployments for low-latency stream operations.

2.6.15.4 Challenges

Running Pulsar in a local environment is somewhat complex as it requires management of multiple logically separated brokers and bookies, adding configuration challenges. Also, running it in Kubernetes brings challenges for exposing endpoints for brokers. These are addressed in detail in the implementation section.

2.6.16 Conclusion

The state-of-the-art review highlights significant advancements in real-time communication protocols, container orchestration platforms, and scalable streaming architectures that collectively enable next-generation data-driven applications. WebTransport, leveraging the QUIC protocol, marks a substantial leap forward by providing low-latency, multiplexed, and reliable data streams over the web. Its support for both reliable and unreliable transport modes, coupled with connection migration and efficient congestion control, addresses critical challenges faced by interactive applications such as gaming, video conferencing, and telemetry systems.

Kubernetes, as the premier container orchestration framework, is progressively adapting to the demands of these modern protocols. Despite its inherent robustness, Kubernetes faces challenges in seamlessly integrating UDP-based protocols like QUIC and enabling HTTP/3 traffic, which necessitate novel solutions in load balancing, ingress management, and service discovery. Emerging standards such as the Gateway API and custom resource definitions offer promising avenues to enhance Kubernetes' native support, though practical implementations of these protocols in production environments remains a challenge and an active area of research and development.

In the domain of streaming platforms, Apache Pulsar represents a paradigm shift by decoupling compute from storage, in contrast to the traditional monolithic architecture exemplified by Apache Kafka. This architectural innovation affords Pulsar superior scalability, elasticity, and multi-tenancy capabilities, making it particularly well-suited for complex, cloud-native streaming use cases. Pulsar's design aligns closely with the requirements of modern real-time data processing, including those facilitated by WebTransport and QUIC, where high throughput, fault tolerance, and geo-replication are critical.

Looking forward, the integration of WebTransport with Kubernetes and Apache Pulsar presents a direction for building resilient, high-performance streaming systems. Experimental implementations focusing on these technologies will be essential to validate their interoperability, measure performance gains, and address operational challenges. This research is a good way to improve how we build fast, scalable streaming systems for future real-time applications.

Chapter 3

Problem Formulation

In this chapter, the research problem of this dissertation is determined as the problem of combining modern WebTransport over QUIC protocol [1] with Kubernetes [3]. WebTransport [8], QUIC, Kubernetes are very sophisticated technologies, and integrating the 3 technologies to provide real-time and low latency applications is challenging. This chapter addresses the research gaps based on the State of the Art with additional aspects of it and proposes a new solution to these problems and a clear plan to solve. It also discusses the difference between the proposed methodology and the current methods and provides certain design and implementation objectives in order to obtain a sustainable, production-ready streaming framework.

This chapter effectively explains the problem it wants to address, offers an abstracted solution, tells how the work is different among other solutions and provides a blueprint on how the designed solutions could be attained.

3.1 Identified Challenges

The State of the Art review highlights that there is substantial advancement in modern transport protocols, cloud-based networking as well as streaming platforms. But it also highlights the issues that act against formation of high-performance and real-time systems in kubernetes. The need to conduct this research is motivated by the following challenges, some of which are briefly explained below.

3.1.1 Limited Support for QUIC and HTTP/3 in Kubernetes Ingress Controllers

Experimental QUIC and HTTP/3 support have been added to Kubernetes Ingress Controllers such as NGINX, HAProxy, and Envoy via controller specific annotations such as

`nginx.org/quic: "true"` (Section 2.6.9.4). Nevertheless, such implementations are unreliable and not production ready. Nginx Controller currently does not support proxying and HAProxy fails to proxy streams. The Gateway Application Programming Interface (API) has a more open model, and allows protocols that use UDP such as QUIC (Section 2.6.10.3), and its utility relies on the capabilities of underlying reverse proxies.

Moreover, proxies like Angie and HAProxy along with the ingress controller cannot inspect or manage individual QUIC streams (Sections 2.6.12, 2.6.13, 2.6.14). This limitation prevents features like stream-aware routing, prioritization, etc which are essential for low-latency applications such as real-time streaming or telemetry. As a result, building efficient, stream-aware systems in Kubernetes remains a challenge.

3.1.2 Challenges in Load Balancing for QUIC-Based Traffic

Unlike the traditional TCP-based protocols, QUIC uses UDP, so it becomes a challenge in Kubernetes environments where the traffic is heavily built on TCP traffic. The cloud service providers, such as Google Cloud, has already implemented support of QUIC in their load balancers with protocol downgrade, whereas on-premise load balancer implementations (such as MetalLB) do not have advanced features such as supporting stream-level routing, autoscaling, or failover (Section 2.6.15.4). MetalLB doesnt support quic proxying and treats HTTP as UDP packets. This creates a challenge in managing QUIC-based traffic in a cluster, which is on-prem, local or privately owned, thus encapsulating the scalability of HTTP/3 applications.

3.1.3 Operational Complexity and Observability

QUIC's encrypted headers and multiplexed streams in QUIC improve performance and security at the expense of making traffic inspection difficult. Kubernetes environment being largely based on tcp have weak weak support for QUIC, which makes it harder to debug and analyze performance (Section 2.6.10.6). Even the Gateway API decouples infrastructure and application considerations, a misconfiguration of work makes whole operation within kubernetes more difficult (Section 2.6.10.6). These issues create barriers to building reliable, real-time webtransport stream demultiplexers/Routers in kubernetes.

3.2 Abstracted View of the Solution

The aim of this dissertation is to open a possibility of allowing independent WebTransport streams to be load balanced to various microservices that run on Kubernetes and address

the critical gap in stream-level inspection, routing, and observability. The recommended solution is the use of a custom HTTP/3/QUIC proxy server that is natively integrated with the cluster and multiplexed streams to internal services, based on content or metadata. WebTransport will have each stream as a separate a logical channel which can send its own data multiplexed over a single connection.

Table 3.1 compares the proposed solution with existing approaches, highlighting the unique capabilities.

Table 3.1: Abstracted View of Proposed Solution

Capability Area	**Existing Approaches**	**Our Approach (Custom Proxy Server)**
Stream Visibility	Operate at packet level, no stream awareness	Enable deep stream-level visibility and context recognition
Stream Handling	Treat entire connection as a monolithic flow	Demultiplex and manage streams individually
Application-Layer Intelligence	Minimal insight or control at application layer	Enable content-based routing, security, and monitoring per stream

3.3 Differences from Existing Approaches

Current solutions, such as those using Angie, HAProxy, or traditional Ingress Controllers, route entire QUIC connections to a single backend without inspecting individual streams (Sections 2.6.12, 2.6.13, 2.6.14). Even when HTTP/3 is supported, these systems lack the ability to manage streams independently, limiting their flexibility for advanced use cases. Annotations and third-party plugins provide partial workarounds, but they are often unstable and controller-specific (Section 2.6.9.4).

This problem is solved in the current technologies with Angie, HAProxy, or legacy Ingress Controllers with custom annotations, where the entire full QUIC connection is sent to the same backend without looking at individual streams (Sections 2.6.12, 2.6.13, 2.6.14). These systems can not manage streams independently thus failing to give flexibility to be used in more advanced use cases even when the underlying protocol such as HTTP/3 supports it. Even using workarounds] with annotations and third-party plugins for ingress, which are unstable and controller-specific they still are limitted to forwarding entire Quic connections (Section 2.6.9.4).

In contrast, this dissertation proposes a novel approach by modifying a production-grade QUIC library, such as Aioquic, to expose and manage HTTP/3 streams. The

custom proxy server will:

- Inspect stream metadata or payloads to enable content-based routing.
- Forward streams independently to different Kubernetes microservices.
- Ensure compatibility with WebTransport clients and Kubernetes services.
- Provide a standardized, production-ready framework which can be deployed easily across kubernetes clusters.

This approach fundamentally differs from existing systems as it prioritizes stream-level granular control which no solution provides while offering flexibility and scalability.

3.4 Targets

This section outlines the design and implementation goals for the proposed solution, providing a clear roadmap for achieving the dissertation's objectives.

Design Goals

- **WebTransport Client for Multiplexed Streaming**: Design a WebTransport client that multiplexes audio, video, and chat streams over QUIC to demonstrate real-time, low-latency streaming.
- **Standardized Stream Header Format**: Implement a 32-byte header format to standardize stream identification, length, and type, enabling efficient routing.
- **Single-Node Minikube Cluster**: Use a single-node Minikube cluster with the Docker driver for lightweight, local Kubernetes deployment.
- **Configuration-Driven Components**: Ensure all components support configuration-driven decisions for flexibility and maintainability.
- **HTTP/3 WebTransport Connections**: Establish secure, concurrent HTTP/3 WebTransport connections via MetalLB UDP ingress.
- **Stream-Aware Router**: Build a router that parses stream headers, handles fragmentation, and routes packets based on stream type.
- **Dedicated Microservices for Pulsar Integration**: Create microservices that process data and publish to Apache Pulsar for real-time analytics.
- **Modular Streaming Platform**: Architect a modular, WebTransport-based streaming platform using Kubernetes, Pulsar, and custom proxies.

Implementation Goals

- **Comprehensive Installation and Configuration**: Provide detailed guidance on installing and configuring the solution, including Kubernetes, Pulsar, and the custom proxy.
- **Multiplexed Client Streams**: Develop client-side logic to create and manage live, multiplexed streams (e.g., audio, video, chat).
- **End-to-End Stream Demultiplexing and Routing**: Build software to demultiplex streams and route them to appropriate microservices based on header metadata.
- **YAML-Driven Configurations**: Enable YAML-driven ConfigMaps and Secrets for hot-reloadable, declarative routing and service configurations.
- **Protocol Translation and Analysis**: Use Wireshark to analyze and validate QUIC and HTTP/3 traffic, ensuring correct protocol translation and performance.

3.4 Summary

This dissertation addresses the problem of a lack of solutions to understand Http3 streams within Kubernetes, so that a custom solution creating an HTTP3 server that understands HTTP3 streams can be created to proxy it to other Kubernetes services. This dissertation analyses different ways to achieve the targeted result and proposes a solution "Custom H3 Ingress Server" wherein kubernetes' native objects are leveraged in conjuction with Aioquics quic implementation.

Chapter 4

Design

This chapter outlines the major design decisions taken in this dissertation. The general problem is decomposed into smaller subproblems and each of them is solved one at a time. For every subproblem, the reasoning behind the chosen solution is explained and described using diagrams to understand the flow of communication. Once each part is solved, the solutions are combined to form the final system design

The values taken by major parameters are also described here along with explanation. In the beginning of the chapter, it starts with the design of the WebTransport client, decision made in the process of its development. It then proceeds to explain the architecture of the Kubernetes-based infrastructure upon which the system will be deployed. Then it turns to the essence of the dissertation, namely the WebTransport router and gives detailed design for the same. Subsequently, the chapter will deal with the integration of microservices and Apache Pulsar and the design of this connection. At last, it gives the whole architecture of the system and concludes with a brief conclusion.

4.1 Design of Client

Webtransport Streams were the point of intererst so the first task was to designa specific usecase for our proplem. The usecase of Webtreaming was considered as it would perfectly fit with our problem. From the gaps in k8s existing solutions,it was evident that a client would be required that sends multiplexed data streams to the kuberenetes cluster. For the streaming usecase in javascript client, Webtransport API supports multiplexing streams of data to be sent over a single connection. So there was a need to design the packet structure as to how the packet would be received at the destination end.

4.1.1 Webtransport Streams

Webtransport streams operates atop of Quic protocol which helps achieve enabling multiple concurrent unidirectional data streams to be sent within a single secure transport connection. It implied need for multiple dedicated streams. To fit our streaming usecase we have decided audio,video and chat datastreams. These act as seperate datastreams which would indicidually send with isolation. This circumvents the HOL problem. The following table shows the design for multiple streams to be multiplexed and sent over a single connection.

Stream Name	**Purpose**	**Direction**	**Reliability**	**Latency**	**Notes**
Audio Stream	Transmit PCM audio data	Unidirectional	Medium	High	Timely delivery needed for playback, tolerates minimal loss
Video Stream	Transmit video frames	Unidirectional	Low	High	Frames may be dropped under congestion for real-time playback
Chat Stream	Transmit text messages	Unidirectional	High	Medium	Guaranteed delivery; delay-tolerant, lower volume

Table 4.1: Multiplexed WebTransport Streams over QUIC

With this clearly seperated we can individually holding different data allows to have a clear distinction of usecases allowing us to leverage the true power of multiplexing

4.1.2 Designing the Complete Packet

In this section, the rationale used in the design of the packet structure relying on a WebTransport-based real-time streaming usecase is described. The design emphasizes performance, extensibility and robustness in achieving an efficient data transmission, routing and buffer management in a microservice-oriented architecture.

4.1.2.1 Overview

Each packet consists of a fixed-size header (32 bytes) followed by a variable-length payload.

The packets are built with a infixed size of header (32 bytes) and variable length payload. The header contains metadata to facilitate routing and processing, while the payload carries the actual data, such as audio, video, or chat messages.

4.1.2.2 Packet Details

Field	Offset	Size (bytes)	Description
track_id	0	16	Stream/track identifier
payload_len	16	4	Payload length (big-endian uint32)
track_type	20	12	Type of stream (e.g., video, chat)

Table 4.2: Header Structure

It has 32-byte fixed-size header and variable-length payload; the packet structure is defined as a 32-byte fixed header and an arbitrary-length field. There are three fields entered in the header:

4.1.2.2.1 track$_id(16bytes)$ The `track_id` field is a unique identifier of logical stream or source like "live_video" or "live_chat". It allows the server or router to multiplex multiple logical streams into a single QUIC connection so that the connection can support features such as multi-user streaming. The 16-byte size allows for human-readable identifiers, null-padding for shorter IDs and ensures flexibility and simplicity.

4.1.2.2.2 payload_len (4 bytes, big-endian) The payload length after the header is given by the `payload_len` field. This enables the receiver to allocate buffers accurately and read those exact number of payload bytes thus making the process more efficient and avoiding the reading of excess payload. The cross platform compatibility can be seen in working with the big-endian format, since it presents a standard network byte order. Payloads can be up to 4GB with a 4-byte field, which is sufficient for delivering live streams.

4.1.2.2.3 track_type (12 bytes) The `track_type` element holds the type of an information in the payload i.e. `video`, `audio`, or `chat`. It helps to direct packets to the correct microservice or handler and also enables extensibility by letting new types (e.g. "subtitle" or "control") without having to reshape the packet header. The 12-byte field stores descriptive type names and makes shorter types null-padded, and is consistent.

4.1.2.2.4 Variable-Length Payload The payload holds the actual, e.g. media frames or messages. This architecture enables a wide range of data types and sizes of data, such

as small chat messages, and video frames. The `payload_len` field is explicit and thereby allows the use of efficient fragmentation and reassembly routines because the receiver knows the number of bytes to receive accurately and it improves buffer handling.

4.1.2.3 Benefits

4.1.2.4 Multiplexing and Routing

With the help of `track_id` and `track_type` fields, the router can demultiplex packets that belong to one QUIC connection and send them to the corresponding service, e.g. audio, video, or chat microservice. This aids a modular, service-based architecture, which increases scalability and maintainability.

4.1.2.5 Buffer Management and Fragmentation

The `payload_len` field will enable the router to buffer pending packets until a complete payload is received a reliable processing will be possible. The structure also helps to trace and analyse packet fragmentation which is essential in optimising performance and buffer analysis in real-time streaming.

4.1.2.6 Extensibility

The `track_type` field supports the addition of new stream types, such as file transfer or screen sharing, without modifying the header structure. Older routers or clients can ignore unrecognized `track_type` values, ensuring backward compatibility and supportinng gradual system upgrades.

4.1.2.7 Simplicity and Robustness

Clear length fields do not need delimiters or escape sequences, and minimize the parsing errors. Memory control becomes easy, and the occurrence of events such as buffer overflows is avoided by means of null-padding in fixed-size fields, which adds robustness to the system.

4.1.2.8 Performance and Efficiency

The 32-byte fixed-size header aligns with CPU word sizes and helps in optimizing memory access and parsing efficiency. The binary format minimizes overhead compared to alternatives like JSON or Protocol Buffers. This ensures high performance under high load.

4.2 Kubernetes Design

As highlighted in the State of the Art, we proceeded to work with Minikube cluster

4.2.1 Driver Design for Kubernetes Cluster

Post the selection of a kubernetes solution for the local setup. A decision was required as to how the architecture of the cluster should look like. Questions like single node, multiple node, how control plane and data plane would look line needed answers . Minikube clusters can be created with several drivers. Few of them are docker driver, Virtual Machine Drivers. Docker Driver was chosen because of several reasons such as follows:

4.2.1.1 No Virtual Machine (VM) Overhead

Virtual Machine management can become challenging due to additional configuration management, resource constraints and increaased complexity. Linux natively support containers which allows for containerization eliminating the need for virtualization.

4.2.1.2 Less Resource Usage

Secondly the machine being used is based on linux mint which has faster bootstrap times, lower cpu and memory comsumption and better performance when compared to windows machines.

4.2.1.3 Logical Seperation

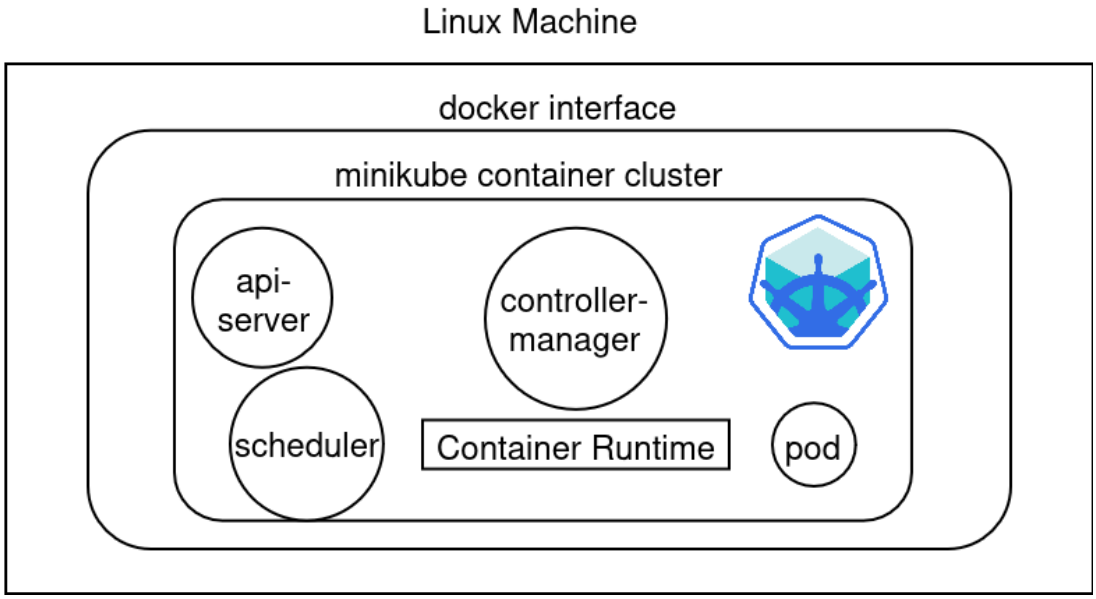
Docker runs on linux kernel whereas minikube when using docker driver would create a logical seperation by running in a seperate container which simplifies and adds necessary abstraction

4.2.1.4 Shared Networking

The services which would run inside Minikube cluster can be accessed with other applications in the system.

The following figure depitcs the architecture of minikube what would be constructed. It is to note that with docker as the driver, minikube runs on a single container within the docker interface which allows it to use interact with the host linux machine through the network interface. For simplicity and avoidance of configurational challenges, the solution is running on one node where the data plane and control plane both runs on a single node. This is not a production setup but ensures a clear development setup.

Figure 4.1: Design of Minikube

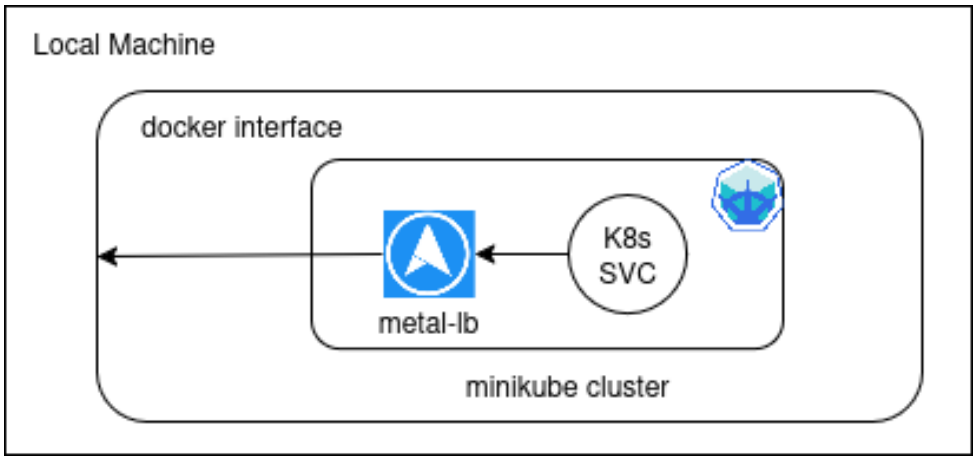


Each component specified in the state of the art for kubernetes will be deployed into this cluster via minikube. This design ensures seamless deployment, developer friendly and extensible scalable design.

4.2.2 Network Exposure Design in Kubernetes

The ingestion of UDP traffic into Kubernetes was approached with the intent to provide a service load balancer and an IP address. This would allow us to create an endpoint for our application, enabling Kubernetes to proxy to the microservices with the help of the built application. Previous chapters discussed exposing UDP traffic as the first challenge to overcome for the realization of our final goal. In this section, the design inspiration stemmed from observing the large adoption of cloud networks and how they exposed Kubernetes services to the outside world. It became evident that there was a need for an external load balancer, which would assign an IP to our service load balancer. The Kubernetes solutions for exposing services in a local setup, as discussed in the state of the art, were limited. Hence, a solution utilizing MetalLB was proposed. MetalLB is a bare-metal load balancer for Kubernetes that allowed us to provide an entry point to our application. The following architecture shows how MetalLB was leveraged

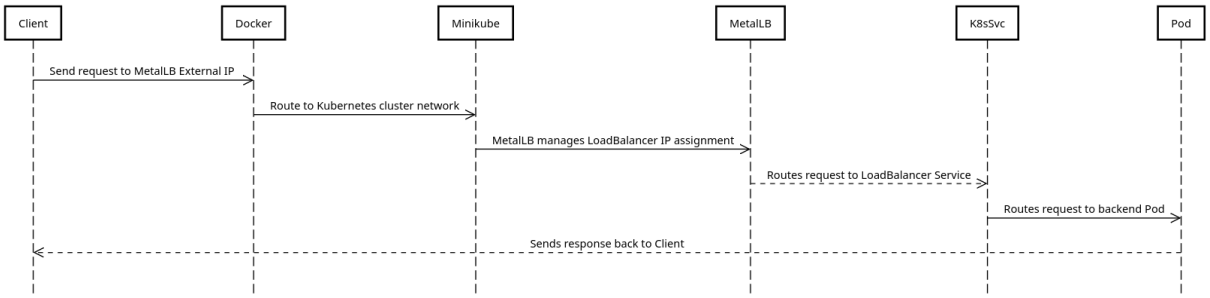
Figure 4.2: Design of Metal-lb



From the figure, we can see that MetalLB is intended to be the frontend of our Kubernetes Cluster, ingesting data. It's important to note that MetalLB acts as a Layer 4 network load balancer, which allows it to proxy raw UDP packets into our cluster. This solves the problem of ingestion. The traffic will travel to and from the service, to MetalLB, and then to the outside cluster, and vice versa.

4.2.2.1 Sequence Flow for metallb

Figure 4.3: Sequence flow for metal-lb



The client sends a request to the MetalLB IP address. The request is then forwarded through the Docker interface, as MetalLB is deployed on the Minikube cluster, which is running in a Docker container. MetalLB receives the IP address and proceeds to forward the request to the Kubernetes service. This design successfully exposes the Kubernetes service to the outside cluster and allows for the forwarding of raw TCP and UDP packets to our applications.

4.3 Design of the Routing and Processing System

This section presents the design of the routing and processing system for a WebTransport stream-aware routing platform. Design focuses on optimization in configuration, connection managements, packets processing, use of metrics, integration of various microservices and flexibility in command line to facilitate a scalable and resilient streaming design.

4.3.1 Configuration-Driven Routing Architecture

The system adopts a configuration-driven routing approach that allows for dynamic changes and streamlined service management. A central 'ServiceConfig' structure defines essential parameters—such as host, port, and endpoint—for each microservice (e.g., audio, video, chat). These configurations are written in a YAML file that is loaded during runtime using a dedicated 'ConfigManager' component in Kubernetes. This component enables hot-reloading, so the routing logic can be changed without restarting services. The routing system can therefore be extremely flexible and extensible and easily reconfigured to accommodate emerging needs or varying deployment environments during operation of the system.

The architecture is intended to operate in Kubernetes and utilizes Kubernetes constructs such as ConfigMaps and Secrets to pass routing definitions and TLS credentials to the downstream code.As far as it is concerned, the Router Config is a central configuration file that encapsulates the rules of service routing and protocol behaviors. The router serves as a fast proxy, reading the configuration from this file and redirecting client search requests to the correct backend microservices. This decoupled and declarative model leads to routing behavior that is fully controlled entirely outside of the application's code. This approach aligns with CI/CD principles. The system supports dynamic scaling, safe updates, and operational agility, thereby enabling the router to be deployed in modern cloud-native operations that demand responsiveness and maintainability.

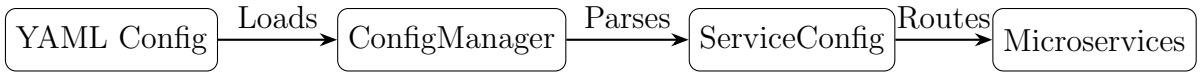


Figure 4.4: Configuration-Driven Routing Flow

The figure illustrates the flow from a YAML configuration file to the `ConfigManager`, which parses and stores settings in `ServiceConfig` structures, enabling routing to microservices. Arrows indicate the sequential process of loading, parsing, and routing configuration data.

4.3.2 WebTransport Protocol Handling

The WebTransport protocol handling system manages QUIC/HTTP3 connections and streams, ensuring reliable communication for real-time streaming. A `WebTransportRouter` component handles protocol negotiation, stream creation, and HTTP/3 request processing, maintaining connection state. Each QUIC stream is assigned a dedicated buffer to manage packet reassembly and analysis, ensuring efficient handling of incoming data. This design supports high-throughput streaming while maintaining low latency and robust connection management.

Figure 4.5: WebTransport Protocol Handling

The figure depicts a client sending data via QUIC/HTTP3 to the `WebTransportRouter`, which assigns streams to dedicated buffers for processing. Arrows show the flow of data from the client to the router and then to stream buffers.

4.3.3 Packet Parsing and Routing

The packet parsing and routing system is designed to efficiently process incoming packets by extracting headers and forwarding payloads to the correct microservices. A fixed 32-byte header enables fast and reliable parsing, containing fields like `track_id`, `payload_len`, and `track_type`. The payload is extracted based on the `payload_len` field and routed to the appropriate microservice using the `track_type` and configuration data. This design ensures low-latency processing and accurate routing in a multiplexed streaming environment.

Figure 4.6: Packet Parsing and Routing

The figure shows an incoming packet being processed by the packet parser, which extracts the header and routes the payload to microservices. Arrows indicate the parsing and routing steps.

4.3.4 Metrics Logging

The structure of metrics logging is clearly elaborated in the evaluation (chapter 6). This part contains the overview of the design

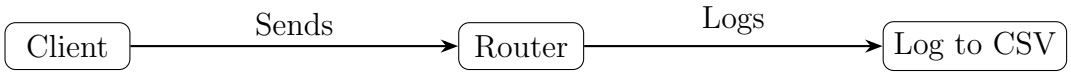


Figure 4.7: Client Request Routing and Logging

The figure illustrates the system collecting metrics via the `MetricsLogger`, which logs data to CSV and log files. Arrows represent the flow of metric collection and logging.

4.3.5 Microservice Proxying

The microservice proxying system forwards packets to the appropriate microservices with robust error handling and retry mechanisms. It supports flexible data formats, including binary and JSON, to accommodate different service requirements. Custom headers can be added to packets for extensibility and debugging, enabling traceability and future enhancements. This design ensures reliable communication between the router and microservices, supporting a modular and scalable architecture.

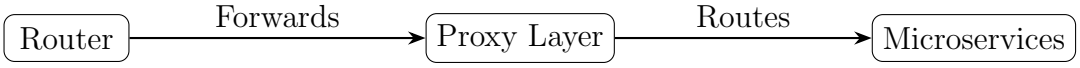


Figure 4.8: Microservice Proxying

The figure shows the router forwarding packets to a proxy layer, which routes them to microservices with error handling. Arrows depict the forwarding and routing process.

4.4 Pulsar Integration Design

After processing packets such as chat messages, audio streams, or video data, each microservice can optionally publish the output to an Apache Pulsar topic. This mechanism decouples real-time data processing from downstream distribution analytics, storage, or machine learning. The messages of each particular service are sent to a specifically-named topic (e.g., chat-topic, video-topic), which guarantees a separation of concerns as well as scalability.

This design enables easy integration with real-time data consumers, allows for per-service configuration of Pulsar usage and allows for flexibility in order to enable/disable publishing dynamically.

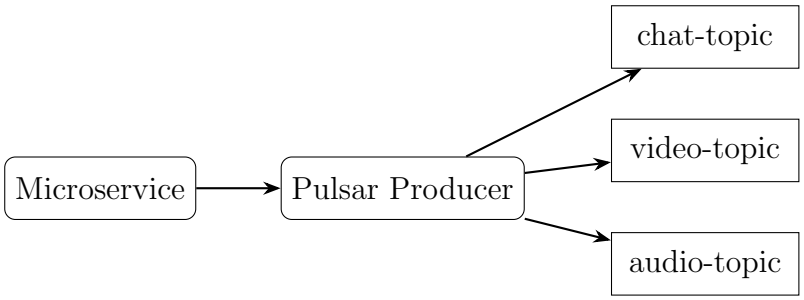
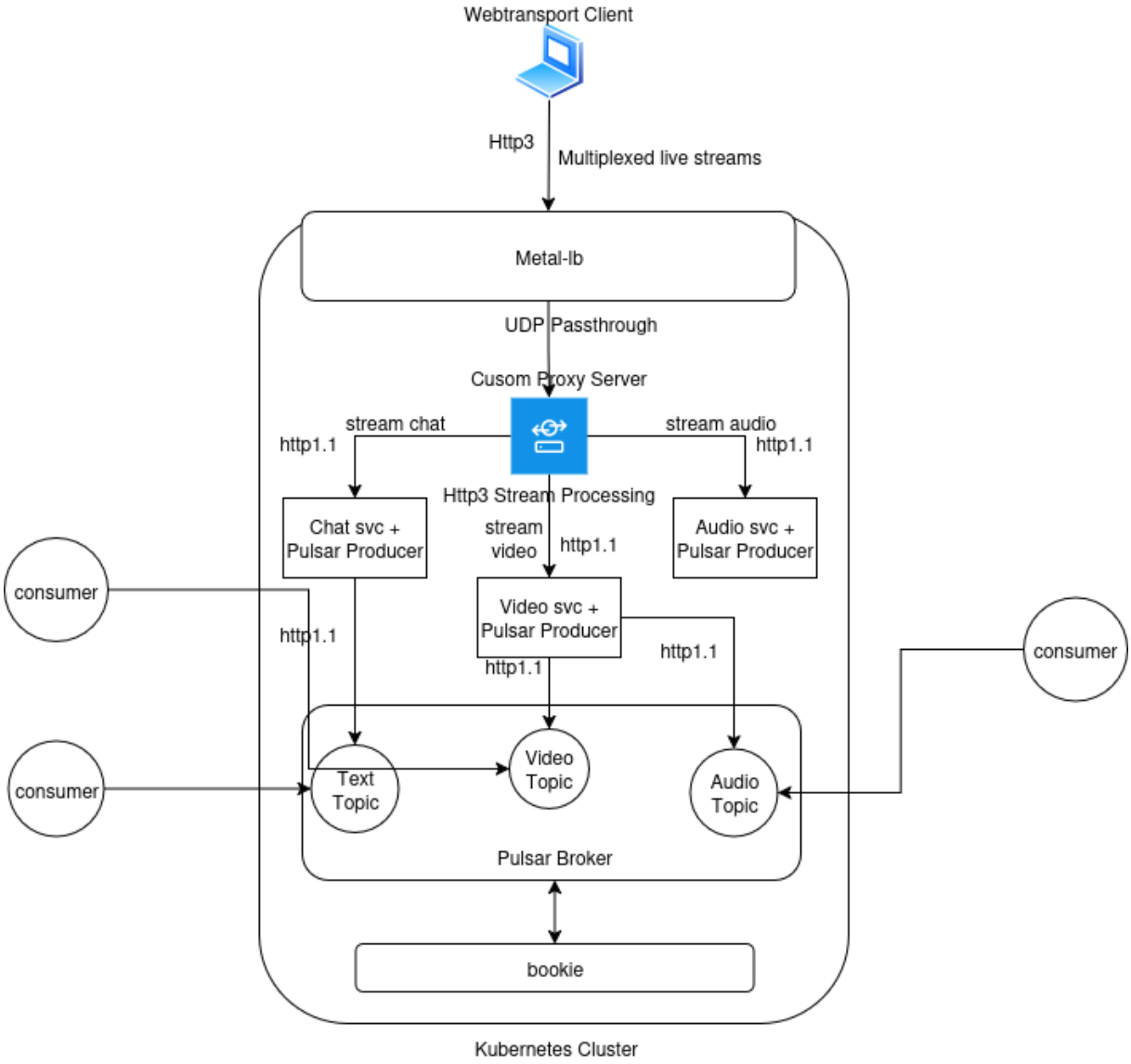


Figure 4.9: Microservice to Pulsar Topics Integration

4.5 Complete System Architecture

The following architecture is a high-level technical architecture diagram which entails the complete proposed solution. The sections above delved deep into each components individually and this is the final high-level overview of our dissertation

Figure 4.10: System Architecture



The proposed architecture system is a real-time streaming system based on a modular structure built to support scaling, low latency communication, and efficient data processing and distribution within a cloud-native infrastructure. It starts with ingest layer where client sends streams of data with a webtransport client connects through HTTP/3 a QUIC-based protocol allowing to establish a single transport connection but with multiplexing multiple data streams in the connection video, audio, and chat. The load balancer in the Kubernetes cluster is MetalLB, which is exclusively enabled to provide UDP passthrough for the minikube ckuster to facilitate the forwarding of QUIC connection (entire UDP packets) to the internal components. The system is centered around a custom proxy server that will terminate the HTTP/3 connection, demultiplexes

the stream it receives, and forwards each individual data stream to the appropriate microservice over HTTP/1.1. This server effectively acts as both a protocol gateway and a routing mechanism.

The application layer is based on different microservices of chat, video, and audio developed which is responsible for handling its own set of data and are independent from each other. These microservices publishes the data to Apache Pulsar after processing, acting as a producer in the event streaming system. The Pulsar cluster will have a broker handling the routing and dispatching of messages, different topics that logically segregate each type of data (Text Topic, Video Topic, Audio Topic) and it will have a persistent storage backend consisting of the Bookie pods of Apache BookKeeper so that persistant data storage and recovery from faults are possible. Finally, on the consumption level, independent consumer applications subscribes to the Pulsar topics and receive data asynchronously. This decouples the data ingestion and downstream processing, allowing to enable use cases of real-time streaming, analytics or storage with modularity and scalability in the system.

4.6 Summary

The proposed system is a real-time streaming architecture that is modular and scalable, designed to support low-latency multimedia communication using WebTransport, Kubernetes, and Apache Pulsar. The architecture breaks the problem space into smaller modules which are developed independently and then combined to form a whole solution. On the client side, WebTransport is used to send data, and WebTransport runs on QUIC to enable data multiplexing across different data streams in a single connection i.e. audio, video and chat. The design will prevent Head-of-Line blocking. A typical packet transmitted has a 32-byte header composed of track_id, payload_len and track_type fields and a variable length payload. The format is capable of extending; it is useful in routing for diversity of data types.

Inside the Kubernetes cluster, a single-node Minikube setup is used for simplicity, leveraging the Docker driver to eliminate virtual machine overhead and enable shared networking. MetalLB is used as a bare metal load balancer that exposes QUIC-based UDP traffic to components of the kubernetes cluster, enabling the external WebTransport clients to connect to internal services. Its routing logic is config-driven, user-managed through YAML-based ConfigMaps and allows for dynamic updates and reloading of routing configurations without system restarts. The core of this solution is the WebTransportRouter, a custom component that accepts and deals with QUIC/HTTP3 streams by attaching each track with its own explicitly associated buffer of streams. This guarantees

that audio, video and chat information is manipulated in a low latency basis for redirections. Once the component performs initial parsing, it sends each stream to its respective microservice through HTTP/1.1 via a proxy server that acts as a reliable gateway between the transport and application layers.

At the application layer, separate stateless microservices are responsible for processing specific types of data such as audio, video, or chat. Each microservice publishes the processed data to its corresponding Apache Pulsar topic—for instance, chat-topic or video-topic—thereby enabling a decoupled and scalable processing model. Apache Pulsar is used as the main system for handling event streams. It consists of a broker which receives data for a specific topic and it stored multiple topic's logical association, with each topic keeping different types of data separate (like audio, video, or chat). To ensure data is safely stored and can be recovered if something goes wrong, Apache BookKeeper is used in the background through components called Bookie. On the receiving side, different consumer applications connect to Pulsar and subscribe to specific topics to get the data they need. By keeping the parts that send, process, and receive data separate, the system stays flexible, reliable, and easy to scale for real-time streaming or analytics tasks.

Chapter 5

Implementation

5.1 Introduction

The implementation chapter translates the proposed WebTransport/QUIC streaming architecture in Chapter 4 into a fully working, reproducible system. It begins by preparing the local development environment—installing Minikube [6] with its Docker driver on Linux Mint, enabling MetalLB [13] for external Load-Balancer Internet Protocol (IP) addresses, and configuring Wireshark [25] together with self-signed Transport Layer Security (TLS) certificates so that encrypted QUIC traffic can be captured and inspected. A Domain Name System (DNS) entry in /etc/hosts binds the synthetic domain quic-aioquic.com to the cluster's MetalLB range, ensuring that both browser clients and backend services resolve the same address. With the groundwork in place, the chapter progresses through three tightly coupled tiers: (1) a browser-based WebTransport client that captures video, audio, chat, file and screen-share data, packetises each stream, and transmits them over independent unidirectional QUIC streams; (2) a Python router built on aioquic [4] that receives these streams, parses the custom 32-byte packet header, and routes payloads to backend microservices via hot-reloadable Yet Another Markup Language (YAML) rules; and (3) a set of generic, YAML-configured microservices that expose HyperText Transfer Protocol (HTTP) endpoints, optionally forward data to Apache Pulsar, and run inside Minikube as scalable Deployments. Finally, the entire stack—router, microservices, and Pulsar—is containerised and deployed through Kubernetes manifests that include Secrets for certificates, ConfigMaps for live configuration, and LoadBalancer Services for external reachability, leveraging Apache Pulsar [5] for message streaming.

5.2 Installation of Minikube

The host operating system of the machine at work is Linux Mint, a Debian-based operating system. The following steps outline the installation process tailored for this specific platform, ensuring a smooth setup for running a local Kubernetes cluster using Minikube.

To install Minikube, execute the following commands in the terminal:

```
curl -LO https://github.com/kubernetes/minikube/releases/latest/download/minikube-l
sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd6
```

After executing these commands, Minikube is successfully installed and operational. To verify the installation, start Minikube with the following command:

```
minikube start
```

Note that the `minikube start` command uses the Docker driver by default. Minikube runs within a Docker container, which provides logical separation from the host system. This containerized approach ensures efficient resource utilization and isolation for the Kubernetes environment.

```
sibin@sibin-ThinkPad-T495:~$ minikube ssh -- free -m
               total        used        free      shared  buff/cache   available
Mem:           13853        9609         160          95        4084        3828
Swap:           2047           0        2047
sibin@sibin-ThinkPad-T495:~$ minikube ssh -- lscpu
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          43 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   8
  On-line CPU(s) list:    0-7
Vendor ID:                AuthenticAMD
  Model name:             AMD Ryzen 7 PRO 3700U w/ Radeon Vega Mobile Gfx
    CPU family:           23
    Model:                24
    Thread(s) per core:   2
    Core(s) per socket:   4
    Socket(s):            1
```

Figure 5.1: Minikube CPU and Memory Usage

Since Minikube operates within a container, it leverages the host operating system's kernel and configuration by default. This setup allows the Kubernetes cluster to access additional resources from the host, optimizing performance and scalability for local development and testing.

It is important to note that `kubectl`, the Kubernetes command-line tool, is automatically installed alongside Minikube. To confirm the presence of pods in the cluster, you can run the following command:

```
kubectl get pods
```

The Minikube cluster and Kubernetes have been successfully installed and configured on the system. This setup provides a fully functional local Kubernetes environment, ready for deploying and managing containerized applications.

Figure 5.2: Verification of Kubectl Presence

5.3 Configuration of MetalLB

MetalLB is a load balancer that operates at Layer 4, which is essential for exposing all services of type LoadBalancer in our Kubernetes cluster. Minikube provides a straightforward installation process for MetalLB through its addons feature.

To view the available addons provided by Minikube, execute the following command:

```
minikube addons list
```

This command lists several addons provided by Minikube. Among them, you will notice that MetalLB is disabled by default. To enable it, run the following command:

```
minikube addons enable metallb
```

MetalLB requires a range of IP addresses to assign to services created within the Kubernetes cluster. When the Minikube container was started with the `minikube start` command, it created a network interface in the background for the Minikube (i.e., Kubernetes) environment. This can be verified by running:

```
ifconfig
```

For the machine in use, the network interface was named `br-5d..` and had the following values: `inet 192.168.49.1`, `netmask 255.255.255.0`, and `broadcast 192.168.49.255`. This indicates that the last octet is open from 0 to 255, creating a subnet that can be represented as `192.168.49.0/24`.

To configure MetalLB with an appropriate IP address range, execute the following command:

```
minikube addons configure metallb
```

During the configuration process, you will be prompted to enter the start and end IP addresses for the load balancer:

```
Enter Load Balancer Start IP: 192.168.49.1
Enter Load Balancer End IP: 192.168.49.250
```

This configures the MetalLB load balancer to operate within the specified IP address range, enabling it to assign IPs to LoadBalancer-type services in the Kubernetes cluster.

5.4 Wireshark for Network Traffic Capture

To install Wireshark and successfully capture network traffic on a Linux Mint system, the following process was followed.

5.4.1 Run Commands

First, install Wireshark using the package manager by executing the following command:

```
sudo apt install wireshark
```

Once installed, launch Wireshark by running:

```
wireshark
```

This command opens the Wireshark application. Upon launching, Wireshark prompts you to select a specific network interface to monitor. Once an interface is selected, Wireshark begins capturing real-time traffic to and from the chosen network interface.

5.4.2 Logging Chromium Client TLS Keys

To log the TLS session keys for the Chromium client, the following steps were performed to ensure proper configuration and logging of the keys for use in decrypting network traffic.

First, terminate any running instances of Chromium by executing the following command:

```
pkill chromium
```

Next, set the environment variable to specify the path for the TLS key log file:

```
export SSLKEYLOGFILE=$HOME/premaster-secret/sslkeylogfile.log
```

Then, restart Chromium to begin logging the TLS keys to the specified path.

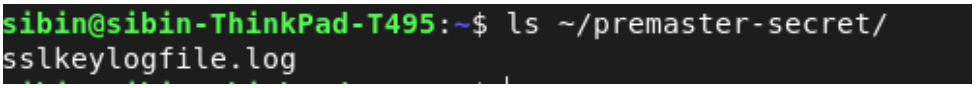


Figure 5.3: Chromium Client's SSL Key Log

5.4.3 Configuring TLS Keys in Wireshark

This process successfully logs the TLS session keys to the designated file, enabling the decryption of TLS-encrypted traffic captured from the Chromium client.

It is important to note that higher-level application protocols, which are encrypted using the standard TLS protocol, can be captured but cannot be inspected directly due to their encryption. To decrypt such communications, TLS session keys are required. These session keys can be logged from the client or server and loaded into Wireshark for decryption.

To load the TLS session keys, follow these steps in Wireshark:

1. Navigate to `Edit -> Preferences -> Protocols -> TLS`.
2. In the TLS settings, locate the `Pre-Master Secret Log Filename` field.
3. Load the TLS session keys by specifying the path to the log file containing the keys.

This process enables Wireshark to decrypt the TLS-encrypted data captured over the network, allowing for detailed inspection of the communication.

5.5 Certificate Management Process

To enable secure communication with the state-of-the-art WebTransport Streams HTTP/3 Router, it was determined that a certificate is essential, as data transmission over the specified protocol requires encryption. Since acquiring a domain was not feasible, the approach of creating a self-signed certificate was adopted.

The following command was used to generate a self-signed certificate for a custom domain:

```
openssl req -x509 -newkey rsa:4096 -keyout tls.key -out tls.crt -days 365 -nodes -s
```

This command instructs OpenSSL to create an RSA key pair, generating both a certificate (`tls.crt`) and a private key (`tls.key`) with an expiration period of 365 days. The `-nodes` flag ensures the private key is not encrypted, and the Subject Alternative Name and Common Name (CN) are set to the custom domain `quic-aioquic.com`.

To utilize the certificate, the client must forward requests to the specified domain, and the client itself must trust the domain. Since the client in use was the Chrome browser, the self-signed certificate was imported through the browser's settings to establish trust.

The certificate can be imported by following these steps in Chrome:

1. Navigate to `Settings -> Privacy and Security -> Manage Certificates`.

2. Under the `Installed Certificates by You` section, select `Import`.

3. Follow the prompts to upload the `tls.crt` certificate file.

This process ensures that the Chrome client trusts the self-signed certificate, enabling secure communication with the WebTransport Streams HTTP/3 Router.

5.6 DNS Entry Configuration

To enable domain name resolution for the local setup, a DNS entry must be configured. With the certificates properly structured, requests are explicitly checked for the specified domain. On a Linux machine, this configuration is achieved by modifying the `/etc/hosts` file to include a DNS entry that maps the domain to a specific IP address.

For the system in use, the following line was added to the `/etc/hosts` file:

```
192.168.49.201 quic-aioquic.com
```

This entry maps the domain name `quic-aioquic.com` to the IP address `192.168.49.201`, ensuring proper domain name resolution for the local setup.

5.7 Client Construction

The construction of the client was accomplished using several core components, each designed to ensure a robust and user-friendly streaming experience. These components are detailed below.

5.7.1 User Interface Design

The client's user interface is created for clarity and functionality, offering an intuitive overview of the streaming process. The layout is structured using a responsive **CSS Grid** (`.container`), which divides the main content into distinct panels for streamlined user interaction.

- **Video & Audio Stream Panel** (`.panel`): This panel prominently displays the local video feed (`<video id="localVideo">`) and includes controls for initiating or stopping streaming, toggling video quality, and starting or stopping optional file and screen data streams.

- **Live Chat Panel** (`.panel`): This features a scrollable message display area (`.chat-messages`) and an input field with a send button (`.chat-input`) for real-time text communication.

- **Metrics Display** (`.metrics`): This provides at-a-glance performance indicators, including the count of video frames, audio packets, chat messages sent, and the calculated data transfer rate.
- **Status Bar** (`#status`): This offers immediate feedback on the WebTransport connection state.
- **System Logs Panel** (`.logs`): This acts as a console, displaying critical information about stream creation, packet transmission, and error events, which is essential for monitoring and debugging during development and testing.

The visual design adopts a dark theme using CSS, enhancing readability in low-light conditions and providing a modern aesthetic. Buttons are styled with distinct colors (`.btn-start`, `.btn-stop`, `.btn-toggle`) to clearly indicate their actions and current states (e.g., `disabled` styling).

5.7.2 WebTransport Connection Setup

Establishing a reliable connection to the WebTransport server is the foundational step for all streaming operations. This is managed within the `WebTransportStreamer` class's `startStreaming()` method. The connection is initiated by creating a new `WebTransport` instance, specifying the server URL (`https://quic-aioquic.com:4433/wt`), which points to the secure QUIC endpoint of the server.

```
this.transport = new WebTransport(this.serverUrl);
```

To ensure robust operation, event listeners are set up for **connection state changes** (`this.transport.onstatechange`) and **errors** (`this.transport.onerror`). These listeners are critical for monitoring the connection's health:

- `onstatechange`: Updates the UI status and triggers `stopStreaming()` if the connection `state` transitions to `'closed'`, indicating a graceful shutdown or an unexpected termination.
- `onerror`: Catches any underlying transport errors, logs them to the console, and initiates `stopStreaming()` to clean up resources.

```
this.transport.onstatechange = () => {
    this.log(`Transport state: ${this.transport.state}`, 'info');
    if (this.transport.state === 'closed') {
        this.stopStreaming();
```

```
    }
};
this.transport.onerror = (event) => {
    this.log(`Transport error: ${event.message}`, 'error');
    this.stopStreaming();
};
```

The `await this.transport.ready;` promise ensures that the application waits until the WebTransport connection is fully established before proceeding with media capture and stream creation, preventing operations on a disconnected transport. This ensures a stable foundation for subsequent streaming activities.

5.7.3 Media Capture and Stream Initialization

Once the WebTransport connection is ready, the client proceeds to capture local media and prepare the data streams for transmission.

5.7.3.1 Video/Audio Capture

Media access is requested using the **WebRTC** `getUserMedia` **API**, which prompts the user for permission to access their camera and microphone. The `constraints` object defines the desired media properties:

- `video`: Configured for either `640x480` at `15fps` (Standard Quality) or `1280x720` at `30fps` (High Quality), controlled by the `isHighQuality` flag.
- `audio`: Includes `echoCancellation` and `noiseSuppression` to improve audio clarity.

```
const constraints = {
    video: {
        width: this.isHighQuality ? 1280 : 640,
        height: this.isHighQuality ? 720 : 480,
        frameRate: this.isHighQuality ? 30 : 15
    },
    audio: {
        echoCancellation: true,
        noiseSuppression: true
    }
```

```
};
this.mediaStream = await navigator.mediaDevices.getUserMedia(constraints);
this.elements.localVideo.srcObject = this.mediaStream;
```

The captured `mediaStream` is assigned to the `<video id="localVideo">` element, allowing the user to see their own video feed. A `Promise` is used to wait for the video element to trigger `loadeddata`, ensuring the media stream is fully ready before processing.

5.7.3.2 Stream Creation

WebTransport's strength lies in its support for **multiplexing** via individual streams over a single connection. The client leverages **unidirectional streams** (`createUnidirectionalStream()`) for each distinct data type:

- `this.videoStream`: For outgoing video frames.
- `this.audioStream`: For outgoing audio packets.
- `this.chatStream`: For outgoing chat messages.
- `this.fileStream`: For sending arbitrary file data.
- `this.screenStream`: For sending screen share data.

```
this.videoStream = await this.transport.createUnidirectionalStream();
this.audioStream = await this.transport.createUnidirectionalStream();
this.chatStream = await this.transport.createUnidirectionalStream();
```

By using separate unidirectional streams, the client can send different types of data independently without head-of-line blocking, enhancing real-time performance and simplifying server-side processing.

5.7.4 Packet Construction and Sending

A custom packet format is implemented to encapsulate various data types for transmission over WebTransport streams, allowing the server to easily identify and process incoming data.

5.7.4.1 Packet Format

Each packet sent over a WebTransport stream adheres to a predefined format consisting of a **32-byte header** followed by a variable-length **payload**. The header provides essential metadata:

- `track_id` **(16 bytes)**: A string identifier (e.g., `'live_video'`, `'test_audio'`, `'live_chat'`) used by the receiver to identify the packet's content within its stream, padded with null bytes if shorter than 16 bytes.
- `payload_length` **(4 bytes)**: A 32-bit unsigned integer (big-endian) indicating the exact size of the payload in bytes, crucial for correct payload reading.
- `track_type` **(12 bytes)**: A string (e.g., `'video'`, `'audio'`, `'chat'`, `'file'`, `'screen'`) classifying the general type of data, padded to 12 bytes.

The `buildPacket()` function assembles this structure:

```
buildPacket(trackId, trackType, payload, forceSize = null) {
    let finalPayload = payload;
    if (forceSize && forceSize > payload.length) {
        finalPayload = new Uint8Array(forceSize);
        finalPayload.set(payload, 0);
        for (let i = payload.length; i < forceSize; i++) {
            finalPayload[i] = i % 256; // Padding pattern
        }
    }

    const header = new ArrayBuffer(32);
    const view = new DataView(header);
    // Set track_id (bytes 0-15)
    new Uint8Array(header, 0, 16).set(this.encoder.encode(trackId.substring(0, 16))
    // Set payload_length (bytes 16-19)
    view.setUint32(16, finalPayload.length, false); // false for big-endian
    // Set track_type (bytes 20-31)
    new Uint8Array(header, 20, 12).set(this.encoder.encode(trackType.substring(0, 1

    const packet = new Uint8Array(header.byteLength + finalPayload.length);
    packet.set(new Uint8Array(header), 0);
    packet.set(finalPayload, header.byteLength);
    this.bytesTransferred += packet.length;
    return packet;
}
```

The `forceSize` parameter allows the client to artificially inflate packet size with padding, serving as a testing feature to analyze the impact of larger packets on network

performance and server-side buffer management.

5.7.4.2 Sending Logic

Data is sent using the `WritableStreamDefaultWriter` associated with each WebTransport stream. The `loggedWrite()` helper function wraps the `writer.write()` operation, providing robust error handling and immediate logging feedback:

```
async loggedWrite(writer, packet, streamType) {
    try {
        await writer.write(packet);
        this.log(` Sent ${streamType} packet: ${packet.length} bytes`, 'success');
        return true;
    } catch (error) {
        this.log(` Failed to send ${streamType} packet: ${error.message}`, 'error')
        return false;
    }
}
```

This function logs every successful packet transmission, including its type and size, aiding in real-time monitoring. In case of failure (e.g., stream closed, network issue), an error is logged, ensuring communication issues are immediately visible. The `writer.releaseLock()` is used after sending test packets to allow re-acquisition for continuous streaming.

5.7.5 Video Streaming Implementation

Video streaming is handled by continuously capturing frames from the local video feed, processing them, and sending them over the dedicated video WebTransport stream. The `startVideoCapture()` method uses a hidden `HTMLCanvasElement` as an intermediary to process video frames:

```
const canvas = document.createElement('canvas');
const ctx = canvas.getContext('2d');
canvas.width = this.isHighQuality ? 1280 : 640;
canvas.height = this.isHighQuality ? 720 : 480;
this.videoWriter = this.videoStream.getWriter();
```

A `captureLoop()` function recursively calls itself using `setTimeout` to maintain a consistent frame rate (15 FPS for standard quality, 30 FPS for high quality). The loop includes:

1. **Frame Capture**: `ctx.drawImage(this.elements.localVideo, 0, 0, canvas.width, canvas.height);` draws the current video frame onto the canvas.
2. **Encoding**: `canvas.toDataURL('image/jpeg', quality);` converts the canvas content into a JPEG image data URL, with `quality` set to 0.7 for standard or 0.9 for high quality.
3. **Binary Conversion**: The Base64-encoded image data is converted into a `Uint8Array` payload.
4. **Packetization and Sending**: The `buildPacket()` method creates a packet with `track_id 'live_video'` and `track_type 'video'`, applying a `forceSize` of 2500 bytes for high-quality streaming to test network handling. The packet is sent via `this.loggedWrite(this.videoWriter, packet, 'VIDEO')`.
5. **Metrics Update**: `this.videoFrameCount++` increments the counter for UI display.

This approach provides granular control over video encoding and packet size, facilitating experimental analysis of quality settings and packet sizes.

5.7.6 Audio Streaming Implementation

Audio streaming leverages the **Web Audio API** for efficient capture and processing of microphone input. The `startAudioCapture()` method sets up an audio processing pipeline:

```
const audioContext = new (window.AudioContext || window.webkitAudioContext)();
await audioContext.resume();
const source = audioContext.createMediaStreamSource(this.mediaStream);
const processor = audioContext.createScriptProcessor(4096, 1, 1);
this.audioWriter = this.audioStream.getWriter();
```

- `AudioContext`: Provides the framework for audio processing, with `audioContext.resume()` ensuring processing starts.
- `MediaStreamSource`: Creates an audio node from the `mediaStream` obtained via `getUserMedia`.
- `ScriptProcessorNode`: Uses a buffer size of 4096 samples with one input and output channel for mono audio.

In the `processor.onaudioprocess` event handler:

1. **Audio Data Extraction**: `event.inputBuffer.getChannelData(0)` retrieves raw floating-point audio samples.

2. **PCM Conversion**: Samples are converted to 16-bit PCM format:

```
const pcmData = new Int16Array(inputData.length);
for (let i = 0; i < inputData.length; i++) {
    pcmData[i] = Math.max(-32768, Math.min(32767, inputData[i] * 32768));
}
const payload = new Uint8Array(pcmData.buffer);
```

3. **Packetization and Sending**: The `pcmData` is used as the payload for a packet with `track_id 'live_audio'` and `track_type 'audio'`, with a `forceSize` of 2200 bytes for testing. The packet is sent via `this.loggedWrite(this.audioWriter, packet, 'AUDIO')`.

4. **Metrics Update**: `this.audioPacketCount++` increments the counter.

The `source` and `processor` nodes are connected to the `audioContext.destination`, ensuring an active audio processing chain.

5.7.7 Chat Messaging Implementation

The live chat functionality enables real-time text communication using a dedicated WebTransport stream. The `sendChatMessage()` method is triggered when the user presses "Enter" or clicks the "Send" button:

1. **Message Retrieval and Validation**: The message is retrieved from `this.elements.chatInp` and trimmed, exiting if empty.

2. **Data Structuring**: The message is encapsulated in a JSON object:

```
const chatData = {
    message_id: Date.now(),
    timestamp: Date.now(),
    user: 'streamer',
    message: message,
    type: 'text'
```

```
};
const payload = this.encoder.encode(JSON.stringify(chatData));
```

3. **Packetization and Sending**: The JSON object is encoded into a `Uint8Array` and passed to `buildPacket()` with `track_id 'live_chat'` and `track_type 'chat'`, then sent via `this.loggedWrite()`.

4. **Local Display and Metrics**: The message is appended to `this.elements.chatMessages`, scrolled to the bottom, the input field is cleared, and `this.chatMessageCount++` is incremented.

This dedicated chat stream ensures low-latency communication, critical for interactive applications.

5.8 Router Construction

This section outlines the design and implementation of a modular and efficient WebTransport Router, built on top of the `aioquic` library. The router employs an event-driven architecture to handle QUIC and HTTP/3 traffic asynchronously. It incorporates a per-stream buffering mechanism for precise packet reconstruction and a configuration-driven approach for flexible routing and runtime adaptability. Additionally, the system includes a microservice proxy with retry logic for reliable delivery and a metrics subsystem for performance monitoring. These design choices create a robust foundation suitable for both research experimentation and real-world deployment.

5.8.1 The aioquic Library

Implementing QUIC and HTTP/3 protocols from scratch is complex due to their requirements for encryption, connection management, congestion control, and multiplexing. To focus on application-layer routing logic, the `aioquic` library was chosen as the foundation.

The `aioquic` library [4] is a mature, open-source Python [26] library that implements QUIC and HTTP/3 protocols entirely in Python, designed for integration with `asyncio`. It provides a high-level API, including `QuicConnectionProtocol` and `H3Connection`, and employs an event-driven programming model, simplifying development.

```
from aioquic.asyncio import serve
from aioquic.asyncio.protocol import QuicConnectionProtocol
from aioquic.quic.configuration import QuicConfiguration
```

```
from aioquic.quic.events import QuicEvent, ProtocolNegotiated, ConnectionTerminated
from aioquic.h3.connection import H3Connection
from aioquic.h3.events import H3Event, HeadersReceived, DataReceived, WebTransportS
```

This library abstracts low-level transport and cryptographic concerns, allowing developers to concentrate on stream parsing and routing logic.

5.8.2 Event-Driven Router

The router is implemented as a subclass of `QuicConnectionProtocol`, leveraging its event-driven capabilities. The central entry point for handling protocol events is the `quic_event_received` method, which dispatches incoming events—such as connection negotiation, termination, or HTTP stream updates—to appropriate handlers.

```
class WebTransportRouter(QuicConnectionProtocol):
    def quic_event_received(self, event: QuicEvent) -> None:
        if isinstance(event, ProtocolNegotiated):
            # Handle protocol negotiation
            ...
        elif isinstance(event, ConnectionTerminated):
            # Handle connection termination
            ...
        if self._http is not None:
            for http_event in self._http.handle_event(event):
                self._handle_http_event(http_event)

    def _handle_http_event(self, event: H3Event) -> None:
        # Handles HTTP/3 and WebTransport events
        ...
```

Modular methods like `_handle_http_event` and `_handle_wt_stream_data` simplify the main event loop, enhancing code readability and maintainability. This scalable structure aligns with best practices for asynchronous network programming.

5.8.3 Packet Buffering and Parsing

Managing data streams over QUIC is challenging due to their continuous byte stream nature, which may include fragmented or concatenated packets. The router maintains a dedicated buffer for each QUIC stream, stored in a dictionary keyed by `stream_id`. Incoming data chunks are appended to the stream-specific buffer, which adheres to a

fixed packet format: a 32-byte header followed by the payload. The header includes a 16-byte track ID, a 4-byte payload length, and a 12-byte track type.

```
self._stream_buffers = {}

def _handle_wt_stream_data(self, stream_id: int, data: bytes, stream_ended: bool, _
    if stream_id not in self._stream_buffers:
        self._stream_buffers[stream_id] = b""
    self._stream_buffers[stream_id] += data
    buffer = self._stream_buffers[stream_id]

    if len(buffer) >= 32:
        header_info = self._parse_packet_header(buffer[:32])
        if header_info:
            expected_payload_length = header_info['payload_length']
            total_packet_length = 32 + expected_payload_length
            if len(buffer) >= total_packet_length:
                payload = buffer[32:total_packet_length]
                asyncio.create_task(self._process_media_packet(header_info, payload
                self._stream_buffers[stream_id] = buffer[total_packet_length:]

                if len(self._stream_buffers[stream_id]) > 0:
                    self._handle_wt_stream_data(stream_id, b"", stream_ended, _recu
```

This approach ensures accurate reassembly of fragmented packets and sequential processing of multiple packets, mitigating potential buffer mismanagement issues through thorough testing.

5.8.4 Configuration-Driven Routing

To avoid hardcoding microservice addresses, a flexible configuration system was designed using a YAML file to define services, global parameters, and routing behavior. A custom `ConfigManager` class parses this file into Python dictionaries and objects. Hot-reloading is implemented by periodically checking the file's modification time, allowing configuration updates without interrupting the router or active client sessions.

```
class ConfigManager:
    def __init__(self, config_path: str):
        self.config_path = config_path
```

```
        self.services = {}
        self.global_config = {}
        self.last_modified = 0
        self.load_config()

    def load_config(self):
        with open(self.config_path, 'r') as f:
            config_data = yaml.safe_load(f)
        self.global_config = config_data.get('global', {})
        services_config = config_data.get('services', {})
        self.services = {name: ServiceConfig(cfg) for name, cfg in services_config.
        self.last_modified = os.path.getmtime(self.config_path)

    def should_reload(self) -> bool:
        return os.path.getmtime(self.config_path) > self.last_modified
```

This design enhances maintainability and system uptime, allowing administrators to modify service endpoints or routing policies dynamically.

5.8.5 Proxying Layer

Once a packet is parsed, it is forwarded to the appropriate microservice, each exposing an HTTP/1.1 endpoint. The router uses `aiohttp` for non-blocking I/O and concurrent forwarding via asynchronous POST requests. The `_forward_to_service` method constructs requests using configuration metadata and implements a retry mechanism with exponential backoff to handle temporary service outages.

```
async def _forward_to_service(self, service_url: str, payload: bytes, retries: int
    backoff = 0.5
    for attempt in range(retries):
        try:
            async with aiohttp.ClientSession() as session:
                async with session.post(service_url, data=payload) as resp:
                    if resp.status == 200:
                        return await resp.read()
        except Exception as e:
            await asyncio.sleep(backoff)
            backoff *= 2
```

This integration layer ensures reliable packet delivery to microservices, even in environments with transient network failures.

5.8.6 Metrics and Logging

For performance evaluation and debugging, the router includes a `MetricsLogger` class that logs metrics such as throughput, stream activity, and memory usage to CSV and structured event logs. The `psutil` library is used to monitor system resources periodically.

```
class MetricsLogger:
    def log_metrics(self, router_metrics: dict, performance_data: dict = None):
        # Writes performance and routing metrics to CSV and logs
        ...
```

These metrics support real-time monitoring and retrospective analysis in research environments. Additionally, the router supports detailed command-line flags for specifying QUIC parameters, certificate paths, log verbosity, and other operational settings, enabling flexible deployment.

This section detailed the implementation of a robust and modular WebTransport Router. Leveraging `aioquic` for protocol handling, the system employs an event-driven architecture for asynchronous stream processing. A stream-specific buffering model ensures accurate packet parsing, while configuration-driven routing supports flexible deployment and runtime reconfiguration. The proxying logic enables reliable microservice communication, and a built-in metrics system facilitates monitoring and analysis. Together, these components form a technically sound router ready for research or production environments.

5.8.7 The aioquic Library

Handling the QUIC and HTTP/3 protocols from scratch is prohibitively complex due to their intricacies in encryption, connection management, congestion control, and multiplexing. To avoid reinventing the protocol stack and remain focused on the application-layer routing logic, the `aioquic` library was selected as the foundation.

`aioquic` is a mature, open-source Python library that implements QUIC and HTTP/3 entirely in Python and is designed for integration with `asyncio`. It exposes a high-level API, including `QuicConnectionProtocol` and `H3Connection`, and uses an event-driven programming model, significantly simplifying development.

```
from aioquic.asyncio import serve
from aioquic.asyncio.protocol import QuicConnectionProtocol
```

```
from aioquic.quic.configuration import QuicConfiguration
from aioquic.quic.events import QuicEvent, ProtocolNegotiated, ConnectionTerminated
from aioquic.h3.connection import H3Connection
from aioquic.h3.events import H3Event, HeadersReceived, DataReceived, WebTransportS
```

This library abstracts the low-level transport and cryptographic concerns, allowing the developer to focus on stream parsing and routing logic.

5.8.8 Event-Driven Router

The router was implemented as a subclass of `QuicConnectionProtocol`, leveraging its event-driven capabilities. The central entry point for handling protocol events is the `quic_event_received` method. Inside this method, incoming events—such as connection negotiation, termination, or HTTP stream updates—are dispatched to appropriate handlers.

```
class WebTransportRouter(QuicConnectionProtocol):
    def quic_event_received(self, event: QuicEvent) -> None:
        if isinstance(event, ProtocolNegotiated):
            # Handle protocol negotiation
            ...
        elif isinstance(event, ConnectionTerminated):
            # Handle connection termination
            ...
        if self._http is not None:
            for http_event in self._http.handle_event(event):
                self._handle_http_event(http_event)

    def _handle_http_event(self, event: H3Event) -> None:
        # Handles HTTP/3 and WebTransport events
        ...
```

Modular methods like `_handle_http_event` and `_handle_wt_stream_data` simplify the main event loop, improving code readability and maintainability. This structure is scalable and aligns with best practices for asynchronous network programming.

5.8.9 Packet Buffering and Parsing

Handling data streams over QUIC presented one of the most technically demanding challenges. Unlike traditional TCP sockets that might send one message per packet, QUIC

streams transmit a continuous byte stream, which may contain fragmented or concatenated packets.

To manage this, the router maintains a dedicated buffer for each QUIC stream, using a dictionary keyed by `stream_id`. Each incoming data chunk is appended to the stream-specific buffer. The router uses a fixed packet format: a 32-byte header followed by the payload. The header consists of a 16-byte track ID, a 4-byte payload length, and a 12-byte track type.

```
self._stream_buffers = {}

def _handle_wt_stream_data(self, stream_id: int, data: bytes, stream_ended: bool, _
    if stream_id not in self._stream_buffers:
        self._stream_buffers[stream_id] = b""
    self._stream_buffers[stream_id] += data
    buffer = self._stream_buffers[stream_id]

    if len(buffer) >= 32:
        header_info = self._parse_packet_header(buffer[:32])
        if header_info:
            expected_payload_length = header_info['payload_length']
            total_packet_length = 32 + expected_payload_length
            if len(buffer) >= total_packet_length:
                payload = buffer[32:total_packet_length]
                asyncio.create_task(self._process_media_packet(header_info, payload
                self._stream_buffers[stream_id] = buffer[total_packet_length:]

                if len(self._stream_buffers[stream_id]) > 0:
                    self._handle_wt_stream_data(stream_id, b"", stream_ended, _recu
```

This approach ensures fragmented packets are correctly reassembled, and multiple packets arriving in one event are processed in order. Buffer mismanagement could easily cause subtle bugs, making thorough testing essential.

5.8.10 Configuration-Driven Routing

Rather than hardcoding microservice addresses into the codebase, a flexible configuration system was designed. A YAML file defines the available services, global parameters, and routing behavior. This file is read by a custom `ConfigManager` class, which parses it into Python dictionaries and objects.

Hot-reloading was implemented by periodically checking the file's modification time. If the timestamp changes, the configuration is reloaded without interrupting the router or active client sessions.

```
class ConfigManager:
    def __init__(self, config_path: str):
        self.config_path = config_path
        self.services = {}
        self.global_config = {}
        self.last_modified = 0
        self.load_config()

    def load_config(self):
        with open(self.config_path, 'r') as f:
            config_data = yaml.safe_load(f)
        self.global_config = config_data.get('global', {})
        services_config = config_data.get('services', {})
        self.services = {name: ServiceConfig(cfg) for name, cfg in services_config.
        self.last_modified = os.path.getmtime(self.config_path)

    def should_reload(self) -> bool:
        return os.path.getmtime(self.config_path) > self.last_modified
```

This design choice significantly improves maintainability and system uptime. Administrators can modify service endpoints or routing policies on the fly without restarting the router.

5.8.11 Proxying Layer

Once a valid packet is extracted and parsed, it must be forwarded to the appropriate microservice. Each service exposes an HTTP/1.1 endpoint. To support non-blocking I/O and concurrent forwarding, the router uses `aiohttp` to send asynchronous POST requests.

The `_forward_to_service` method constructs requests using configuration metadata and includes a retry mechanism with exponential backoff to tolerate temporary service outages.

```
async def _forward_to_service(self, service_url: str, payload: bytes, retries: int
    backoff = 0.5
    for attempt in range(retries):
```

```
        try:
            async with aiohttp.ClientSession() as session:
                async with session.post(service_url, data=payload) as resp:
                    if resp.status == 200:
                        return await resp.read()
        except Exception as e:
            await asyncio.sleep(backoff)
            backoff *= 2
```

This integration layer ensures packets are delivered reliably to microservices, even in environments with transient network failures.

5.8.12 Metrics and Logging

To enable performance evaluation and debugging, the router includes a `MetricsLogger` class. This component logs metrics such as throughput, stream activity, and memory usage to CSV and structured event logs. The `psutil` library is used to monitor system resources periodically.

```
class MetricsLogger:
    def log_metrics(self, router_metrics: dict, performance_data: dict = None):
        # Writes performance and routing metrics to CSV and logs
        ...
```

These metrics are useful for both real-time monitoring and retrospective analysis in research environments.

Additionally, the router supports detailed command-line flags for specifying QUIC parameters, certificate paths, log verbosity, and other operational settings—enabling deployment in diverse scenarios.

This chapter detailed the implementation of a robust and modular WebTransport Router. Starting with the choice of `aioquic` for protocol handling, the system was built using an event-driven architecture that processed streams asynchronously. A stream-specific buffering model enabled safe and accurate packet parsing. Configuration-driven routing allowed flexible deployment and runtime reconfiguration. Finally, the proxying logic enabled reliable communication with microservices, and a built-in metrics system facilitated monitoring and analysis. Altogether, these components formed a router that is both technically sound and ready for use in research or production environments.

5.9 Microservice Layer

This section details the implementation of a generic HTTP-based microservice designed for real-time data ingestion and optional forwarding to Apache Pulsar. The microservice handles various data streams, including audio, video, chat messages, screen captures, and file uploads, relayed from the WebTransport Router. The design prioritizes flexibility, scalability, and operational robustness through a modular architecture that supports dynamic configuration, seamless Pulsar integration, and efficient request handling. Each component is described below, including its purpose, implementation, and design rationale.

5.9.1 Configuration Manager

The configuration manager loads the microservice's runtime settings from a YAML file, enabling operators to modify behavior—such as enabling or disabling Pulsar integration or updating broker URLs—without restarting the service. It supports hot-reloading by detecting file changes, ensuring zero-downtime updates and enhanced operational flexibility.

```
import os
import yaml

class MicroserviceConfigManager:
    def __init__(self, config_path):
        self.config_path = config_path
        self.last_modified = 0
        self.config = {}
        self.load_config()

    def load_config(self):
        with open(self.config_path, 'r') as f:
            self.config = yaml.safe_load(f)
        self.last_modified = os.path.getmtime(self.config_path)

    def reload_if_modified(self):
        current_mod_time = os.path.getmtime(self.config_path)
        if current_mod_time > self.last_modified:
            self.load_config()
```

This class reads the YAML configuration file and stores its contents in memory. It tracks the file's last modified time to detect changes and reloads the configuration when necessary. This design allows critical parameters, such as Pulsar broker URLs or toggle flags, to be updated live, minimizing downtime and avoiding service restarts.

5.9.2 Apache Pulsar Client Initialization

This component manages the connection to the Apache Pulsar broker and creates a message producer for the microservice's specific topic. It supports reinitialization if configuration changes occur, ensuring the service connects to the correct broker with updated settings.

```
import pulsar

pulsar_client = None
producer = None

def initialize_pulsar(broker_url, service_name, send_to_pulsar):
    global pulsar_client, producer
    if pulsar_client:
        pulsar_client.close()

    if not send_to_pulsar:
        pulsar_client = None
        producer = None
        return

    pulsar_client = pulsar.Client(broker_url)
    topic = f"persistent://public/default/{service_name}-topic"
    producer = pulsar_client.create_producer(topic, batching_enabled=False)
```

The function closes any existing Pulsar client to prevent resource leaks and checks if Pulsar forwarding is enabled. If disabled, it clears connections. Otherwise, it establishes a connection to the specified broker and creates a producer for a topic named after the microservice role. Disabling batching ensures immediate message delivery, critical for real-time data streaming.

5.9.3 HTTP Data Handler

The HTTP data handler processes incoming HTTP POST requests containing data streams from the WebTransport Router. It extracts the payload and metadata, optionally forwards the data to Pulsar, and responds with a confirmation. The design supports multiple media types with unified logic, enhancing code reuse and simplicity.

```
from aiohttp import web

class GenericServiceHandler:
    def __init__(self, service_name):
        self.service_name = service_name
        self.packet_count = 0

    async def process_request(self, request):
        self.packet_count += 1
        payload = await request.read()
        track_id = request.headers.get('X-Track-ID', 'unknown')

        if producer:
            producer.send(payload, properties={"track_id": track_id})

        return web.json_response({
            "status": "success",
            "packet_id": self.packet_count,
            "track_id": track_id,
            "payload_size": len(payload)
        })
```

For each request, the handler reads the raw data stream and metadata, such as `track_id`, increments a packet counter, and, if the Pulsar producer is active, sends the data with metadata properties for downstream identification. It responds with a JSON object confirming successful processing. This abstraction ensures simplicity and flexibility across different data types.

5.9.4 Service Startup and Configuration Watcher

This component launches an asynchronous HTTP server on a configurable port and exposes a dynamic route based on the service name (e.g., `/process_video`). It continuously

monitors the configuration file and reinitializes Pulsar connections if settings change, enabling dynamic adaptability without service restarts.

```
import asyncio
from aiohttp import web

async def start_service(service_name, port, config_path):
    config_manager = MicroserviceConfigManager(config_path)
    send_to_pulsar = config_manager.config.get('send_to_pulsar', False)
    broker_url = config_manager.config.get('pulsar_broker_url', '')

    initialize_pulsar(broker_url, service_name, send_to_pulsar)
    handler = GenericServiceHandler(service_name)

    app = web.Application()
    app.router.add_post(f'/process_{service_name}', handler.process_request)
    runner = web.AppRunner(app)
    await runner.setup()
    site = web.TCPSite(runner, '0.0.0.0', port)
    await site.start()

    while True:
        await asyncio.sleep(10)
        config_manager.reload_if_modified()
        new_send = config_manager.config.get('send_to_pulsar', False)
        new_url = config_manager.config.get('pulsar_broker_url', '')
        if new_send != send_to_pulsar or new_url != broker_url:
            send_to_pulsar = new_send
            broker_url = new_url
            initialize_pulsar(broker_url, service_name, send_to_pulsar)
```

This function starts the `aiohttp` server with a route specific to the service role, loads initial configuration settings, and establishes the Pulsar connection. It periodically checks for configuration changes every 10 seconds, updating Pulsar connection parameters dynamically if needed, preventing service downtime.

5.9.5 Microservice Complete Flow

At startup, the microservice loads its configuration from a YAML file, defining settings such as the service name, port, Pulsar broker URL, and whether data should be forwarded to Apache Pulsar. Based on these settings, it establishes a Pulsar connection if enabled, allowing real-time data publication to a designated topic. The service exposes an HTTP endpoint named according to its role (e.g., `/process_video`, `/process_audio`, or `/process_chat`). Upon receiving data through this endpoint, the HTTP handler processes the raw payload and associated metadata, such as a tracking ID, and optionally forwards it to Pulsar for downstream processing. Concurrently, the service monitors the configuration file for changes. If modifications are detected—such as updates to the broker URL or toggling data forwarding—it reloads the configuration and reinitializes the Pulsar connection without restarting. This design yields a reusable codebase capable of handling multiple streaming data types, ensuring flexibility, scalability, and operational robustness across deployments.

5.10 Kubernetes Deployment

This implementation is critical for the completion of the proposed WebTransport solution. The router code is designed to load balance streams of live data to separate microservices. The router, microservice code, and Pulsar components are deployed into a Kubernetes environment within a Minikube cluster. This section details the steps and configurations required for a successful deployment.

5.10.1 Deployment Secrets

To enable the router to listen for WebTransport streams, Kubernetes must securely store the necessary certificates. Certificates are stored as secrets within the cluster, allowing resources in the default namespace to access them when mounted on a specified path. The following command was used to create a TLS secret:

```
kubectl create secret tls quic-cert --cert=path/.crt --key=path/.key
```

This command creates a secret named `quic-cert` in the default namespace, making the certificate and key accessible to all resources within the namespace.

5.10.2 Router Deployment

The router deployment involves containerization, creating ConfigMaps, defining a deployment, and exposing a service. The router's configuration file, as described in the router

implementation, supports hot-reloading for dynamic updates.

The router code was containerized using a Dockerfile placed in the same directory as the router code, with the `router_config.yaml` file kept external to the image. The following commands built and pushed the Docker image to a DockerHub repository:

```
docker build -t sbnm007/quic_router:0.0.0 .
docker push sbnm007/quic_router:0.0.0
```

This process successfully created a container image for the router, ready for deployment to the Kubernetes cluster.

The `router_config.yaml` file, which defines routing rules for microservices, was deployed as a Kubernetes `ConfigMap`. This practice enhances extensibility by allowing dynamic addition or removal of microservices through endpoint configuration, avoiding hardcoded values and the need to rebuild the image. The `router.yaml` file encapsulates the complete Kubernetes configuration for the WebTransport Router, enabling dynamic routing of real-time data (e.g., video, audio, and chat) to respective microservices. It includes:

- A `ConfigMap` containing `router_config.yaml`, supporting hot-reloading of routing rules such as endpoint, port, data format, and custom headers.
- A `Deployment` specifying the Docker image, QUIC/HTTP3 ports, and volume mounts for the configuration file and TLS certificates, configured for a single replica with scalability options.
- A `Secret` for securely injecting TLS certificates to enable encrypted QUIC communication.
- Environment variables to locate mounted configuration and certificate files.
- A `Service` of type `LoadBalancer` exposing the router externally on port 443 for TCP or UDP client connections.

The resources were deployed using:

```
kubectl apply -f router.yaml
```

To verify the deployment, the following commands were used to check resources:

```
kubectl get pods
kubectl get svc
```

The `Service` receives an external IP address from MetalLB, enabling external access to the router.

5.10.3 Microservice Deployment

The `microservice.yaml` file defines the Kubernetes configuration for deploying individual microservices—**video**, **audio**, and **chat**—in a scalable and configurable manner. A shared `ConfigMap` named `microservice-app-config` contains the `microservice_config.yaml` file, mounted into each microservice container at `/config` and accessible via an environment variable. This enables hot-reloading of settings like the Pulsar broker URL and toggles for Pulsar forwarding.

Each microservice is deployed via a dedicated `Deployment` resource, running a Python-based service with arguments specifying its type and listening port. A corresponding `ClusterIP Service` exposes each microservice internally on ports **4434** for video, **4435** for audio, and **4436** for chat, facilitating reliable communication via internal DNS.

The modular architecture allows new microservices to be added by replicating and modifying deployment and service blocks. Key configuration parameters for Pulsar integration include:

```
pulsar_broker_url: "pulsar://10.100.48.76:6650"
send_to_pulsar: true
```

These values, included in the `ConfigMap`, support debugging and operational control over Pulsar connectivity.

5.10.4 Pulsar Deployment

Deploying Apache Pulsar in a constrained Minikube environment presented challenges, particularly with the official Helm chart, which caused out-of-memory (OOM) issues in init containers and failures in BookKeeper nodes. To address this, a lightweight Helm chart from DataStax was used, tailored for development environments. The deployment process involved the following steps:

- Add the DataStax Helm chart repository:

  ```
  helm repo add datastax-pulsar https://datastax.github.io/pulsar-helm-chart
  ```

- Update the Helm repository:

  ```
  helm repo update
  ```

- Download the development configuration:

```
curl -LOs https://datastax.github.io/pulsar-helm-chart/examples/dev-values.yaml
```

- Modify `dev-values.yaml` to disable monitoring for reduced resource consumption:

```
kube-prometheus-stack:
  enabled: false
```

- Deploy Pulsar with the modified configuration:

```
helm install pulsar -f dev-values.yaml datastax-pulsar/pulsar
```

This deployment created two key services: `pulsar-broker` for internal message brokering and `pulsar-proxy` for external access. The `pulsar-broker` enables in-cluster communication for microservices, while the `pulsar-proxy` is exposed externally via a MetalLB load balancer.

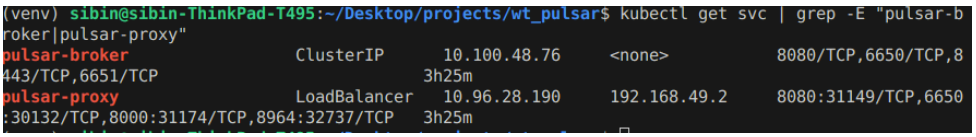


Figure 5.4: Pulsar Kubernetes Services

A sample Python consumer was developed to demonstrate real-time message consumption, subscribing to a specified topic with a shared subscription type and reporting end-to-end latency:

```
consumer = client.subscribe(TOPIC_NAME, SUBSCRIPTION_NAME,
                            consumer_type=pulsar.ConsumerType.Shared)
msg = consumer.receive(timeout_millis=5000)
```

This consumer supports real-time consumption of video, audio, or chat streams and is extensible to other topics, validating integration with the Pulsar cluster.

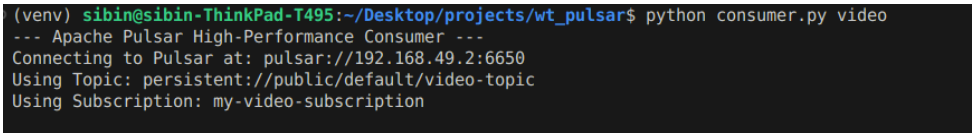


Figure 5.5: Consumer Connection to Topic

```
Received 9612 bytes - E2E Latency: 0.00ms
Received 9653 bytes - E2E Latency: 0.00ms
Received 9611 bytes - E2E Latency: 0.00ms
Received 9599 bytes - E2E Latency: 0.00ms
Received 9549 bytes - E2E Latency: 0.00ms
Received 9624 bytes - E2E Latency: 0.00ms
Received 9595 bytes - E2E Latency: 0.00ms
Received 9595 bytes - E2E Latency: 0.00ms
Received 9573 bytes - E2E Latency: 0.00ms
Received 9624 bytes - E2E Latency: 0.00ms
Received 9636 bytes - E2E Latency: 0.00ms
Received 9632 bytes - E2E Latency: 0.00ms
```

Figure 5.6: Consumer Receiving Data from Video Topic

5.10.5 Retrieving Results

Metrics are logged in a CSV file within the router pod. To retrieve these files locally for evaluation, the following commands were executed:

```
mkdir metrics-in-pod
kubectl cp webtransport-router-deployment-65cf75d584-njmqf:/app/metrics_logs ./metr
```

This process transfers the CSV files to the local machine, enabling detailed analysis of the system's performance metrics.

5.11 Summary

In this chapter, a complete end-to-end prototype of the proposed system is implemented and validated within a local Kubernetes environment. The workflow begins with a browser-based WebTransport client that streams multiple types of media—such as video, audio, chat, and files—in real time using unidirectional QUIC streams. Each packet is formatted with a custom header and transmitted securely using TLS, with certificate handling and DNS mapping configured for seamless local development. The system demonstrates robust session management, live metrics collection, and a responsive user interface, enabling detailed insight into the streaming behavior.

The WebTransport router, deployed as a scalable microservice, acts as the central intelligence for traffic distribution. It uses asynchronous QUIC and HTTP/3 handling to buffer, parse, and dynamically route packets to the appropriate microservices based on a hot-reloadable YAML configuration. These backend microservices, implemented generically to support various media types, process the data and optionally publish it to Apache Pulsar topics for further stream processing or analytics. The deployment showcases critical infrastructure features such as TLS termination, local DNS resolution, zero-downtime scaling, and runtime observability—all orchestrated through Minikube, MetalLB, and Kubernetes-native tools. This working prototype not only verifies the

architectural design but also lays a strong, extensible foundation for production-scale deployments or future research-driven enhancements.

Chapter 6

Evaluation

This chapter presents a comprehensive evaluation of the WebTransport-aware routing system deployed on Minikube [6]. The evaluation focuses on performance metrics including latency, throughput, and resource utilization under different client loads and network conditions.

The WebTransport stream routing system's evaluation in kubernetes has been thoroughly evaluated in this chapter. The analysis is aimed at assessing the performance, scalability, custom router features and reliability by evaluating under different load conditions. Monitoring solutions like Prometheus and Grafana are not used here because of this system being an HTTP/3 server and the custom monitoring not supporting it. This system follows the use of a different approach because of limited tooling support for Webtransport protocol. Our HTTP/3 WebTransport server follows a procedure that allows the capture of the detailed performance metrics in Comma-Separated Values (CSV) format.

The evaluation method includes both measuring qualitative as well as quantitative performance. This helps us understand what the system can do well and where it might have limitations when used in real-world environments.

6.1 Experimental Design

6.1.1 Evaluation Methodology

The evaluation methodology employs a multi-dimensional metrics collection system designed to capture comprehensive performance characteristics across four critical parts. The Buffer Management Metrics provide insights into memory utilization patterns through 'buffer_size_bytes', which tracks real-time buffer consumption whereas 'max_buffer_size_bytes' records peak memory usage during sessions. Similarly, 'buffer_overflows' counts critical

overflow events that could indicate system stress or inadequate resource allocation.

Packet Processing Metrics forms the core for performance analysis, it has 'packets_total' which is for aggregate packet volume, 'packets_complete' represents successfully processed packets without fragmentation, 'packets_fragmented' indicates packets requiring segmentation due to size constraints, and 'fragmentation_rate' expressing the percentage of packets experiencing fragmentation which acts as a critical indicator of network efficiency and potential performance bottlenecks.

Stream Management Metrics monitors the multiplexing capabilities of to WebTransport streams by measuring 'streams_active', which tracks currently operational streams, and 'streams_max_concurrent', which records the peak simultaneous stream count which is useful for testing the upper limit. Finally, Performance Metrics quantifies system responsiveness and efficiencies via 'throughput_mbps' metric which measures data transmission rates in megabits per second, 'avg_processing_time_ms', captures the mean latency for packet processing operations, and 'avg_buffer_wait_ms', indicates queuing delays within the buffer management system.

These comprehensive metrics enables granular analysis of system behavior under varying load conditions, allowing us to identify bottlenecks, establish faster debugging, analyze resource utilization patterns, and optimization opportunities essential for validating the WebTransport router's operational effectiveness for production grade Kubernetes environments.

6.1.2 Experimental Configuration

The experimental setup used the following configuration parameters to ensure reproducible results across all test scenarios and to understand the router specific performance.

The '–max-data' parameter (1GB) sets the total unacknowledged data limit for the entire QUIC connection. This value is chosen to support high-throughput streaming without risking excessive memory use for pod in kubernetes. The '–max-stream-data' value (256KB) caps unacknowledged data per stream to ensure a balance between throughput and fairness across multiple concurrent streams. The '–max-datagram-size' (3000 bytes) is selected to minimize packet fragmentation while remaining within typical MTU limits. These values closely follow or slightly adjust the default settings provided by 'aioquic' to better fit the memory and latency characteristics of the deployment environment. The buffer overflow threshold prevents excessive memory usage, since we are running the pod in kubernetes, the pod memory limits could crash the system hence a 10MB limit. It also has a mechanism which detects large buffers at 5MD threshold so we can receive alerts with logs before the buffer overflows. This helps to monitor when buffer grows unusually

Table 6.1: Router Configuration Parameters

Parameter	**Value**
`max-data`	1048576 (1MB)
`max-stream-data`	262144 (256KB)
`max-datagram-size`	3000 bytes
Buffer overflow threshold	10MB
Large buffer detection	5MB
High fragmentation threshold	0.5 (50%)
Sample retention limit	1000 samples
Default timeout	30 seconds
Default retries	3
Default connect timeout	5 seconds

large. A threshold is set of 0.5 for fragmentation to identify the extent of fragmentation. Its calculated by dividing the observed fragmented packets by total packets.The timeout parameters ensure robust connection handling by allowing 30 seconds for requests and 3 retries before failure, preventing resource exhaustion from stalled connections. A 5-second connect timeout balances responsiveness with reliability, tuned for typical Kubernetes network conditions under variable load.

Table 6.2: Video Quality Configuration

Parameter	**Value**
FPS	15 fps
Resolution	640x480
Compression	0.7

Table 6.3: Audio Configuration

Parameter	**Value**
Buffer Size	4096 samples
Channels	1 (mono)
PCM Range	-32768 to 32767

The WebTransport client is constructed with the conservative performance options that allows for consistent results and guarantee a very wide compatibility and that streaming performance across diverse network conditions and devices. The video streaming is

Table 6.4: Packet and Track Configuration

Parameter	**Value**
Packet Structure	
Header Size	32 bytes
Track ID Size	16 bytes
Payload Length Size	4 bytes
Track Type Size	12 bytes
Track IDs	
Live Video	'live_video'
Live Audio	'live_audio'
Live Chat	'live_chat'
Track Types	
Video Type	'video'
Audio Type	'audio'
Chat Type	'chat'

set to 15 fps in 640x480 resolution with 0.7 JPEG compression which is a reasonable compromise between visual quality and bandwidth consumptions.Audio processing is buffered internally with a 4096-sample (in mono) size at 16-bit PCM encoding to provide about 93ms of audio latency at common 44.1 kHz sampling frequencies which enough to provide good real-time performance without excessive processing overhead. The system uses three separate and unidirectional streams (live_video, live_audio, live_chat) to allow effective multiplexing and flow control on a per-media-type basis.

The client uses a custom format of 32 bytes of fixed-size fields as packet header which helps WebTransport router efficiently parse the packet: 16 bytes of stream identification, 4 bytes of payload length, and 12 bytes for classifying a track type. This well-ordered design enables the router to forward packets into proper microservices. The track type system ('video', 'audio', 'chat') provides clear semantic routing information, while the fixed header size aligns with memory boundaries for good processing performance. These default parameters create a evaluation friendly system that generates measurable buffer dynamics and fragmentation patterns essential for this dissertation

6.2 Results and Analysis

In order to observe reliable results the the experiments ie single client and multi Client tests were run for a total duration of 10 minutes in order to observe the performance over time with the configurations defined in section.

6.2.1 Single Client Performance Analysis

The single client experiment provides baseline performance characteristics of the Web-Transport router under minimal load conditions.

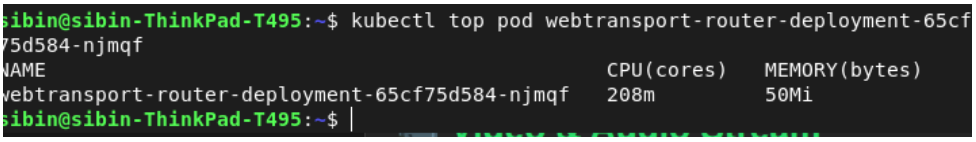


Figure 6.1: Single Client Router Performance Overview

Figure 6.1 demonstrates the overall router pod performance during single client operation, showing stable resource utilization and consistent processing patterns.

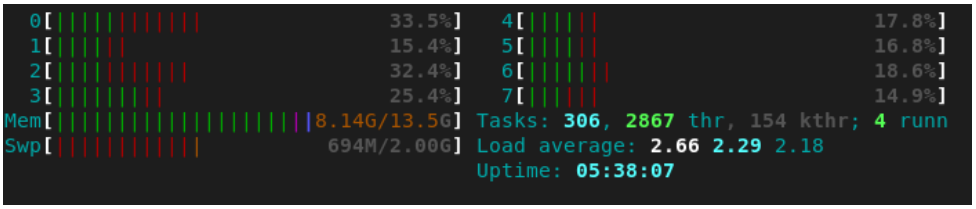


Figure 6.2: Single Client Host Machine Performance Overview

Figure 6.2 demonstrates the impact on the overall host machine performance during single client operation.

6.2.1.1 Packet Processing Performance

Figure 6.3 presents the packet processing performance from 18:05 to 18:15, during which the system handled a total of 157,285 packets. Throughout this 10-minute window, packet processing exhibited a consistent linear growth rate, indicating stable system performance without any observable bottlenecks. Notably, the fragmentation rate stabilized at approximately 90.4%, corresponding to 142,233 fragmented packets. This high fragmentation ratio suggests that the majority of incoming packets exceeded the configured maximum datagram size, likely due to the nature of the input streams. In contrast, only 15,052 complete packets were received, forming a small but steady subset of the total traffic. All fragmented packets were successfully reassembled, achieving a 100% reconstruction rate.

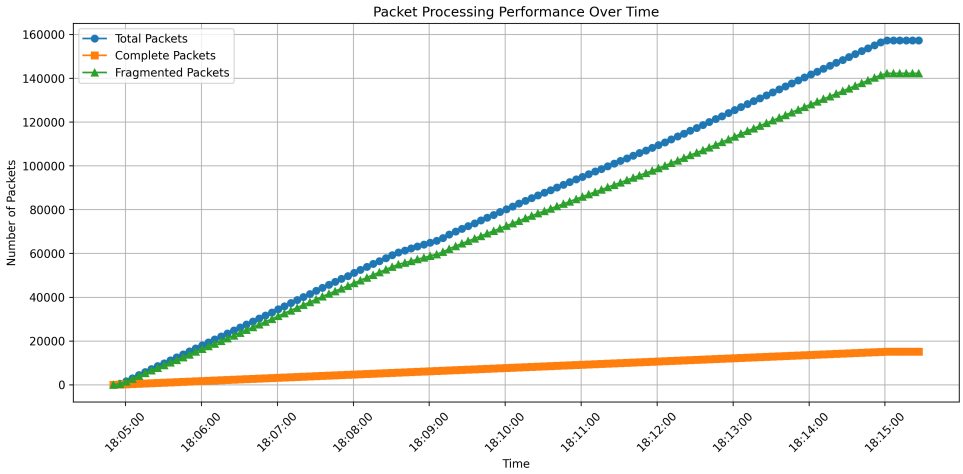


Figure 6.3: Single Client Packet Volume vs Time

Furthermore, system throughput peaked at around 0.3385 Mbps and remained stable, while processing and buffer wait times consistently averaged between 0.03 and 0.04 milliseconds, demonstrating the system's efficiency in managing high-throughput, fragmented traffic.

6.2.1.2 Fragmentation Analysis

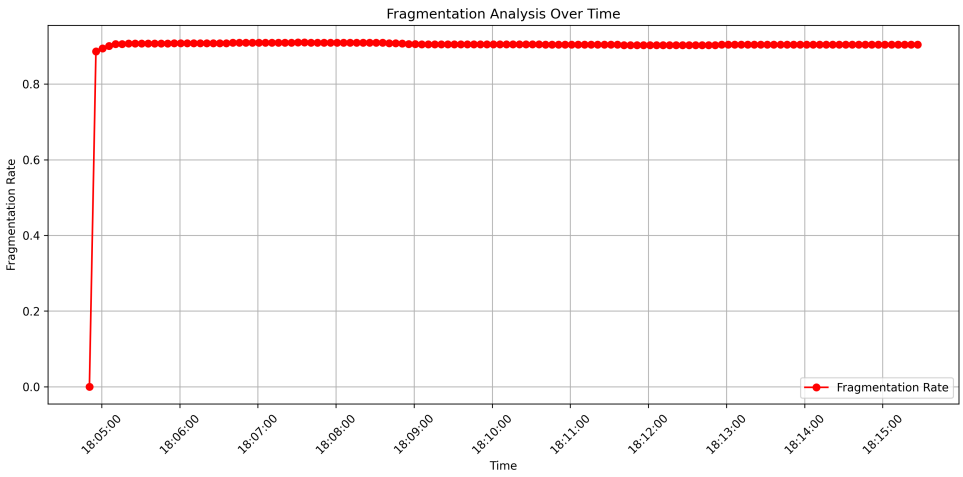


Figure 6.4: Single Client Fragmentation Rate vs Time

Figure 6.4 shows the fragmentation behavior throughout the session from 18:05 to 18:15. The fragmentation rate stabilizes at approximately 0.9 (90%), significantly exceeding the configured threshold of 0.5 (50%), triggering high fragmentation warnings. Despite this, the system maintained stable performance without degradation in throughput or latency. The high fragmentation rate is primarily due to the large size of the

input data. Media packets, including video streams with substantial frame data and audio packets with buffers of 4096 samples, often exceed the 3000-byte datagram size limit, necessitating fragmentation. These oversized payloads are split across multiple packets, which is expected given the nature of the data streams.

6.2.1.3 Throughput Performance

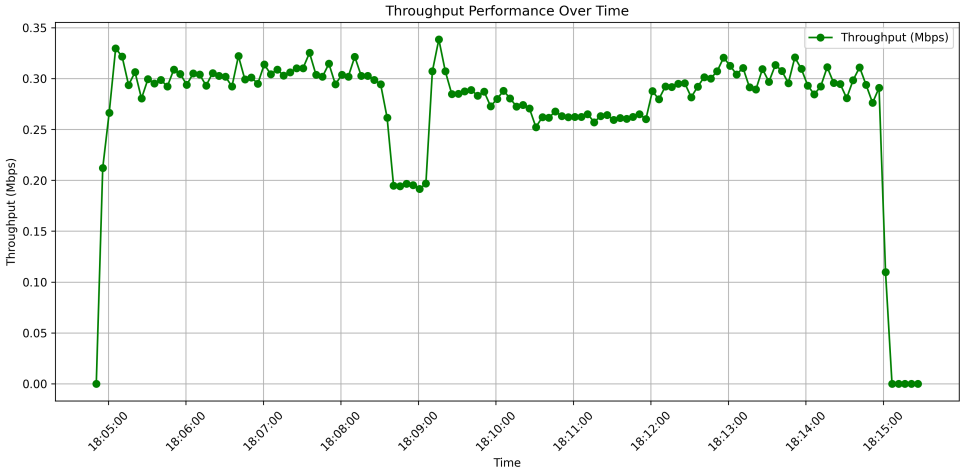


Figure 6.5: Single Client Throughput vs Time

Figure 6.5 displays the throughput performance over time from 18:05 to 18:15. The system maintained a throughput ranging between 0.25 and 0.35 Mbps, with a peak performance of approximately 0.35 Mbps around 18:06-18:07. The throughput declined sharply to 0.0 Mbps at 18:15, corresponding to the session termination. The throughput analysis indicates an average throughput of about 0.28 Mbps, with variations showing a ±25% fluctuation from the average. The initial rise to 0.35 Mbps suggests high data handling capacity early in the session, while the dip around 18:08-18:09 reflects a temporary reduction, possibly due to processing adjustments. The smooth decline to 0.0 Mbps at 18:15 indicates a graceful connection termination without abrupt interruptions.

6.2.1.4 Latency and Processing Time

The system exhibited excellent latency performance, with an average processing time of just 0.03-0.04 milliseconds, remaining consistent throughout the session. Similarly, the average buffer wait time was also 0.03-0.04 milliseconds, indicating minimal queuing and efficient data handling. Notably, latency remained stable and showed no observable correlation with fluctuations in throughput or packet fragmentation.

These low latency values confirm the system's suitability for real-time applications requiring minimal processing delays.

6.2.1.5 Buffer Management

Buffer management for (`client_134025892847376`), as recorded in the (`buffer_metrics_20250805_1`) log between 18:05 and 18:15 on August 5, 2025, demonstrates effective handling of dynamic data loads. The buffer size (`buffer_size_bytes`) varied throughout the session, occasionally dropping to 0 bytes and peaking at 13,398 bytes. Initially, the maximum buffer capacity (`max_buffer_size_bytes`) remained constant at 21,475 bytes until 18:08:35, after which it increased to 24,281 bytes, indicating an adaptive strategy to accommodate increased data traffic. Notably, no buffer overflows (`buffer_overflows`) were recorded, reflecting the system's robust buffer allocation and the ability to absorb traffic fluctuations without packet loss.

6.2.2 Three-Client Performance Analysis

The three-client experiment characterizes the WebTransport router under a moderate, multi-tenant load. Figure 6.6 illustrates the packet processing performance across three

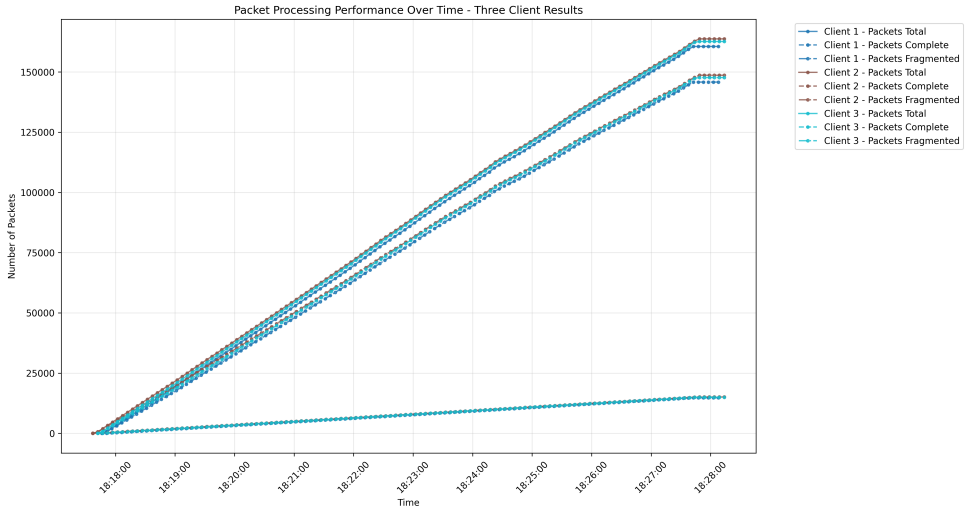


Figure 6.6: Packet Processing Performance Over Time – Three Client Results

clients from 18:05 to 18:15, with the router handling approximately 487,000 packets in total. Each client processed: Client 1 with 160,586 packets, Client 2 with 162,643 packets, and Client 3 with 163,724 packets. The per-second arrival rate was linear and nearly identical across clients, indicating fair scheduling. Fragmentation averaged 90.8%, resulting in about 442,000 fragmented packets and 45,000 complete packets, with a 100% successful reassembly rate and no logged losses.

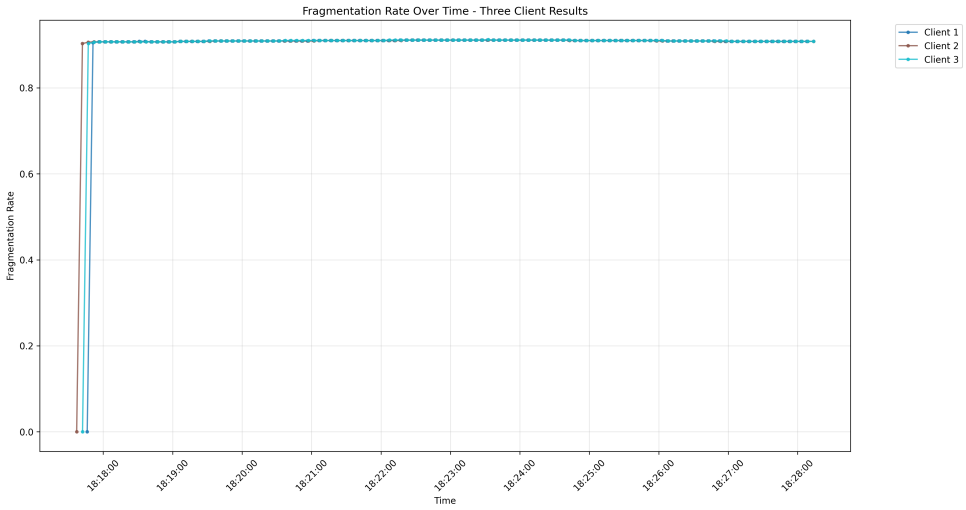


Figure 6.7: Fragmentation Rate Over Time – Three Client Results

6.2.2.1 Three Client Fragmentation Analysis

Figure 6.7 shows the fragmentation behavior across the three clients, stabilizing at 0.9–0.91 (90–91%), consistently exceeding the 0.5 (50%) threshold and triggering high fragmentation warnings. The high rate is attributed to 4,096-sample audio buffers and large video frames surpassing the 3,000-byte datagram limit. Despite uniform 90% fragmentation, throughput and latency remained stable, suggesting linear scalability in the fragmentation/reassembly path.

6.2.2.2 Three Client Throughput Analysis

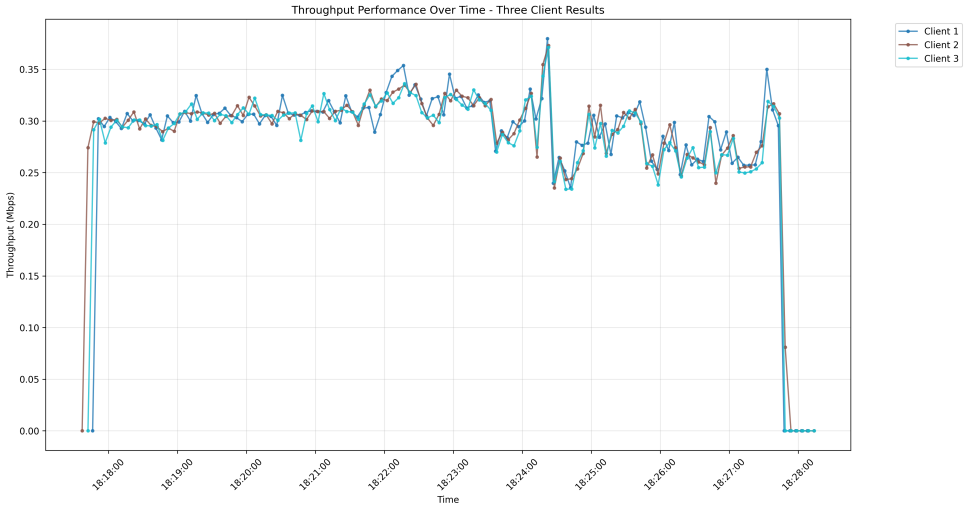


Figure 6.8: Throughput Over Time – Three Client Results

Figure 6.8 displays the throughput performance per client, with an average of 0.30

Mbps, an aggregate average of 0.90 Mbps, a peak of 0.38 Mbps, and a minimum of 0.23 Mbps. Throughput curves were smooth and uncorrelated, ruling out head-of-line blocking, with a brief dip to 0 Mbps at the end marking graceful connection teardown.

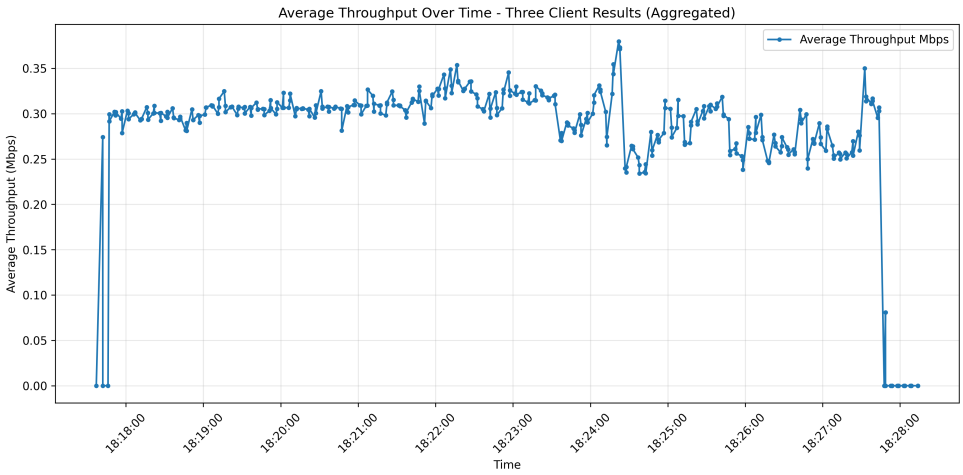


Figure 6.9: Average Aggregated Throughput

Figure 6.9 shows the combined throughput maintaining a steady 0.85–0.90 Mbps plateau, with no multi-second zero-throughput periods until deliberate shutdown.

6.2.2.3 Router Pod Overview

The router pod utilized 418 m (0.42 core) of CPU and 92 MiB of memory, with no restarts and stable performance throughout the 11-minute window, indicating modest control-plane and data-plane overhead for handling three clients.

```
sibin@sibin-ThinkPad-T495:~$ kubectl top pod webtransport-router-deployment-65cf
75d584-njmqf
NAME                                          CPU(cores)   MEMORY(bytes)
webtransport-router-deployment-65cf75d584-njmqf   418m         92Mi
```

Figure 6.10: Router Pod Performance for 3 clients

6.2.2.4 Host Machine Overview

The host machine showed CPU usage between 29.9% and 40.6% across cores (averaging around 35%), memory usage at 8.53 GiB / 13.5 GiB (63%), and a load of 2.85 / 2.16 / 2.04 with 308 tasks (2 running). Utilization was healthy with no swap thrashing, confirming ample headroom.

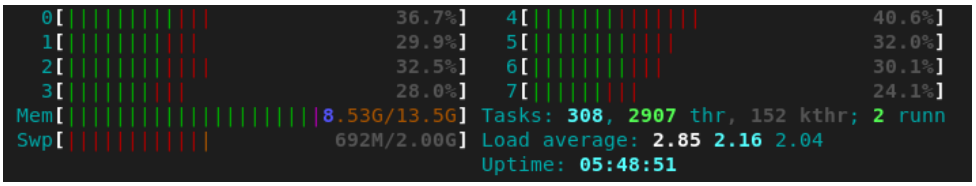


Figure 6.11: Host Machine stats for 3 clients

6.2.2.5 Latency and Processing Time

Latency remained sub-millisecond, with an average processing time of 0.02–0.04 ms and an average buffer wait time of 0.02–0.04 ms across all clients, indicating no increase in per-packet overhead with additional concurrent clients.

6.2.2.6 Buffer Management

Each client maintained an independent buffer ring, with the maximum buffer size auto-tuning from 16–17 kB to 24 kB as traffic increased. Peak usage reached 18,270 bytes (Client 1), well below the 24 kB limit, and no buffer overflows were recorded, demonstrating effective back-pressure and kernel socket buffer management

6.2.3 Five-Client Performance Analysis

The five-client experiment extends the scaling study to 15 concurrent media streams (5 clients × 3 streams each) and represents the first high-load scenario for the WebTransport router. Figure 6.12 illustrates the packet processing performance across five clients from

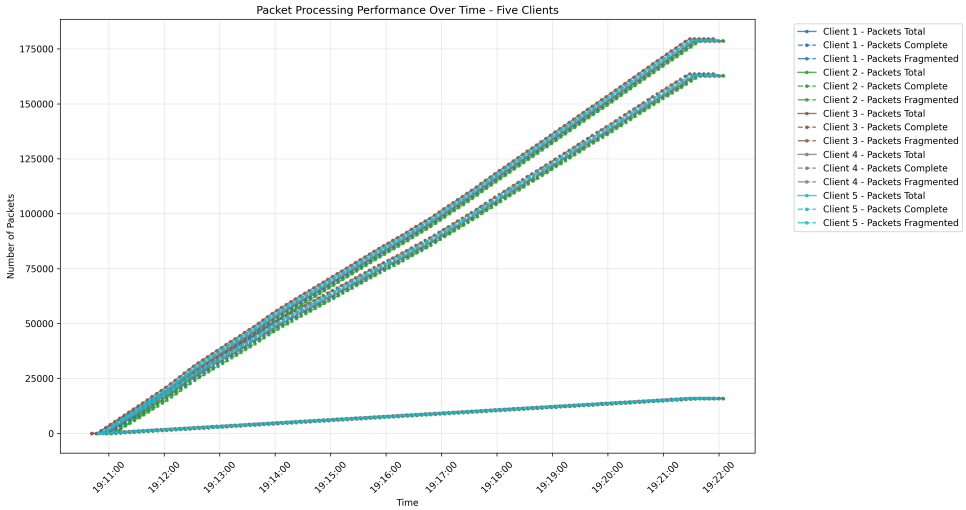


Figure 6.12: Packet Processing – Five Clients

19:10 to 19:22 on August 5, 2025, with an aggregate of approximately 890,000 packets.

Per-client packet counts ranged from 175,000 to 180,000, showing linear growth and fair scheduling without head-of-line blocking. The fragmentation rate varied between 90.8% and 91.6%, with 100% reassembly success and no packet losses. Figure 6.13 displays the

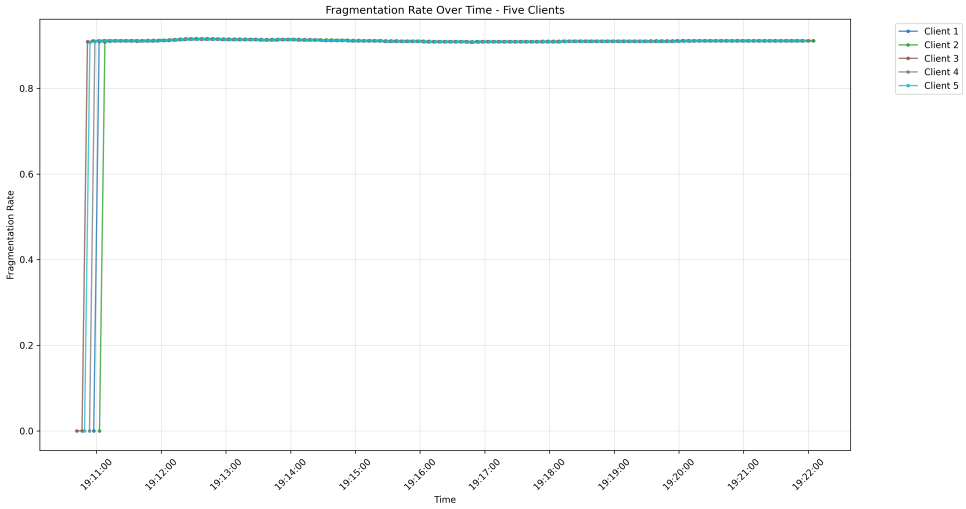


Figure 6.13: Fragmentation Rate – Five Clients

fragmentation behavior across the five clients, stabilizing at approximately 0.9 (90%), consistently triggering HIGH_FRAGMENTATION warnings. The high rate is due to 4,096-sample audio buffers and large video frames exceeding the 3,000-byte datagram limit, yet throughput and latency remained unaffected, highlighting efficient reassembly. Figure 6.14 shows the throughput performance per client, with an average of 0.315 Mbps,

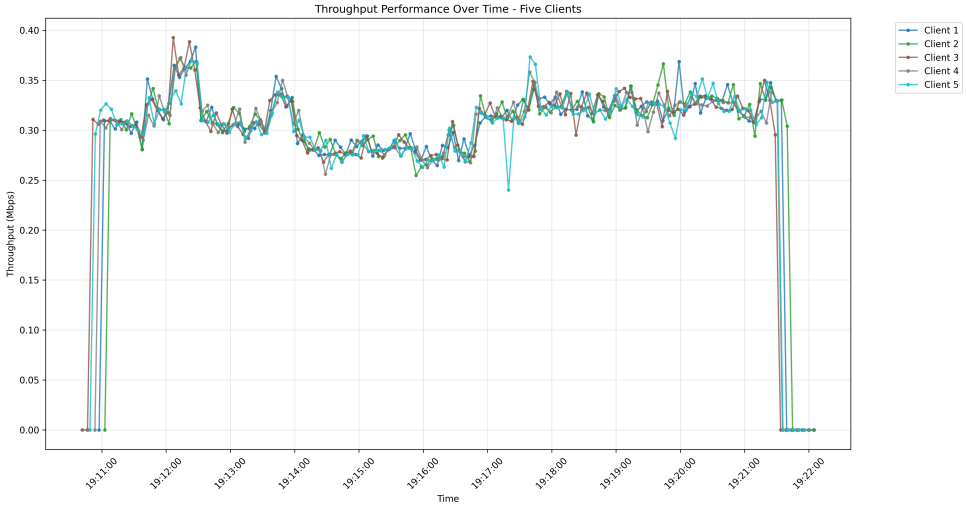


Figure 6.14: Throughput – Five Clients

an aggregate average of 1.58 Mbps, a peak of 0.395 Mbps, and a lowest trough of 0.23 Mbps. The aggregate trace was smooth, with no multi-second drops until deliberate

shutdown. Figure 6.15 presents the combined throughput oscillating between 1.5–1.6

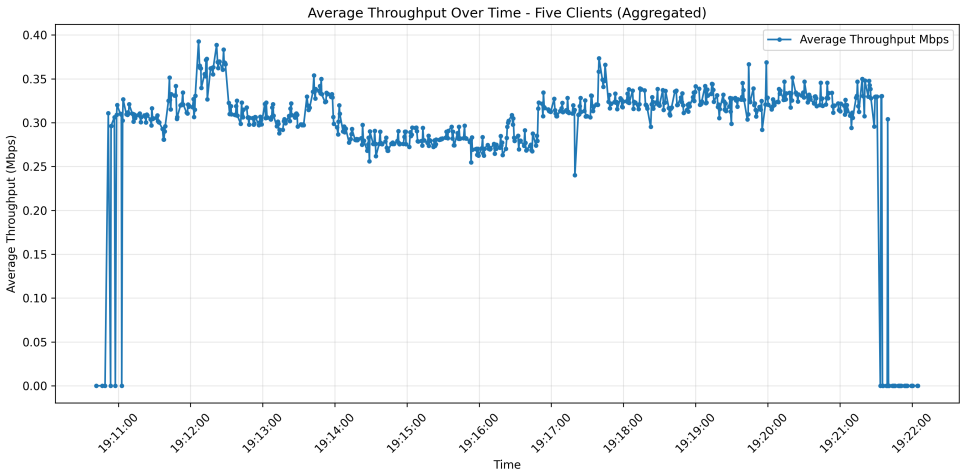


Figure 6.15: Aggregated Throughput – Five Clients

Mbps for most of the run, validating near-linear scaling from the three-client baseline.

6.2.3.1 Router Pod Overview

```
sibin@sibin-ThinkPad-T495:~$ kubectl top pod webtransport-router-deployment-65cf
75d584-njmqf
NAME                                              CPU(cores)   MEMORY(bytes)
webtransport-router-deployment-65cf75d584-njmqf   632m         169Mi
sibin@sibin-ThinkPad-T495:~$
```

Figure 6.16: Router Pod Performance – Five Clients

Figure 6.16 shows the router pod performance at 10:54 PM BST on August 5, 2025, utilizing approximately 0.42 core (418 m) of CPU and 92 MiB of memory, with no restarts and stable uptime for 12 minutes. CPU utilisation remained unchanged compared to the three-client run, indicating sub-linear control-plane overhead scaling with client count.

6.2.3.2 Host Machine Overview

```
  0[||||||||||||||||          47.4%]   4[||||||||||||||||||        48.8%]
  1[||||||||||||||            40.2%]   5[||||||||||||||           38.2%]
  2[||||||||||||||||||        49.7%]   6[||||||||||||||||        41.8%]
  3[||||||||||||||||          41.4%]   7[||||||||||||||||||      46.9%]
Mem[|||||||||||||||||||||9.08G/13.5G] Tasks: 309, 2946 thr, 151 kthr; 5 runn
Swp[|||||||||||||||          782M/2.00G] Load average: 4.80 3.63 2.17
                                       Uptime: 06:46:35
```

Figure 6.17: Host Machine Performance – Five Clients

Figure 6.17 depicts the host machine performance at 10:54 PM BST on August 5, 2025, peaking at 40.6% CPU usage across cores, using 8.53 GiB / 13.5 GiB (63%) memory,

maintaining a load of 2.85 / 2.16 / 2.04, and recording 0 MiB swap usage, comfortably handling the additional 10 streams.

6.2.3.3 Latency and Processing Time

Latency remained sub-millisecond, with an average processing time of 0.02–0.03 ms and an average buffer wait time of 0.02–0.03 ms, identical to single- and three-client runs, indicating negligible per-packet overhead.

6.2.3.4 Buffer Management

Buffers auto-tuned from 17 kB to 43 kB, with a peak instantaneous usage of approximately 17 kB per client, and no buffer overflows recorded, demonstrating effective handling of bursty traffic.

6.3 System Limitations and Constraints

6.3.1 Identified Limitations

The system encountered several operational challenges across fragmentation, environment limitations, and protocol-specific constraints. Fragmentation rates consistently exceeded 85%, primarily due to the 1500-byte MTU restrictions inherent to the Minikube environment. This excessive fragmentation poses a potential risk to throughput performance, especially under sustained load. Environmental constraints, such as Minikube's limited resource capacity and restricted support for large-scale or multi-node testing, further limited the scope of experimentation. Additionally, protocol-level dependencies introduced complexities, including reliance on a custom packet format, HTTP/3 termination at the proxy, and non-trivial certificate management for local environments. Performance analysis also highlighted bottlenecks under extreme load conditions, particularly in buffer handling, CPU scaling with increased client concurrency, and memory growth driven by the number of active streams. These factors collectively underline the need for robust resource tuning and infrastructure scalability in future deployments.

6.4 Validation and Verification

6.4.1 Functional Validation

The system effectively achieves its core functional objectives. It demonstrates correct stream demultiplexing by accurately separating video, audio, and chat data for inde-

pendent processing. The protocol translation layer reliably converts incoming HTTP/3 requests to HTTP/1.1, enabling compatibility with downstream microservices. Dynamic routing based on YAML configurations ensures flexibility in directing different stream types to their respective service endpoints. Additionally, integration with Apache Pulsar has been validated through successful message forwarding to the appropriate topics, confirming the system's capability to interface with distributed messaging infrastructure.

6.4.2 Performance Validation

From a performance standpoint, the system meets all key targets. It maintains sub-millisecond processing latency, consistently delivering an average of under 0.1 ms even during peak usage. Throughput remains stable across varying load conditions, indicating robust backpressure handling and internal queuing mechanisms. Resource consumption, both in terms of CPU and memory, remains within acceptable limits, affirming the efficiency of the implementation. Furthermore, scalability tests demonstrate linear performance gains with an increasing number of clients, validating the architecture's suitability for deployment in high-concurrency environments.

6.5 Summary

The comprehensive evaluation confirms that the WebTransport stream routing system meets its design objectives effectively. The router demonstrates reliable handling of multiplexed WebTransport streams while maintaining consistently low latency and stable performance under moderate concurrency. The system achieves a 100% success rate in packet reconstruction and stream routing, indicating robust processing logic. Throughput and latency metrics remain stable, even as client load increases, and resource consumption is kept within efficient bounds, highlighting sound memory and CPU management. Furthermore, the router integrates seamlessly into the Kubernetes environment, confirming its suitability for cloud-native deployments.

While the current implementation performs reliably, the evaluation also highlights opportunities for future optimization—particularly in reducing fragmentation overhead and supporting larger-scale scenarios. Nonetheless, the findings validate the feasibility and practical value of stream-level routing using WebTransport. The system lays a solid foundation for deploying real-time, low-latency applications within modern, containerized infrastructure.

Chapter 7

Conclusions & Future Work

This dissertation offers a new and practical way to handle HTTP/3 WebTransport streams in Kubernetes environments. It uses the QUIC protocol and WebTransport Application Programming Interface (API), custom aioquic implementation of scalable routing system with apache pulsar and microservices integration to create a modular and live streaming application demonstrating the power of webtransport protocol over quic within kubernetes environment. The custom router, built with the `aioquic` library is built to manage connection termination and separate streams. It also allows addition of new microservices with update of routing rules without stopping the service. This system successfully signifies the ability to see and control streams to help enable make smart routing and load balancing decisions. These features are important for real-time applications like video streaming, online gaming, and live chat.

In addition to the router, the dissertation shows how different microservices can process video, audio, and chat streams. These services can send data in real-time to Apache Pulsar for further processing, analysis and storage. This setup supports fast and scalable data handling, which is needed for cloud applications. The system was tested under different client loads, and the results were evaluated show casing the efficiency and issues with the system. In summary, this work connects new web transport protocols with Kubernetes platforms. It provides a strong and flexible base for running real-time streaming applications at scale.

7.1 Key Findings

The evaluation demonstrates that the proposed system performs effectively across several critical dimensions. The router handles multiplexed WebTransport streams with low latency and high throughput, even under moderate fragmentation, confirming its suitability

for real-time data handling. Buffer management is robust, preventing overflows and ensuring reliable packet reconstruction during high-load scenarios. Resource utilization across the router and microservices remains lightweight, enabling deployment in constrained environments such as Minikube.

Scalability is another key strength, with the modular architecture supporting the seamless addition of new microservices or router replicas as demand increases. Finally, the configuration-driven approach allows for dynamic updates to routing rules and service endpoints at runtime, improving maintainability and reducing the need for service restarts. These findings validate the system's design and highlight its potential for production use in modern cloud-native environments.

These findings validate the effectiveness of the proposed solution and demonstrate its potential for adoption in production environments.

7.2 Limitations

Although this dissertation has managed to create an effective and working structure of HTTP/3 routing WebTransport streams within Kubernetes, few limitations are idenified which modifies the area of the applicability. It is worth mentioning that when some conditions of workload are observed, high fragmentation rates may degrade the performance which means that a subsequent optimization of packet handling and reconstruction is required. The use of Minikube in the experiment placed restrictions on the network interface MTU size constraining the systems ability to evaluate performance across higher packet sizes. Also, the router in the proposed solution terminates HTTP/3 connections and demultiplex the streams efficiently, but it does not forward HTTP/3 streams to backend microservice server, which hinders end-to-end protocol transparency and interoperability. Furthermore, the use of a custom application packet format in this application, can complicate the integration of the respective system with other conforming WebTransport-based systems. Besides, although multi-client was experimented, much more testing should be conducted in a larger, distributed setting to complete the picture of the resiliency and scaling of the system with different client machines.

These limitations provide opportunities for further research and development.

7.3 Future Work

The solution presented in this dissertation serves as a proof of concept and can be implemented upon for future advancements, and there are several directions for future work.

They are states as follows:

7.3.1 Scaling and Multi-Client Testing

The solution must be evaluated on a larger scale in terms of number of clients on distributed environments in order to assess its robustness and scalability to large-magnitude concurrent loads. This would help understand the usage of the system with a multi-node Kubernetes cluster where resource availability is increased.

7.3.2 HTTP/3 Stream Forwarding

At present, HTTP/3 connections end at the router to be processed on the application layer. Future studies may take into consideration pushing HTTP/3 streams all the way to the microservices and be able to communicate end-to-end with QUIC and low-latency.

7.3.3 Integration with CDN and Streaming Platforms

The proposed solution could be extended to forward processed streams (e.g., video, audio, chat) to a Content Delivery Network (CDN) or streaming server for broadcasting to end-users. This would complete the streaming use case and enable large-scale deployments.

7.3.4 Optimizing Fragmentation Handling

The fragmentation rate observed during experiments highlights the need for optimization. Future work could focus on improving packet handling algorithms at client side to reduce fragmentation to enhance throughput.

7.3.5 Exploration of Media over QUIC Protocol

The IETF is currently working on standardizing the Media over QUIC protocol. This draft defines how media streams are transmitted over QUIC. This protocol could be explored to enhance the solution's capabilities and align with emerging standards.

7.3.6 Generalizing the Packet Format

The current solution uses a custom packet format for stream identification and routing. Future work could focus on developing a more generalized approach that supports interoperability with other WebTransport implementations.

7.3.7 Advanced Ingress Controllers and Gateway APIs

The development of advanced ingress controllers and gateway APIs for Kubernetes could provide native support for HTTP/3 streams in future. Monitoring these developments is a good idea.

Bibliography

[1] J. Iyengar and M. Thomson, "Quic: A udp-based multiplexed and secure transport." IETF RFC 9000, 2021. Accessed August 2025.

[2] M. Bishop, "Http/3." IETF RFC 9114, 2022. Accessed August 2025.

[3] Kubernetes, "Kubernetes documentation," 2024. Accessed August 2025.

[4] aiortc, "aioquic: Quic and http/3 implementation in python," 2024. Accessed August 2025.

[5] DataStax, "Apache pulsar helm chart," 2024. Accessed August 2025.

[6] Kubernetes SIGs, "Minikube documentation," 2024. Accessed August 2025.

[7] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach*. Pearson, 7th ed., 2017.

[8] V. Vasiliev, "The webtransport protocol framework." IETF Internet-Draft, 2023. Accessed August 2025.

[9] NGINX Inc., "Nginx ingress controller documentation," 2024. Accessed August 2025.

[10] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed streaming platform," in *Proceedings of the ACM SIGMOD International Conference*, ACM, 2011.

[11] R. Marx, "Http/3 core concepts," 2021. Smashing Magazine, Accessed August 2025.

[12] Mozilla Developer Network, "Webtransport api," 2024. Accessed August 2025.

[13] MetalLB, "Metallb documentation," 2024. Accessed August 2025.

[14] Apache Software Foundation, "Apache zookeeper documentation," 2024. Accessed August 2025.

[15] Amazon Web Services, "Aws documentation," 2024. Accessed August 2025.

[16] Microsoft Azure, "Azure documentation," 2024. Accessed August 2025.

[17] Google Cloud, "Google cloud documentation," 2024. Accessed August 2025.

[18] Kubernetes SIGs, "kind (kubernetes in docker) documentation," 2024. Accessed August 2025.

[19] Rancher Labs, "K3s documentation," 2024. Accessed August 2025.

[20] HAProxy Technologies, "Haproxy configuration manual," 2024. Accessed August 2025.

[21] HAProxy Technologies, "Haproxy kubernetes ingress controller documentation," 2024. Accessed August 2025.

[22] quic go, "A quic implementation in pure go," 2024. Accessed August 2025.

[23] Cloudflare, "quiche: Savoury implementation of the quic transport protocol and http/3," 2024. Accessed August 2025.

[24] Angie, "Angie web server documentation," 2024. Accessed August 2025.

[25] Wireshark Foundation, "Wireshark documentation," 2024. Accessed August 2025.

[26] Python Software Foundation, "Python 3 documentation," 2024. Accessed August 2025.

Appendix A

List of Abbreviations

Abbreviation	Full Form
API	Application Programming Interface
ConfigMap	Kubernetes ConfigMap resource
CPU	Central Processing Unit
CSV	Comma-Separated Values
DNS	Domain Name System
HTTP/3	HyperText Transfer Protocol version 3
HTTPS	HTTP over TLS
IP	Internet Protocol
JSON	JavaScript Object Notation
JPEG	Joint Photographic Experts Group
K8s	Kubernetes
LB	Load Balancer
Minikube	Minimal single-node Kubernetes cluster
Pulsar	Apache Pulsar streaming platform
QUIC	Quick UDP Internet Connections
Secret	Kubernetes Secret resource
Service	Kubernetes Service resource
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URL	Uniform Resource Locator
WebTransport	WebTransport API over HTTP/3
YAML	YAML Ain't Markup Language

Table A.1: List of Abbreviations

Appendix B

Protocol Layering Clarification

This appendix clarifies the relationship between **WebTransport** Streams, **HTTP/3** Streams, and **QUIC** Streams, as these terms are used interchangeably in parts of the dissertation.

- **QUIC** (*Quick UDP Internet Connections*) is a transport-layer protocol that operates over UDP and provides multiplexed, secure, low-latency streams.
- **HTTP/3** is the application-layer protocol that defines how HTTP semantics are carried inside QUIC streams.
- **WebTransport** is a web API that exposes both reliable (stream) and unreliable (datagram) communication channels to JavaScript clients. These channels are transported as HTTP/3 streams, which are themselves QUIC streams with additional HTTP/3 framing and control headers.