Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Leveraging QUIC & HTTP/3 For Routing Streams In Kubernetes

Cornel Jonathan Cicai
Supervisor: Dr. Stefan Weber

A dissertation submitted in fulfilment
of the requirements for the degree of
Integrated Masters in Computer Science (MCS)

Submitted to the University of Dublin, Trinity College, April 2024

Acknowledgements

I want to specially thank Dr. Stefan Weber for supervising this project, for providing guidance, and for his seemingly unending patience. I want to thank my family for supporting me when I needed it the most, and to thank my close friends, Martin, Gintare, Steven, Terlo, and Micheal, for believing in me. I want to thank my partner Katherine, for always being there for me. And most importantly, I thank God for putting all these people in my life - I couldn't have done it without them.

Cornel Jonathan Cicai

University of Dublin, Trinity College
April 2024

Abstract

QUIC and HTTP/3, the latest standards in network protocol technologies, stand to revolutionise how web-based services communicate. Container orchestration tools such as Kubernetes predominantly rely on the HTTP/2 protocol layered over TCP for internal communication between services. This method of communication, while effective, poses limitations in modern environments. The shift from the traditional HTTP/2 over TCP to HTTP/3 is imperative. HTTP/3, with QUIC, offers improved performance, security, and reliability by fundamentally changing how data is transmitted over the internet. This advancement necessitates a change in the internal communication strategies of Kubernetes, as the orchestration of containerised services needs to accommodate the enhanced capabilities of QUIC; namely its multiplexing and connection migration features. Despite Kubernetes' wide adoption as a microservices orchestrator, its native support for HTTP/3 is limited, reflecting a significant gap in its networking stack. As well as this, growing demand for efficient management of diverse data streams, which are part of multiplexed connections like QUIC, within microservices, presents a critical challenge for developers and network engineers. To address these challenges, this project proposes a solution by integrating "ANGIE", a modified ingress controller based on NGINX, to support HTTP/3 traffic within a Kubernetes cluster. By leveraging ANGIE, the system can harness the full spectrum of HTTP/3 features, facilitating a smoother and more efficient ingress routing process. This in turn enables efficient routing of streams within a HTTP/3 connection, by using HTTP headers to determine the content type of each stream. This approach not only showcases the potential of HTTP/3 in optimising network communications within Kubernetes but also underscores the significant benefits QUIC streams bring to the table. Through the proposed implementation, Kubernetes can effectively manage multiplexed HTTP/3 connections.

Contents

Chapter 1
Introduction

1.1 Background

Kubernetes relies heavily on HTTP as the underlying protocol for inter-service communication. Traditionally, this communication has been facilitated by HTTP/2 layered over TCP, which was a significant improvement over its predecessor, particularly with its ability to handle multiplexed streams. However, the evolving needs for faster, more efficient, and secure web interactions have brought to light certain limitations inherent to TCP, such as head-of-line blocking and the overhead introduced by its connection setup, which can hinder performance. QUIC confronts these TCP issues by offering reduced connection and transport latency, improved congestion control, and greater security through integrated TLS 1.3 encryption. The development of HTTP/3, which uses the QUIC transport protocol, represents a substantial leap forward. Adapting Kubernetes' internal communications to leverage HTTP/3 over QUIC necessitates modifications to the existing networking models to accommodate the concurrent, multiplexed communication streams that define QUIC's architecture. The transition from HTTP/2 to HTTP/3 in Kubernetes is essential to meet the current needs of distributed systems and microservices for faster response times, better security, and improved overall performance.

1.2 Project Objective

This project aims to design and develop a system that uses a "stream splitting" mechanism, designed to distribute individual streams from a multiplexed QUIC connection to the appropriate microservices based on the content within these streams. This can be done by using the new "ANGIE" fork of the NGINX Ingress Controller implementation, which enables TLS Termination and consequently, HTTP header inspection, which will be used to identify the content of the streams. This approach marks a transition from a one-size-fits-all model to a nuanced, content-aware routing mechanism that aligns with microservices' granular requirements. Leveraging QUIC is also expected to mitigate the inefficiencies of TCP, particularly in terms of head-of-line blocking and connection overhead. The result will push Kubernetes networking towards a model where different types of data can be prioritised and managed independently.

1.3 Project Overview

Chapter 2 - State Of Art provides a detailed exploration of modern networking technologies and protocols, focusing on the OSI and TCP/IP models, network security measures, and the practical application of containerisation technologies like Kubernetes. It delves into the complexities of internet communications, addresses theoretical and practical aspects of network models, and discusses advancements made in Transport Layer, highlighting the evolution from TCP to QUIC. Additionally, it explores the impact of containerisation on web application deployment, emphasising the use of Kubernetes for orchestrating containerised applications across various environments.

Chapter 3 - Problem Statement addresses the critical challenge of efficient stream handling within multiplexed network connections, a common issue in modern networking that impacts the performance and scalability of web applications. The inherent limitations of traditional TCP connections, such as head-of-line blocking and inefficient congestion control, often degrade the performance of microservices architectures. To tackle these inefficiencies by leveraging QUIC, this chapter proposes routing streams based on their content, enhancing throughput and reducing latency in containerized environments orchestrated by Kubernetes.

Chapter 4 - Design outlines a design strategy for effectively managing streams within a multiplexed HTTP/3 connection using QUIC and Kubernetes. The chapter also proposes a HTTP header based solution for stream splitting and routing, using the headers to identify the content type. It also emphasises the importance of TLS termination at the ingress controller to facilitate the inspection and appropriate routing of encrypted traffic. It describes a high-level architecture involving clients, ingress controllers, and backend services that together enhance the handling of network traffic and data streams.

Chapter 5 - Implementation outlines the implementation of a system architecture focused on resolving the challenges associated with efficient stream handling in multiplexed network connections. It begins with a detailed examination of Kubernetes deployment options, favouring a local setup for development and testing purposes. The chapter progresses to an in-depth discussion on the selection and implementation of the ingress controller, outlining why ANGIE is used eventually. Additionally, the chapter details the development of the other system components: the client setup for establishing connections, and the backend services tasked with processing these routed streams.

Chapter 6 - Evaluation demonstrates the working solution and analyses its capabilities against other potential implementations of systems that address the problem.

Chapter 7 - Conclusion concludes the dissertation, outlining the solutions limitations and potential future work.

Chapter 2
State-of-Art

This chapter provides an extensive overview of modern networking models, security protocols, transport layer protocols, and containerisation technology, to provide readers with the opportunity to grasp the intricacies of internet communications, and how modern web applications can be deployed using Kubernetes.

2.1 Network Reference Models

We begin by addressing the basics of computer networking. There are various stages involved when data is transferred, with each "layer" having its own standardised protocol. There are two main models that are used in online tutorials and lecture halls all over the world, the "OSI Reference Model" (International Organization for Standardization, 1984) and the "TCP/IP Model" (Jain, 2023).

2.1.1 In Theory: OSI Reference Model

The Open Systems Interconnection (OSI) Reference Model provides a structured and detailed hierarchical model of networks, which delineates networking functionalities into seven distinct layers.

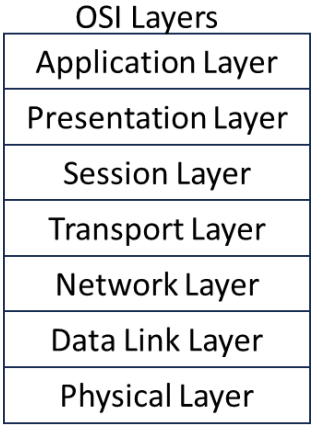

Figure 2.1 OSI Model Layers - An overview of the seven layers of the OSI Model, with the Physical Layer as the foundation, which establishes the hardware basis of networking, up to the Application Layer, where application protocols operate.

2.1.1.1 Physical Layer

The Physical Layer provides the medium through which the data packets as a whole can be transported, over physical connections such as cable connections, optical fibre between entities in the Data Link Layer. The Physical Layer's purpose is to deliver bits in the same order in which they were submitted.

2.1.1.2 Data Link Layer

This layer is made up of the combination of the Media Access Control (MAC) and Logical Link Control (LLC) sublayers. The MAC sublayer manages access to the physical medium, is responsible for addressing at the hardware level, and error checking using Cyclic Redundancy Check (CRC) (Sobolewski, 2003). The LLC sublayer is responsible for framing and flow control, and provides an interface for higher layers by facilitating the addressing at the network layer. In summary, the Data Link Layer's purpose is to provide a common interface for network protocols, control access to the physical layer, packet framing, flow control, and error detection.

2.1.1.3 Network Layer

The Network Layer is responsible for routing, logical addressing, and packet forwarding. It maintains a routing table by exchanging routing information with neighbouring routers, so that it can find the optimal path to destination addresses, and it manages logical addressing by assigning unique Internet Protocol (IP) addresses to devices. The Network Layer is also responsible for breaking down data packets into smaller fragments if they are too large to be transmitted over the network medium, through a process called fragmentation. These fragments are then reassembled again at the network layer by the receiving device.

2.1.1.4 Transport Layer

The Transport Layer is mainly responsible for connection establishment, ensuring the data transmission between end systems using Transport Layer protocols like TCP and UDP. It is also responsible for flow control, congestion control, data segmentation and reassembly.

2.1.1.5 Session Layer

A session is an abstract concept that refers to a temporary interactive information exchange between two applications. A session is different from a connection in the sense that a connection is between two network entities, but a session is specifically between two application entities. The Session Layer is mainly responsible for session establishment, which typically involves the exchange of parameters and security token management, and checkpoint creation, so a session can be resumed rather than restarted completely. The functions of the Session Layer are usually integrated into application-level protocols, but the OSI model makes it a distinct layer in an attempt to highlight the importance of sessions in telecommunication, and to separate the ideas of connection and session.

2.1.1.6 Presentation Layer

This layer serves as a translator for the network, with its primary role being to ensure that data is readable when sent between two end to end systems, by formatting data into the appropriate encodings. It also is responsible for data compression, data encryption and data decryption. Similarly to the Session Layer, the Presentation Layers functionality is often, in practice, implemented as part of the application-level protocol, and has been identified as its own distinct layer by the OSI model to highlight the importance of data presentation in telecommunications.

2.1.1.7 Application Layer

This layer enables users to interact with the network through a software interface, which provides high-level services that directly facilitate end to end communication between applications running on different hosts.

2.1.1.8 Summary Of OSI Model

In Figure 2.2, we see data being transmitted from one system to another. The three highest layers work together to create and format the data, as well as managing the application session. After the transport protocol in the Transport Layer establishes a connection to the destination, the data then gets encapsulated by the other layers as it passes through them, until a complete data frame is created. This data frame gets transmitted across the physical medium between the two endpoints, potentially going through routers and relay systems, which do not unpack the data at a level higher than the network level as it is not necessary for packet forwarding. Once it reaches the destination, the frame is unpackaged and the data is sent up through the layers, so it can be utilised by the destination application.

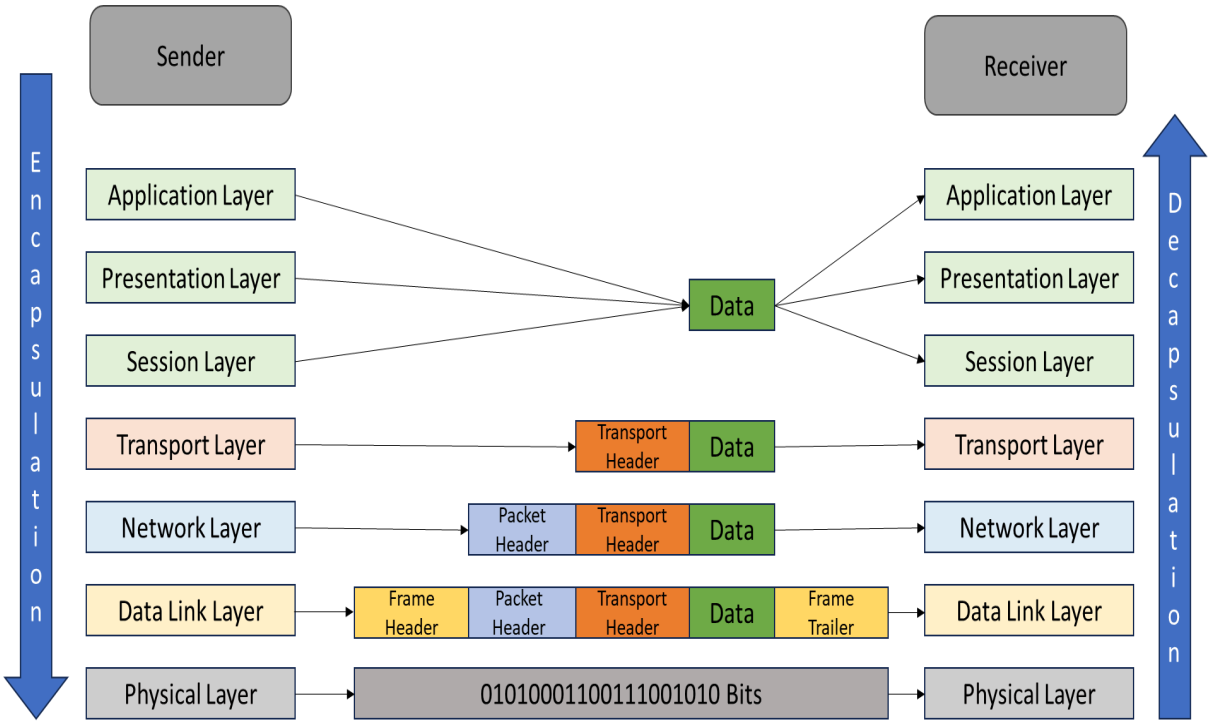

Figure 2.2 The Encapsulation & Decapsulation of Data Through The OSI Layers - The diagram illustrates the journey of data as it is sent from a sender to a receiver, showcasing the sequential addition and removal of headers and trailers at each OSI layer.

The OSI Reference Model is a very detailed and descriptive framework, but in practice, its granularity actually tends to overly complicate the representation of telecommunications, and is mostly regarded as outdated.

2.1.2 In Practice: TCP/IP Reference Model

The TCP/IP Model is usually preferred in a practical environment, as it more closely aligns itself with the reality of the Internet and network communications today. It streamlines the seven layers of the OSI Model into four.

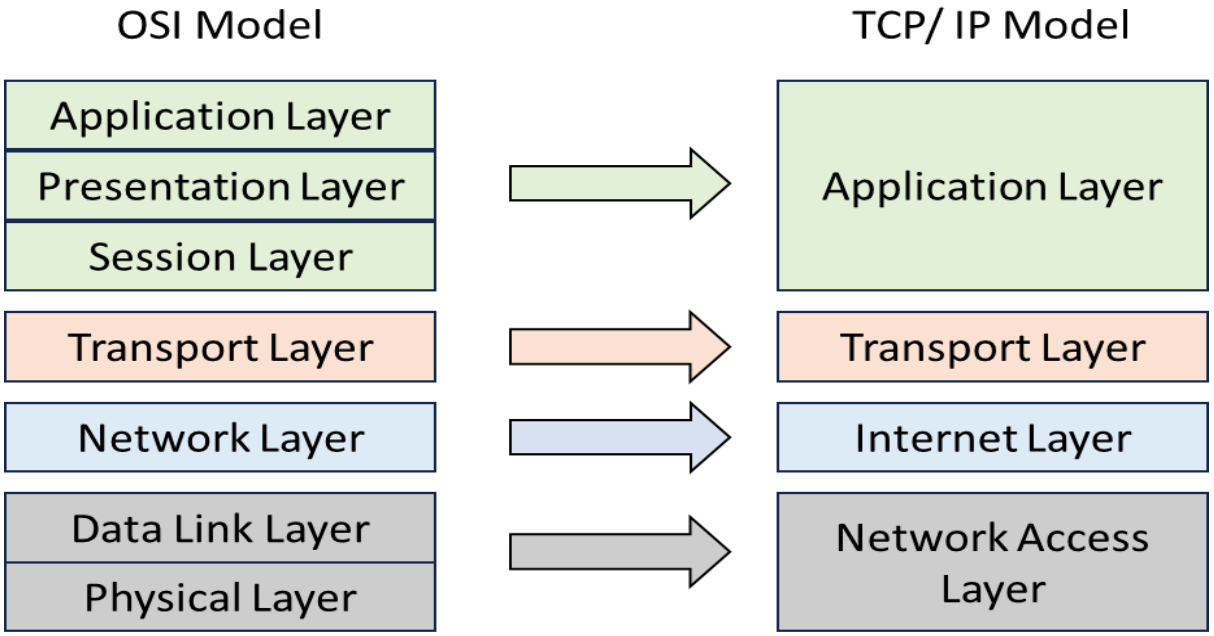

Figure 2.3 Streamlining The OSI Model - A side-by-side mapping illustrating the consolidation of the OSI model's seven distinct layers into the four broader layers of the TCP/IP model

2.1.2.1 Network Access Layer

Also known as the Link Layer, the Network Access Layer encapsulates the physical and data link aspects of networking. This layer is equivalent to the combination of the Physical and Data Link Layers in the OSI model. It is responsible for hardware addressing, through MAC addresses, and the physical transmission of data over various media types such as Ethernet, Wi-Fi, or fibre optics. It controls the hardware devices and media that make up the network. Beyond framing and addressing, it performs error detection and correction to ensure that packets are delivered reliably to the correct destination.

2.1.2.2 Internet Layer

The Internet Layer, analogous to the Network Layer in the OSI model, is tasked with routing data packets across different networks. It employs the Internet Protocol (IP) to define how data should be packetized, addressed, transmitted, routed, and received at the destination. This layer ensures logical addressing through IP addresses and determines the most efficient path for data to travel across networks. The Internet Layer handles packet fragmentation and reassembly, and it deals with errors such as timeouts and undeliverable packets.

2.1.2.3 Transport Layer

The Transport Layer of the TCP/IP model parallels the same-named layer in the OSI model. It is responsible for providing communication sessions between applications across a

network. This includes establishing, maintaining, and terminating connections. It offers reliable data transfer through the Transmission Control Protocol (TCP) and supports a connectionless service through the User Datagram Protocol (UDP). This layer handles segmentation and reassembly of data, provides flow control to prevent network congestion, and manages error detection and recovery.

2.1.2.4 Application Layer

In the TCP/IP model, the Application Layer encompasses the responsibilities of the Application, Presentation, and Session Layers of the OSI model. This layer interfaces directly with software applications to provide end-to-end communication services. It deals with issues such as session management, which in TCP/IP is often handled directly within the application protocols rather than by the network. Data presentation, including encryption, decryption, and encoding, is also managed at this level. It provides a set of interfaces and protocols for specific network services, such as HTTP for internet browsing, SMTP to send emails, FTP for file transfers, and DNS for domain name resolution.

2.1.2.5 Summary of TCP/IP Model

The TCP/IP model provides a simpler representation of network architecture, due to compressing down to only four layers with very clear and distinct functionalities. Data is created and formatted for transmission at the Application Layer, and is then passed down to the Transport Layer, which establishes a connection to the destination and carries the application data in its payload. The Transport Layer protocol packet is encapsulated with an IP header, so that it can be sent through the Internet Layer, which in turn is translated into simple bits for transmission across the Network Layer. The receiver decapsulates the data as it passes through its layers, and finally the application data is available for use.

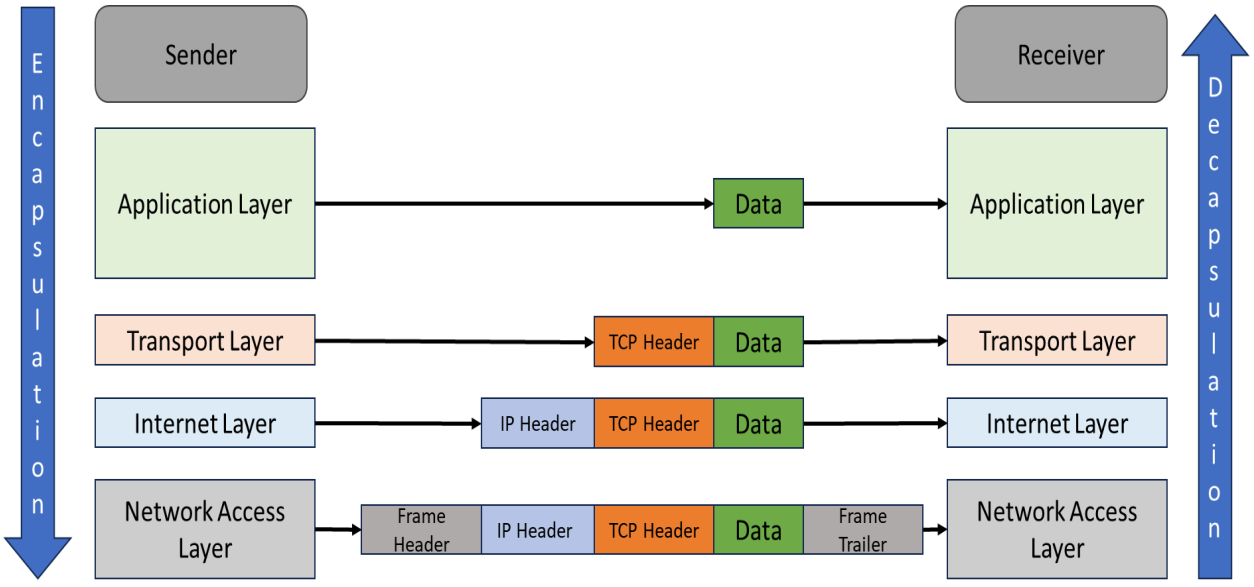

Figure 2.4 The Encapsulation & Decapsulation of Data Through The TCP/IP Model Layers - The diagram illustrates the journey of data as it is sent from a sender to a receiver, showcasing the sequential addition and removal of headers and trailers at each layer.

The model's alignment with the needs of the Internet gives it a practical advantage, leading to it being widely adopted by the industry, which in turn has led to extensive support in terms of protocols developed to fit within this model.

2.2 Network Security

To protect a network and the data transmitted within it, some form of network security needs to be implemented. Network security is established at different layers of the network through various mechanisms and technologies, tailored to address the specific vulnerabilities and threats associated with each layer.

2.2.1 Security Measures At All Layers

The following are some examples of security measures and how they are implemented at all layers:

- **Application Layer:** Applications can implement authentication mechanisms to verify the identity of users and digital signatures can be used to verify the integrity and authenticity of data exchanged between applications, ensuring that it has not been tampered with during transmission.

- **Transport Layer:** Secure Sockets Layer (SSL) and its successor Transport Layer Security (TLS) are cryptographic protocols used to establish secure communication channels between clients and servers at this layer. They provide encryption, ensuring that data transmitted over the network is protected from eavesdropping and tampering.

- **Internet Layer:** Security measures at this layer focus on protecting network infrastructure and preventing unauthorised access to network resources. Firewalls are network security devices that monitor and control incoming and outgoing traffic based on configured security rules. They act as a barrier between internal networks and external networks, to prevent unauthorised access and malicious attacks. Protocols like Internet Protocol Security (IPsec) provide security at the network layer as well, offering encryption, authentication, and integrity protection for IP packets.

- **Link Layer:** Examples of security measures at this layer include MAC address filtering and Wi-Fi Protected Access (WPA). MAC address filtering allows network administrators to control which devices can connect to the network based on their unique MAC addresses. WPA and its successors, WPA2 and WPA3, are security protocols used to secure wireless networks. They provide encryption and authentication mechanisms to protect data transmitted over Wi-Fi networks from interception and unauthorised access.

2.2.2 SSL & TLS

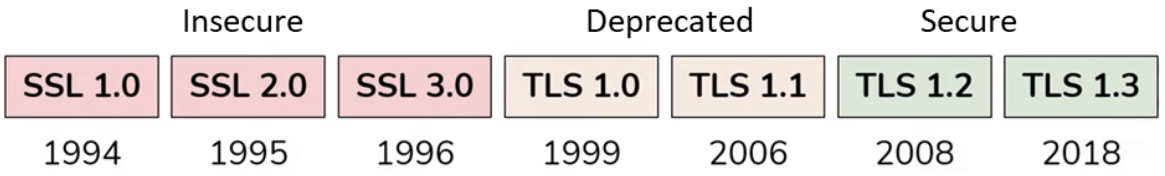

Figure 2.5 Timeline of SSL/TLS Versions

Insecure: has known exploits Deprecated: no longer recommended Secure: safe to use

Secure Sockets Layer (SSL) was originally developed by Netscape in 1994, and has since evolved to the standard we have today. It was renamed Transport Layer Security (TLS) in 1999, motivated partly by concerns regarding trademarks and patents associated with SSL, but also to reflect the collaborative effort of the industry to advance secure communication beyond just secure sockets.
This section begins by explaining fundamental cryptography concepts, and then explaining the latest TLS versions.

2.2.2.1 Cryptographic Keys

In cryptography, a key is used to control the encryption and decryption processes. A Symmetric Key is used for both encryption and decryption. Algorithms like Advanced Encryption Standard (AES) and Data Encryption Standard (DES) are used to generate these keys. The key must be shared secretly between parties, as anyone with the key can decrypt the information.

An Asymmetric Key Pair refers to a pair of mathematically linked keys that can only decrypt messages encrypted by the other, ie, in the Key A and Key B pair, if Key A encrypts the data, only Key B can decrypt it, and vice versa. Having one of the two keys doesn't necessarily mean you have or can obtain the other, as it is a non-trivial task to derive one from the other. RSA (Rivest et al., 1978) and Elliptic Curve Cryptography (ECC) (Koblitz, 1987) are common algorithms used to generate these key pairs. Usually servers will share one key, referred to as the Public Key, while keeping the second one secret, known as the Private Key.

2.2.2.2 Certificates & Certificate Authorities

Certificates are digital documents used to certify the ownership of a public key by the named subject of the certificate. They are issued by Certificate Authorities (CAs).

A TLS certificate must contain at least:

- The certificate holder's name.
- The certificate holder's public key.
- The certificate's expiration date.
- The certificate issuer's name (the CA).
- A digital signature from the CA, which verifies the certificate's authenticity.

Certificate Authorities (CAs) are trusted entities that issue these digital certificates. They verify the identity of entities requesting a certificate and digitally sign the certificate with their own private key, allowing end-users to verify the authenticity of the certificate with the CA's public key. The digital signature employed to digitally sign the certificate is a cryptographic mechanism used to verify its authenticity and integrity. It is created by taking a hash, often called a "digital fingerprint", of the document's contents and then encrypting this hash with the signer's private key. To check a digital signature, the recipient uses the signer's public key to decrypt the signature, retrieving the original hash value produced by the signer. The recipient then independently calculates the hash of the received document, and trusts the certificate if the hashes match.

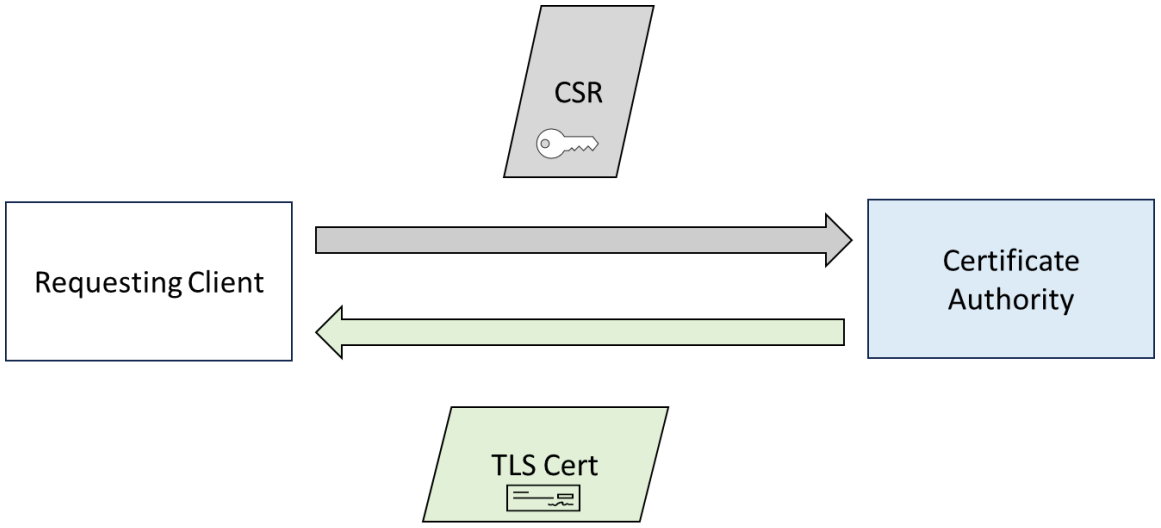

Figure 2.6 TLS Certificate Issuance Process - The requesting client sends a Certificate Signing Request (CSR) to a Certificate Authority (CA), containing the clients public key. The CSR is validated by the CA and issues a TLS certificate

2.2.2.3 TLS 1.2 Handshake

In TLS 1.2, the session key establishment process is essential for securing communications between a client and server. This process varies depending on the chosen cipher suite and generally involves either the client generating a pre-master secret or both parties working together to create a shared secret key. These methods are fundamental to establishing secure connections over networks where data security is paramount. Understanding these initial steps in the TLS 1.2 handshake is key to appreciating how it protects data exchanges by encrypting information transmitted between parties.

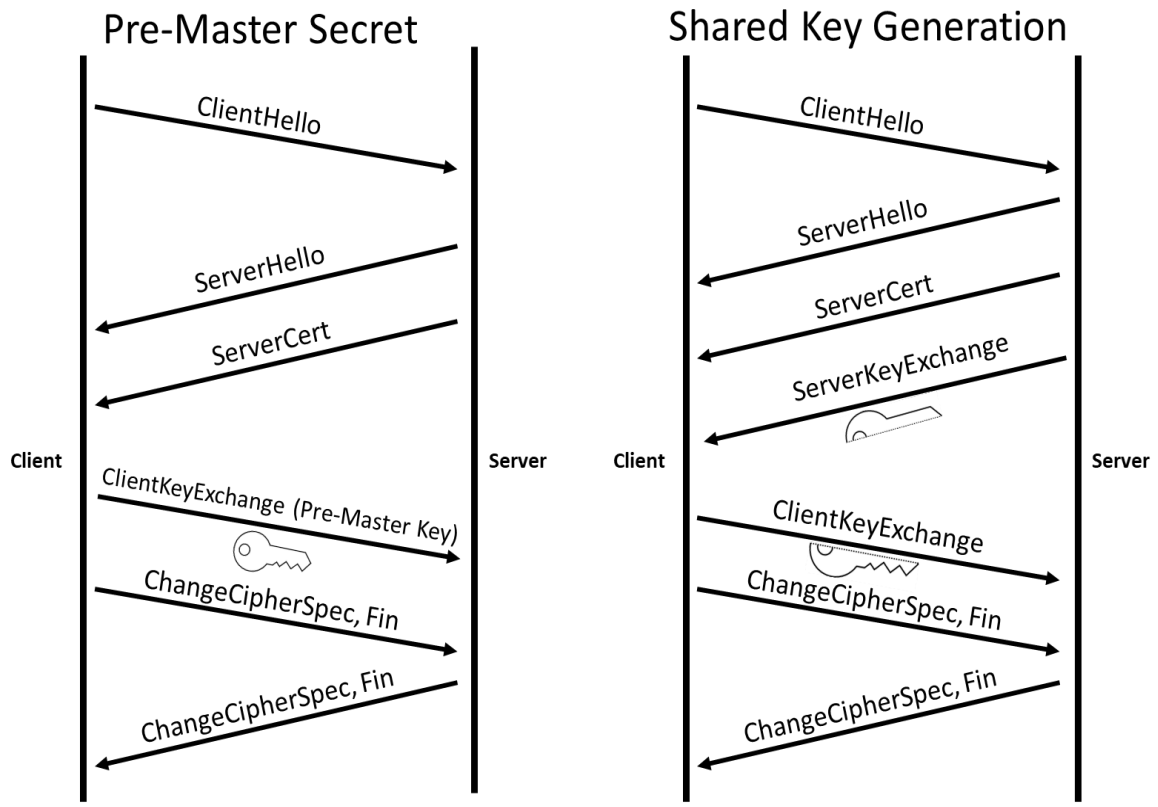

Figure 2.7 The two TLS1.2 handshake approaches: "Pre-Master Secret" (left) & "Shared Generation" (right). The Pre-Master Secret approach results in the client issues the pre-master key that is used to generate the actual master secret. The Shared Key Generation approach involves both parties computing the session key.

Using the Pre-Master Secret approach involves:

- **ClientHello:** The client sends a list of supported cipher suites.
- **ServerHello:** The server chooses a cipher suite that results in a pre-master secret being used, and informs the client.
- **ServerCert:** The server also sends its certificate, containing its public key.
- **ClientKeyExchange:** After verifying the certificate, the client generates a pre-master key, used to calculate the actual master secret, and sends it using the server's public key.
- Both parties compute the master secret and end the handshake.
- **ChangeCipherSpec & Fin Messages:** This completes the handshake.

Using the Shared Key Generation approach involves:

- **ClientHello:** The client sends a list of supported cipher suites.
- **ServerHello:** The server chooses a suite requiring Shared Key Generation.
- **ServerCert:** The server also sends its certificate, containing its public key
- **ServerKeyExchange:** The server sends its calculated portion of the key.
- **ClientKeyExchange:** After verifying the certificate, the client generates the full key using its portion of the key and the received portion from the server, then sends its own portion of the key to the server. Both the server and client have the shared secret key generated .
- **ChangeCipherSpec and Fin Messages** complete the handshake.

2.2.2.4 TLS 1.3 Handshake

TLS 1.3 enhances security by mandating the shared generation approach in all key exchanges, and dropping support for less secure cipher suites and features that don't support this. This means that only protocols like Ephemeral Diffie-Hellman are supported, so both client and server partake in generating the shared key. It also streamlines the handshake process by including a 0-Round Trip-Time (RTT) mode that allows "early data" to be sent even before the handshake is completed in subsequent connections, further reducing latency.

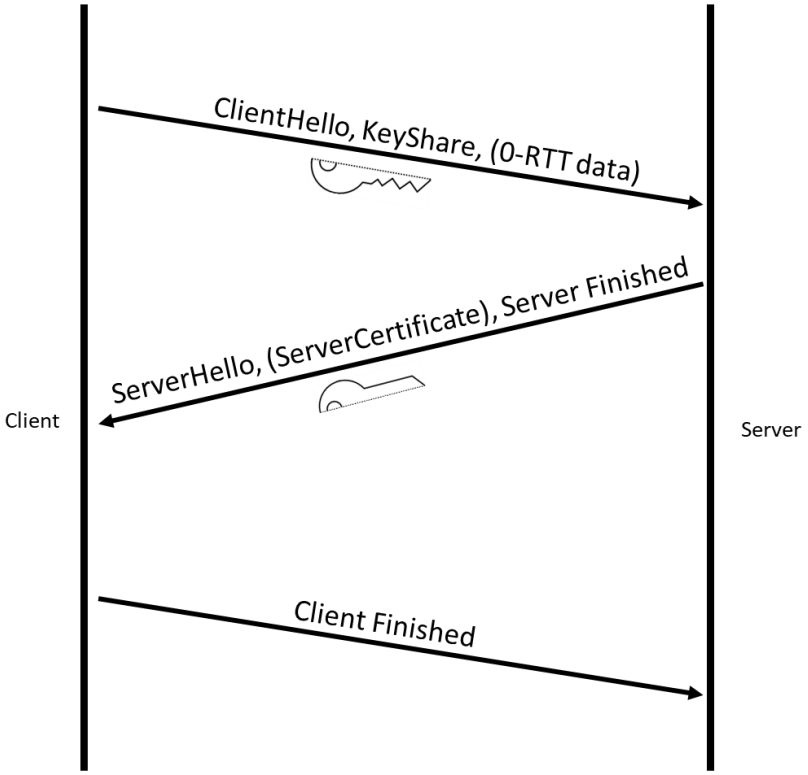

Figure 2.8 The TLS/1.3 Handshake - the streamlined sequence promotes faster secure communications by condensing the handshake to fewer steps and enabling encrypted data transfer at the earliest stage possible.

- **ClientHello:** The client sends a list of supported cipher suites and its portion of the key in the "KeyShare" extension. It may also include early data in "0-RTT" mode.

- **ServerHello:** The server selects the cipher suite, and responds with its "KeyShare" for the chosen key exchange method. It may send a "PreSharedKey" extension if resuming a session.

- **Server Certificate:** If this is the first time the client is connecting, the server includes its certificate and a certificate verify message.

- **Server Finished:** The server sends a Finished message, providing cryptographic proof that it possesses the private key associated with the certificate and finalising its part of the handshake.

- **Client Finished:** The client responds with a Finished message, encrypted with the derived keys, completing the handshake.

- **Application Data:** Following the handshake, all application data is encrypted with the agreed-upon key, ensuring secure communication.

2.2.2.5 Comparing Handshakes

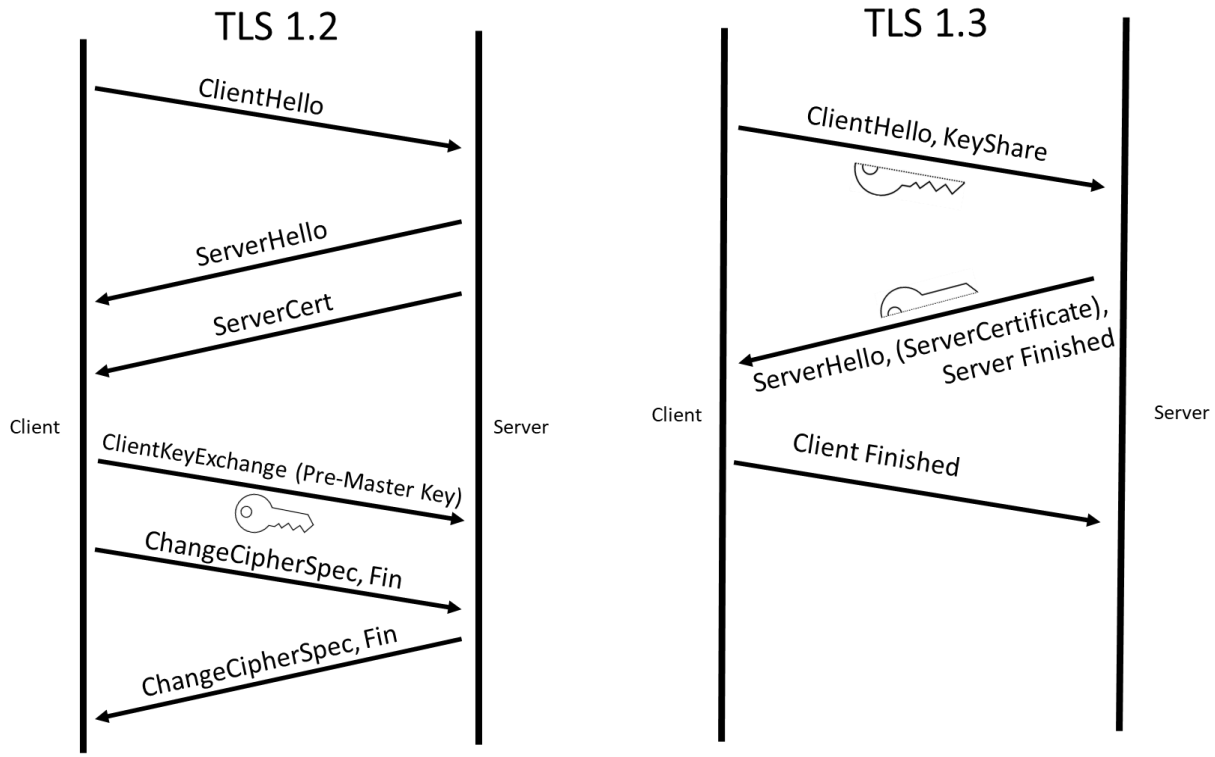

Figure 2.9 TLS/1.2 (left) and TLS/1.3 (right) Handshakes - While TLS 1.2 illustrates a traditional two-round-trip exchange involving a pre-master secret, TLS 1.3 demonstrates a condensed one-round-trip process, integrating KeShare in the initial message to reduce latency and improve security.

2.2.2.5.1 Number of Round Trips

TLS 1.2: The full handshake process requires two round-trips between the client and the server to establish a secure connection. This can introduce latency, particularly noticeable in connections that need to be established quickly or over long distances.

TLS 1.3: One of the most significant improvements is the reduction to just one round-trip in the handshake process, enhancing connection speed and efficiency.

2.2.2.5.2 Cipher Suites & Protocol Negotiation

TLS 1.2: During the handshake, the client and server negotiate the cipher suite from a list provided by the client. The server selects the cipher suite to be used for the session.

TLS 1.3: Simplifies this process by requiring all implementations to support a specific set of algorithms, reducing the negotiation complexity. It removes the negotiation for hash and key exchange methods, focusing only on the negotiation for symmetric encryption algorithms. This not only streamlines the process but also eliminates the support for weaker algorithms, enhancing security.

2.2.2.5.3 Conclusion Of Comparison

The transition from TLS 1.2 to TLS 1.3 brings about substantial improvements in security, efficiency, and speed. By streamlining the handshake process, enforcing stronger cryptographic standards, and removing obsolete and vulnerable options, TLS 1.3 significantly enhances the security and performance of secure communications on the internet.

Still, TLS 1.2 remains a viable option, and is widely used. The adoption of TLS 1.3 is encouraged to leverage the latest in cryptographic protocols for secure internet communications.

2.3 Transport Layer Protocols

This section will outline the workings of the three fundamental transport layer protocols: Transmission Control Protocol (TCP) (Postel, 1981), User Datagram Protocol (UDP) (Postel, 1980), and QUIC (Iyengar & Thomson, 2021). We will delve into the packet structures, connection establishment, and flow control mechanisms, so that we can evaluate and compare their functionalities to understand their distinct roles and efficiencies in data transmission across the internet.

2.3.1 TCP

TCP (Postel, 1981) was designed to offer reliable information transfer of data segments, with mechanisms in place for retries and ordering of out-of-order segments. It ensures data integrity through sequence numbers and acknowledgments, establishing a reliable connection-oriented transmission. TCP's flow control and congestion management protocols maintain network stability and efficiency, and have made it foundational for internet communication.

2.3.1.1 TCP Segments

In TCP, the sender must break up the data into segments made up of the TCP header information and the data payload itself. This header adds a lot of overhead to each segment.

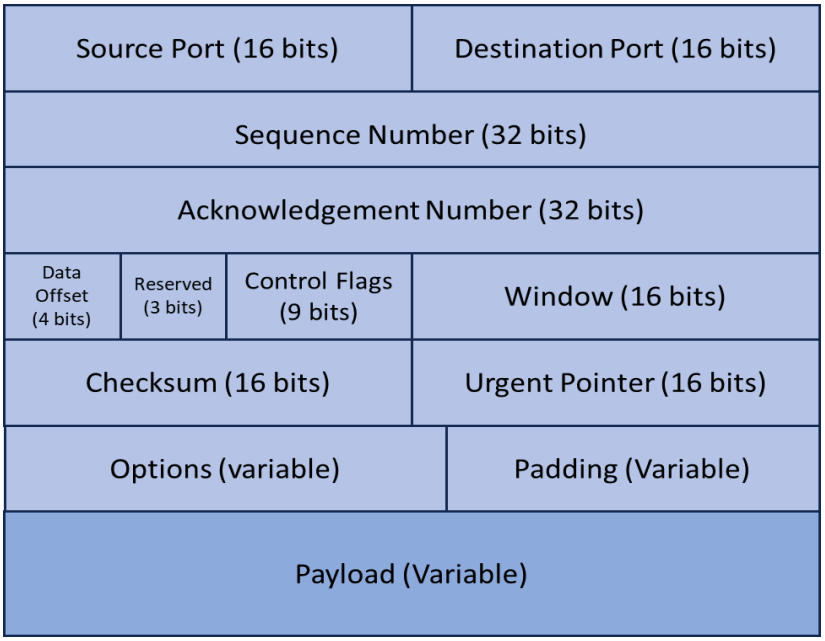

Figure 2.10 Structure of a Standard TCP Segment - highlighting the overhead of the TCP header required to send a payload.

A TCP segment is split up into the following components:

- **Source & Destination Port** (16 bits each)
- **Sequence Number (32 bits)**: SN of current segment
- **Acknowledgement Number (32 bits):** SN of last received segment
- **Data Offset (4 bits)**: specifies the size of the TCP header itself
- **Reserved Field (3 bits)**: reserved for potential future use
- **Control Flags (9 bits):** set based on the purpose of the segment
- **Window Size (16 bits)**: the receive window size of the receiver
- **Checksum (16 bits):** checksum value, used to detect transmission errors
- **Urgent Pointer (16 bits):** points to the SN of the segment following the urgent data
- **Options (Variable length):** optional, can include additional TCP options like timestamps
- **Padding (Variable length):** added to align the options field on a 32-bit boundary
- **Payload (MSS - Data Offset value):** the actual payload itself

2.3.1.2 Connection Establishment

The “three-way-handshake” has become a staple of networking, as it’s a fundamental process for establishing reliable connections between hosts. It ensures that both parties are aware of each other, and that they agree on essential parameters before data transmission, such as the initial sequence numbers, window size and maximum segment size.

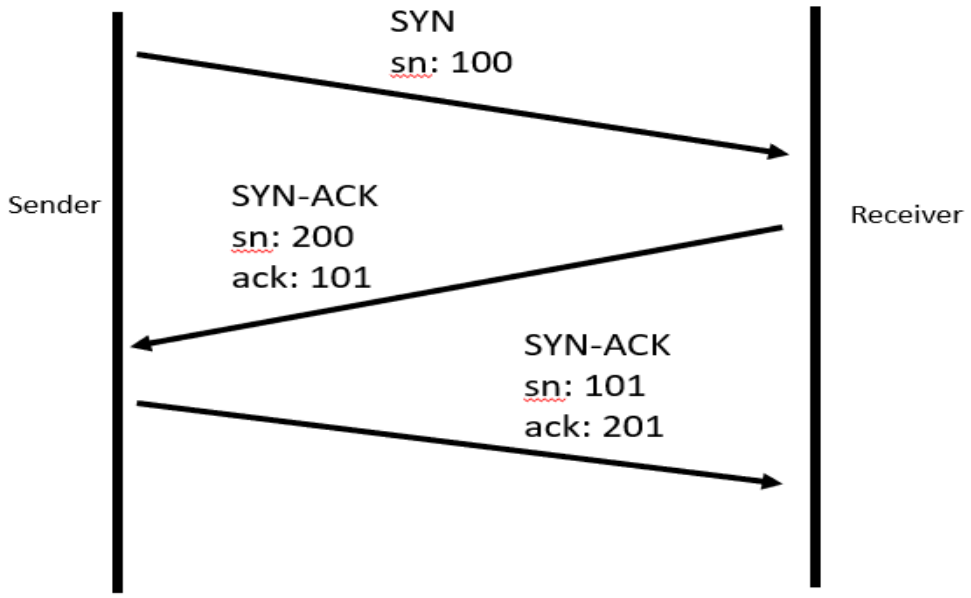

Figure 2.11 The TCP Three-Way Handshake Process - the sequence of packets exchanged to establish a reliable TCP connection

- **SYN:** The client begins by sending a Synchronise (SYN) packet, informing the server that it wishes to establish a connection. The packet contains information such as an initial sequence number (SN) that marks the beginning of the transmission.

- **SYN-ACK:** If the server wishes to accept the connection, it replies with a Synchronise Acknowledgement (SYN-ACK) packet. This acknowledges the last received packet, by specifying the initial SN, and also allows the server to set its own SN, so that the client can also acknowledge the packets sent from the server.

- **ACK:** Finally, the client replies with an ACK packet to acknowledge the server's last packet, which completes the handshake and establishes the connection. Data transmission can begin at this point.

In practice, the next step would be to establish a secure connection by engaging in a TLS handshake as discussed in detail in Section 2.2.2. This adds even more extra round trips to completely establish the connection, which highlights one limitation of TCP - it doesn't have built in security.

2.3.1.3 Reliability & Flow Control

Flow control is managed through a technique known as the sliding window protocol. This method enables the sender to dispatch a series of segments sequentially without awaiting individual confirmations from the receiver, optimising the throughput of the transmission.

The concept of a “window” in this context refers to the count of segments allowed to be in transit at once; the sender can issue up to this number of segments and must then wait for an acknowledgment (ACK) from the receiver before proceeding with more.

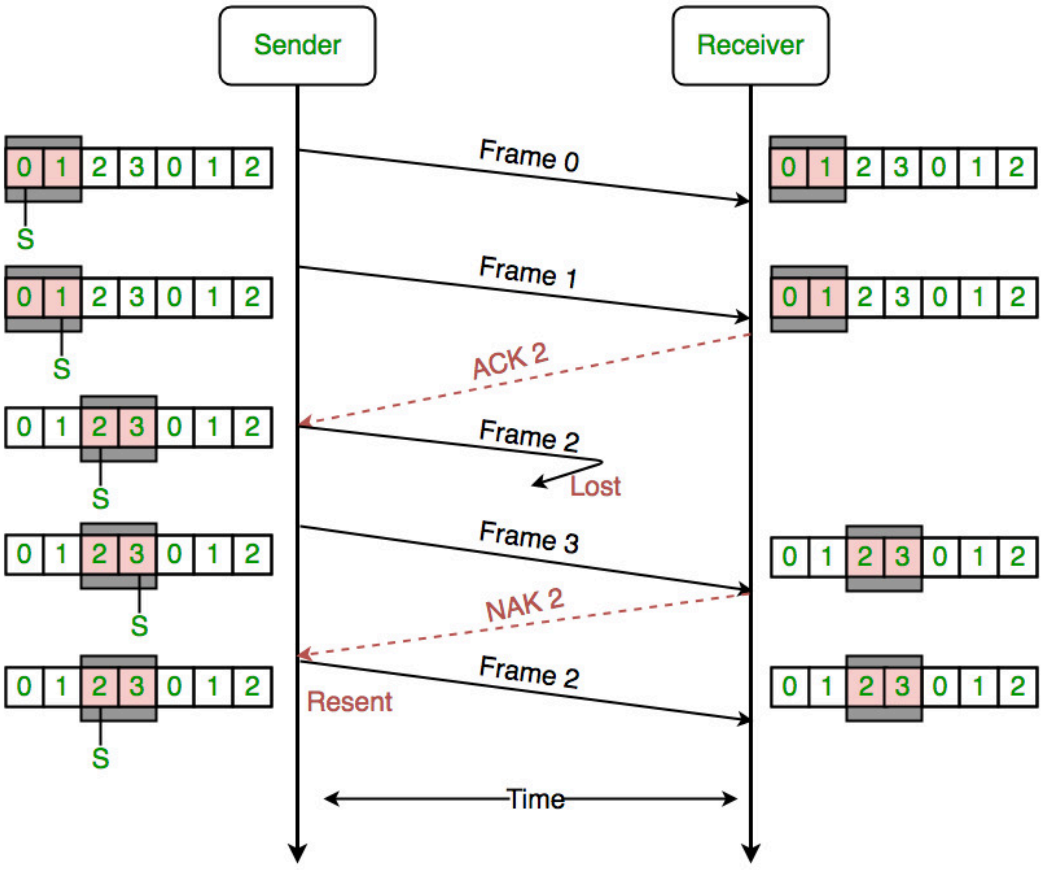

Figure 2.12 Flow of Data within a TCP Connection Utilising the Sliding Window Protocol - Multiple frames can be issued without pausing for individual acknowledgments. The window adjusts with each acknowledgment, sliding forward to permit the continuous, efficient transmission of data packets and the management of packet loss through selective acknowledgments and necessary retransmissions.

Acknowledgments from the receiver not only confirm receipt of data but also communicate the receiver’s readiness to accept additional segments by advertising the available buffer space—its window size. This informs the sender of how many more segments it can send before the receiver’s buffer is full. When segments are received in order, the receiver’s window slides forward, allowing for the continuous flow of the communication stream. In events where segments are received out of order or go missing, indicated by a non-acknowledgment (NAK), the receiver can request retransmission of the specific segments, allowing for recovery without needing to resend all subsequent data. This selective acknowledgment process further refines the efficiency of the protocol.

The sliding window protocol thus balances the efficiency of network throughput with the reliability of data transmission, as discussed in Jacobson (1988).

2.3.1.4 Congestion Control

Congestion control is a vital feature of TCP that ensures efficient and fair use of network resources, aiming to prevent packet loss and network congestion. To manage congestion effectively, TCP manages an internal congestion window (cwnd) size, which represents the number of bytes that can be in transit at any given time.

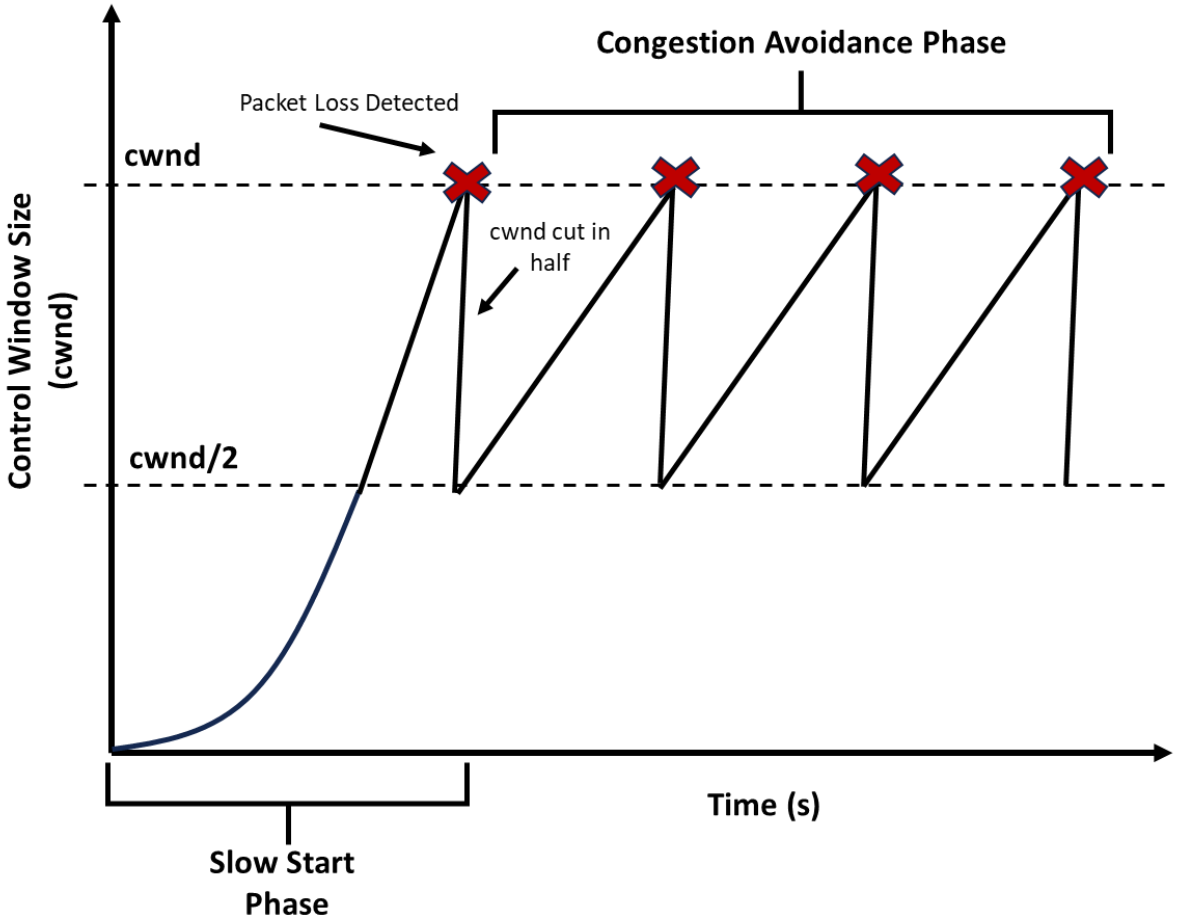

Figure 2.13 TCP Congestion Window Adjustment Phases- The congestion window (cwnd) initially grows exponentially during the "Slow Start" phase, to quickly discover the available network capacity. Upon detecting packet loss, a shift to "Congestion Avoidance" commences, incrementing cwnd additively to probe for bandwidth. Packet loss triggers halving the cwnd, leading to a "saw-tooth" pattern of growth and reduction

Initially, connections begin with a small cwnd, but increase its size exponentially until network congestion is detected. At this point, the cwnd is cut in half, and TCP begins to additively increase it. When network congestion is once again detected, the "Additive Increase Multiplicative Decrease" algorithm (AIMD) repeats, resulting in saw-tooth network activity. This is because TCP enters a congestion avoidance phase after the slow-start phase, as detailed by Jacobson (1988). This congestion control mechanism allows TCP to regulate its data rate and adapt dynamically to changing network conditions, to maintain a stable and reliable network connection between the endpoints.

2.3.1.5 Connection Termination

To gracefully close a TCP connection, the termination process involves several steps to ensure that all remaining data is transmitted, acknowledged, and that both ends of the connection release their resources appropriately.

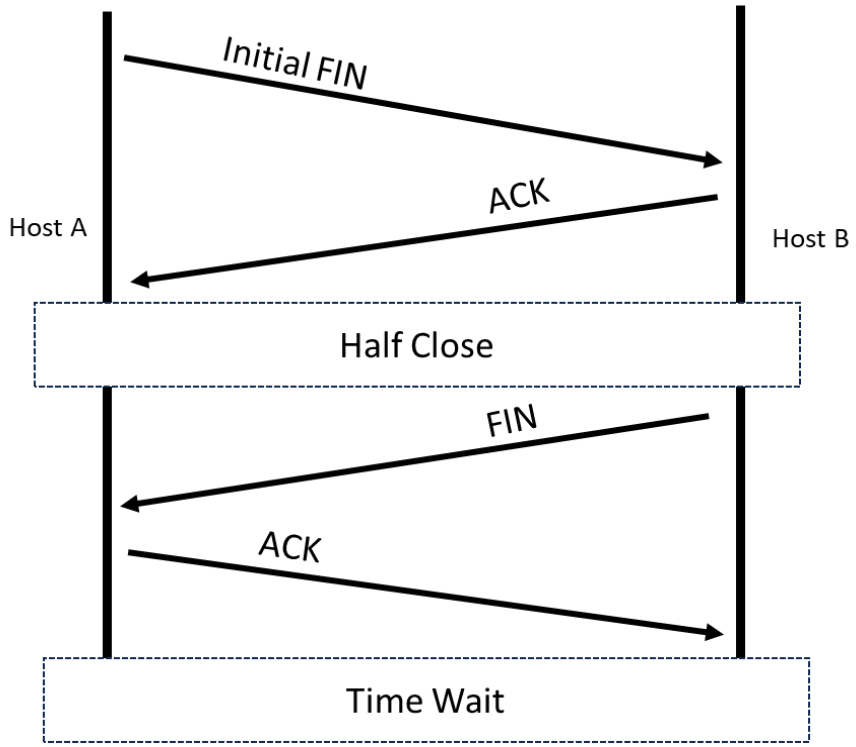

Figure 2.14 TCP Connection Termination Process - Host A sends the "Initial FIN", signalling the intent to terminate the data transfer. The connection proceeds to the "Half Close" state where Host B can still send data, and culminates in a "Time Wait" to ensure all packets have been properly acknowledged, leading to a clean disconnection.

- **Initial FIN:** It begins with one party sending a TCP segment with the FIN finish flag set to signal the end of data transmission.

- **ACK:** Upon receiving this segment, the other party sends an ACK segment, confirming receipt and permitting the sender to close its side of the connection.

- **Half Close:** Both sides then enter a "half-close" state, during which they can still receive already sent data, but cannot send any more. After completing data exchanges, the second party sends a FIN segment to close its side of the connection.

- **Time Wait:** Following this exchange, both sides enter a "time wait" state for a set duration to prevent misinterpretation of stray packets as part of a new connection. Once the time-wait period elapses, resources associated with the connection are released.

This systematic process ensures a graceful conclusion to the TCP connection, facilitating efficient communication termination and resource management.

2.3.1.6 Evaluation of TCP

In summary, TCP is a crucial Transport Layer protocol, which ensures reliable communication between hosts. It establishes connections through a three-way handshake, employing sequence numbers and acknowledgments for reliable data transfer. TCP segments contain essential header information and payload, with a structure facilitating efficient transmission. Flow control mechanisms, such as sliding windows, optimise data exchange, while congestion control algorithms regulate data rates to prevent network congestion. Finally, the TCP connection termination process involves exchanging FIN and ACK segments to gracefully close the connection, followed by a time-wait state to prevent misinterpretation of stray packets.

With a comprehensive understanding of TCP, we can set out to evaluate some of its limitations. One significant drawback is its inherent reliance on congestion control mechanisms, which can lead to degraded performance in scenarios with high latency or packet loss. TCP's slow start and congestion avoidance algorithms can result in inefficient use of network resources and suboptimal throughput, particularly in high-speed or wireless networks.

TCP's connection-oriented nature introduces overhead due to the initial handshake process, which includes multiple round-trip exchanges before data transmission begins. This overhead can be detrimental to applications requiring low-latency communication, such as real-time multimedia streaming or online gaming.

TCP's strict ordering of packets can also lead to head-of-line blocking issues, where the delivery of out-of-order packets is delayed until preceding packets are successfully received and processed. This can impact the responsiveness of applications, particularly those that prioritise timely delivery of individual packets over strict ordering, such as voice and video communication.

It is also important to note that "middle boxes", like NATs and firewalls expect a certain format when processing transport layer protocols at the kernel level (Wang et al., 2013). This means that TCP is limited in its ability to change, as it cannot significantly change its segment format.

Overall, while TCP offers reliability and error recovery features, its congestion control mechanisms, handshake overhead, and strict ordering requirements may pose challenges in certain network environments and for latency-sensitive applications.

In the next section, we will have a look at how other Transport Layer protocols have been developed with these specific limitations in mind.

2.3.2 User Datagram Protocol (UDP)

Unlike TCP, UDP (Postel, 1980) is a connectionless protocol, as it does not involve setting up a connection between two hosts at the transport layer. It also does not guarantee reliable, ordered delivery of data. Instead, its focus lies in speed and simplicity, making it suitable for real-time applications such as live-streaming.

2.3.2.1 UDP Datagrams

UDP's simplicity is reflected in its datagram structure. As there is no guarantee of reliable or ordered delivery of data, many of the fields seen in TCP, such as the SN or the ACK fields, are simply not needed in a UDP datagram.

A UDP datagram is made up of the simple UDP header, appended with the actual payload.

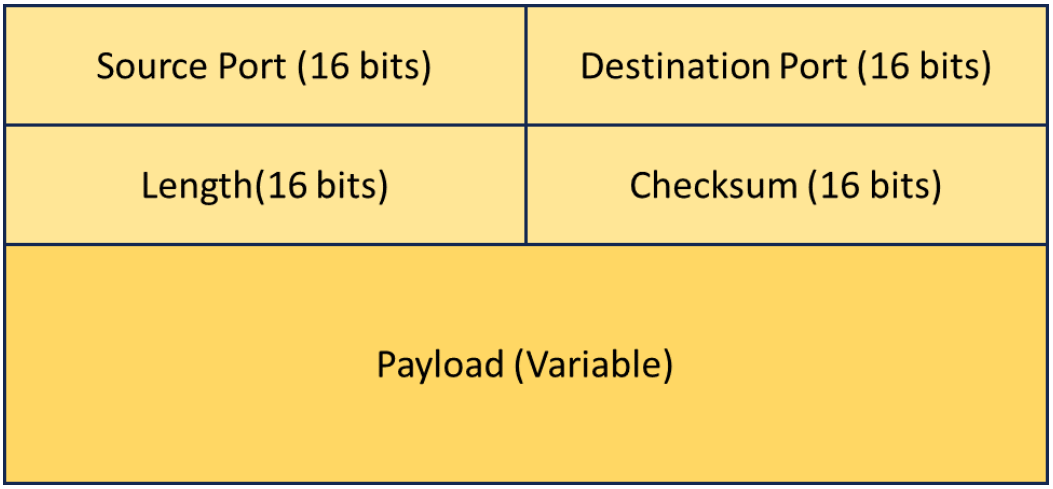

Figure 2.15 Structure of a UDP Datagram - the header required to send a payload is much more concise than the TCP equivalent, but this also reduces the protocols capabilities.

- **Source & Destination Port (16 bits each)**
- **Length (16 bits)**: This indicates the total length of the UDP datagram (header + payload). As the receiving host application should be aware of the fixed sizes of the header fields, it can use this to infer the size of the data.
- **Checksum (16 bits):** checksum value, used to detect transmission errors
- **Data Payload (Variable)**: carries the application-specific data being transmitted

2.3.2.2 Connectionless Communication

The concept of connectionless communication refers to a mode of data transmission where no prior setup or coordination is required between the communicating parties before data transfer occurs. This also means that there is reduced network overhead, as there is no session establishment, maintenance and teardown.

UDP itself is a connectionless protocol, but the application layer protocol it is carrying can still be implemented in such a way that it adopts connected communication. When an application wishes to communicate with another host via UDP, it adopts a "best effort" delivery approach, simply sending the data to the receiver, but the application-layer protocol itself may choose to implement some sort of datagram ordering and acknowledgement.

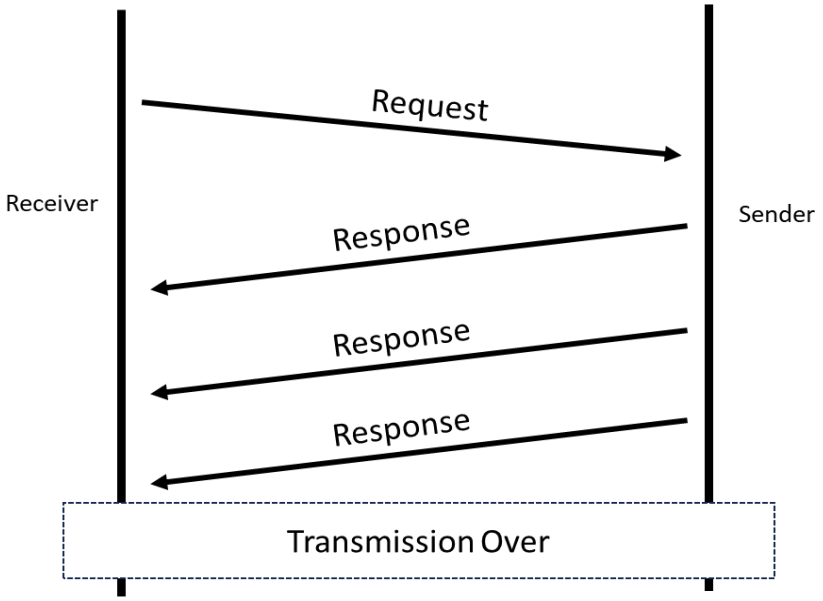

Figure 2.16 Connectionless Communication - Setting up or tearing down connections is not required, transmission simply occurs on request and then simply ends when the data is sent.

As no connection is established, when data is no longer being sent, the sender simply stops, and there is no need for a connection teardown.

2.3.2.3 Evaluation of UDP

UDP's connectionless nature eliminates the overhead associated with connection setup, maintenance, and teardown, making it well-suited for real-time applications where speed and efficiency are paramount. This aspect allows for minimal latency and high throughput.

UDP's minimalist header structure and lack of congestion control mechanisms contribute to its efficiency and simplicity. While TCP provides reliability through features like acknowledgment and retransmission, UDP prioritises speed over reliability, making it ideal for applications where occasional packet loss is acceptable.

UDP's error detection mechanism, represented by the checksum field, offers basic protection against transmission errors, so datagrams can be discarded. However, UDP does not include error recovery mechanisms, such as packet retransmission, which means that applications must handle error detection and recovery at the application level if necessary. This lack of built-in error recovery may be a limitation for applications requiring high reliability.

2.3.3 QUIC

QUIC is a modern transport layer network protocol designed by Google and was standardised by the IETF in RFC 9000 (Iyengar & Thomson, 2021). It aims to improve the performance of internet communications by reducing connection establishment time, providing encryption, and eliminating head-of-line blocking at the transport layer.

2.3.3.1 Built On UDP

Built on top of UDP, QUIC integrates key features of TCP, TLS, and the concept of independent streams, providing reliable and multiplexed connections. As a connectionless protocol, UDP also allows QUIC to have its own implementations for connection management, reliability, and flow control. The QUIC frames are encapsulated by QUIC packets, and then the packets themselves are encapsulated by UDP datagrams. Multiple packets can be chained together into one UDP datagram. By using UDP, QUIC is able to be deployed on existing Internet infrastructure, without requiring changes to network hardware or software that operate at the Internet layer and below, ie, the "middle boxes", which are used to working with TCP and UDP.

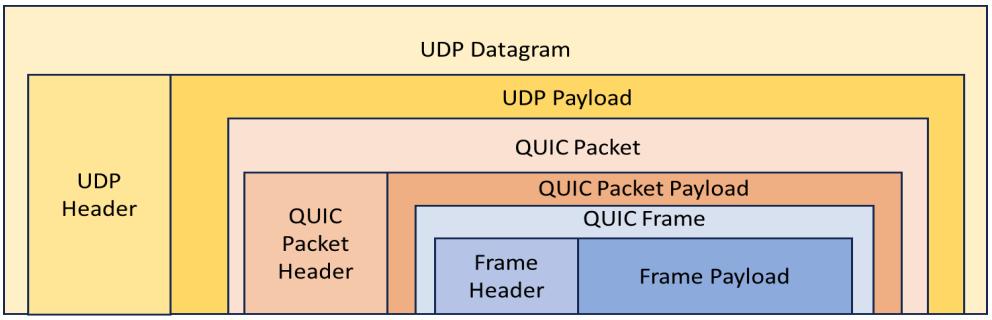

Figure 2.17 Encapsulation of QUIC In UDP - A UDP datagram consisting of the UDP header and the payload containing the actual QUIC packet.

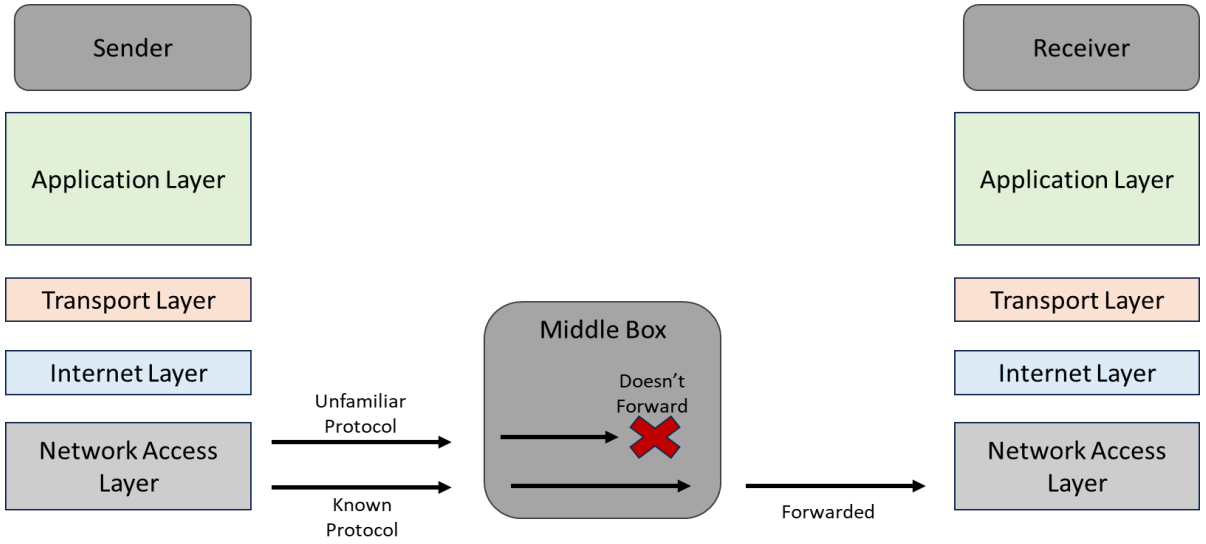

Figure 2.18 Middlebox Protocol Filtering - network middlebox discriminates between familiar and unfamiliar protocols, allowing traffic following known protocols to pass through while ignoring unrecognised ones (Wang et al., 2013).

2.3.3.2 Frame Types

QUIC frames are the fundamental building blocks of QUIC packets. There are many frame types used in QUIC, but they generally fall into one of five categories:

Connection Management Frames

These frame types handle the setup, maintenance, and closure of a QUIC connection.

- *HANDSHAKE_DONE*: Signals the completion of the handshake process.
- *NEW_TOKEN*: Provides a token for future connections, enhancing connection resumption and address validation.
- *PATH_CHALLENGE* and *PATH_RESPONSE*: Used in path validation to ensure that the endpoint can receive packets on a new network path.
- *CONNECTION_CLOSE*: Used to signal the termination of a connection due to errors or normal closure.

Stream Management Frames

Frame types related to the creation, use, and termination of streams, facilitating the multiplexed communication QUIC supports.

- *STREAM:* Carries application data, with control flags for managing the end of the stream and its flow, and a stream id.
- *RESET_STREAM:* Abruptly terminates a stream, typically due to an error
- *STOP_SENDING:* Requests cessation of data sending on a specific stream, typically due to application layer decisions.

Flow and Congestion Control Frames

These frame types manage the flow of data across the entire connection and individual streams, preventing congestion and ensuring efficient use of available bandwidth.

- *MAX_DATA*: Sets the total data transfer limit across all streams.
- *MAX_STREAM_DATA*: Sets the data transfer limit for an individual stream.
- *DATA_BLOCKED*, *STREAM_DATA_BLOCKED*, *STREAMS_BLOCKED*: Signal that the sender has hit a data limit and is blocked until the limit is raised.
- *MAX_STREAMS* (Bidirectional and Unidirectional): Controls the maximum number of concurrent streams.
- *ACK*: Provides receipt acknowledgments for packets, for reliable data transmission and congestion control.

Security and Cryptography Frames

Frame types that are involved in the cryptographic negotiation and security of the QUIC connection.

- *CRYPTO*: Transports cryptographic handshake messages that are necessary for establishing a secure connection.

5. Miscellaneous Frames

- *PADDING*: Used to increase packet size, to reach the minimum size or for protection against unwanted analysis.
- *PING*: Can be used to keep a connection alive or to measure round-trip time, aiding in congestion control and flow management.

2.3.3.3 Packet Types

There are two categories of packet types defined in RFC-9000: packets using a "long header" and packets using a "short header". There is also the special Stateless Reset packet, which is used as a last resort for an endpoint that is not in the connection state.

2.3.3.3.1 Long Header Packets

Long header packets are used primarily during the initial setup of a QUIC connection.

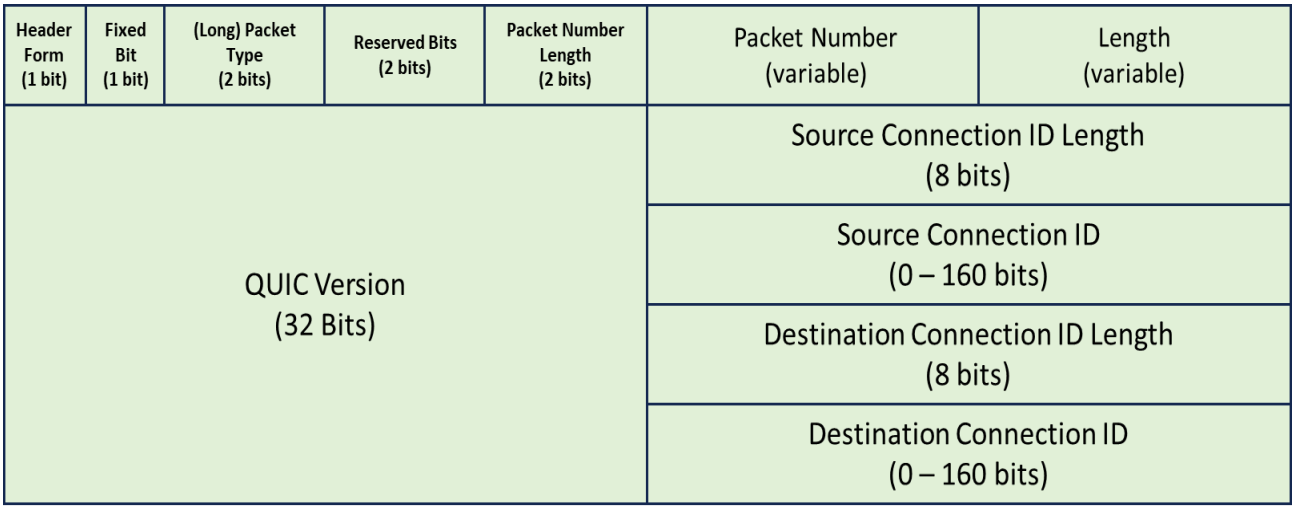

Figure 2.19 QUIC Long Header Packet Structure - this header facilitates the initial handshake and version negotiation processes in establishing a QUIC connection.

They have the following header format:

- **Header Form:** Value is always set, indicating the header is a Long Header
- **Fixed Bit:** always set to identify QUIC, aiding protocol multiplexing per RFC 7983
- **Long Packet Type:** Value is between 0-3 and indicates packet type
- **Reserved Bits**
- **Packet Number Length**
- **Packet Number:** a unique Packet Number, necessary for acknowledgements
- **Length:** represents the length of the remainder of the packet
- **QUIC Version:** Indicates the QUIC version, needed for version negotiation
- **Source Connection ID Length**
- **Source Connection ID:** The connection ID associated with the source
- **Destination Connection ID Length**
- **Destination Connection ID:** The connection ID associated with the destination

1. Initial Packet

Used for the initial handshake process, carrying the first CRYPTO frames sent by the client and server to negotiate a connection.

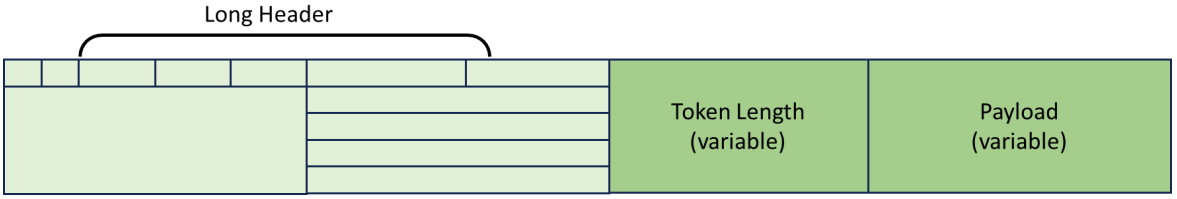

Figure 2.20 QUIC Initial Packet Format

Format:

- Long Header
- Token Length: indicates the value of the potentially included token
- (optional) Token: the token associated with a retry or stateless reset
- Payload: Usually *CRYPTO* and *ACK* frames

2. 0-RTT Packet

Optionally sent by the client after an Initial packet to carry early data with the first flight.

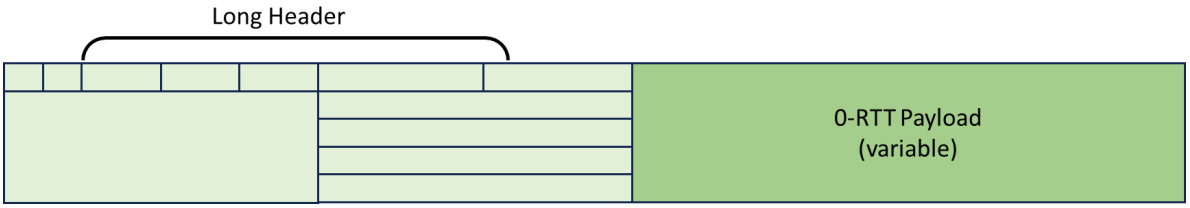

Figure 2.21 QUIC 0-RTT Packet Format

Format:

- Long Header
- Payload (variable): Usually *STREAM* frames with application data

3. Handshake Packet

Used by both clients and servers after the Initial packet for cryptographic handshake messages not suitable for 0-RTT.

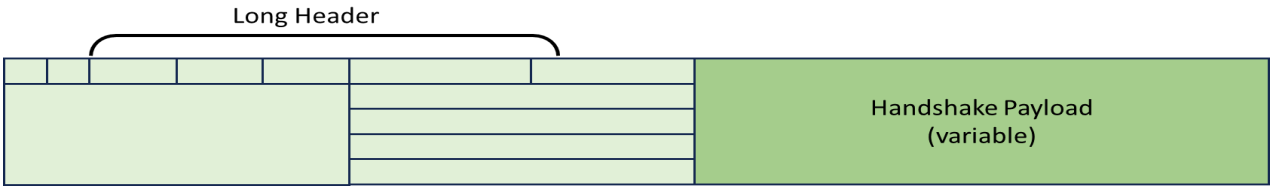

Figure 2.22 QUIC Handshake Packet Format

Format:

- Long Header
- Handshake Payload (variable): Usually *CRYPTO* frames, *PING, PADDING, ACK,* and *CONNECTION_CLOSE* frames are also permitted

4. Retry Packet

Sent by the server in response to an Initial packet from a client, requiring the client to echo back a server-provided token within a new Initial packet. The Retry mechanism helps ensure that the client is not merely spoofing its IP address. Additionally, the integrity protection provided by the Retry Integrity Tag ensures that the Retry mechanism cannot be exploited by attackers to disrupt legitimate connection attempts. The packet does not set the Length or Packet Number fields, and is not protected by encryption.

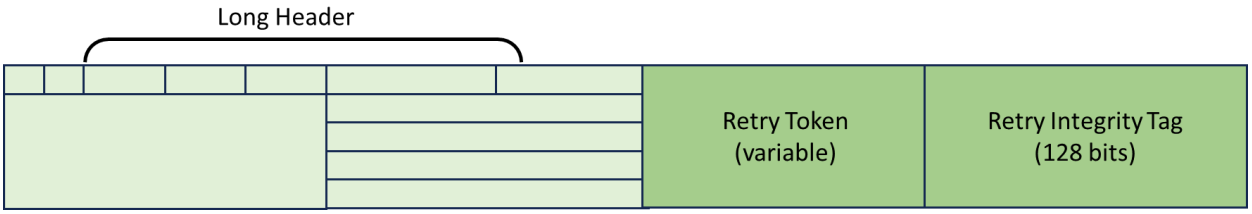

Figure 2.23 QUIC Retry Packet Format

Format:

- Long Header
- Retry Token (variable): value to be included in the next Initial Packet
- Retry Integrity Tag (128): Used by the client to verify the integrity of the Retry packet and ensure that it was sent by the server

2.3.3.3.2 1-RTT Short Header Packet

There is only one packet type that is defined under the short header, and that is the 1-RTT packet. This packet type is used after the initial connection establishment process, for the sending of application data.

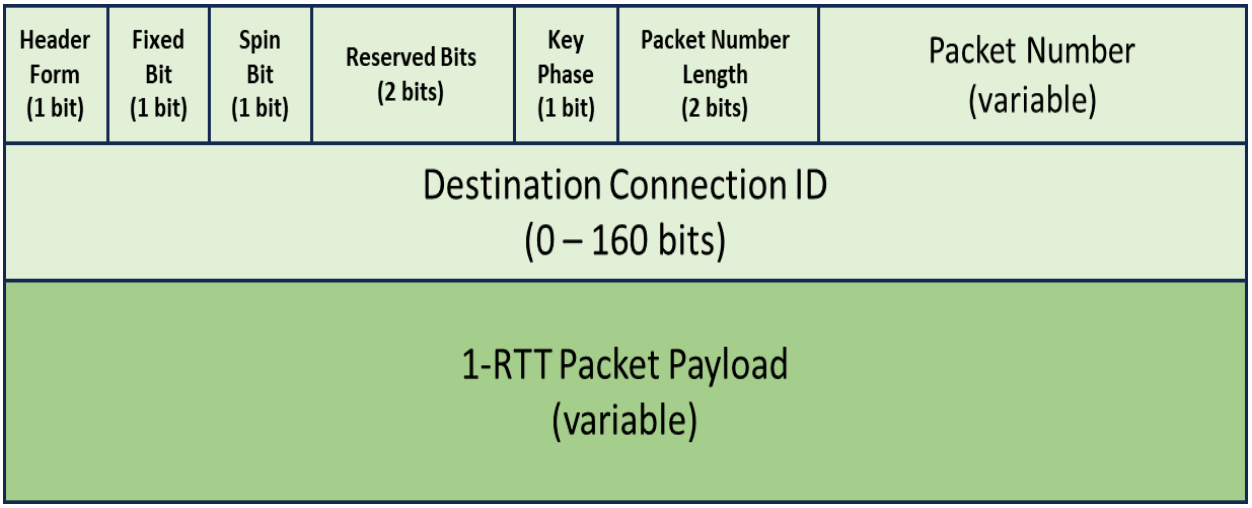

Figure 2.24 QUIC 1-RTT Short Header Packet Format - using a shorter header allows for reduced header overhead when sending application data in the payload, as opposed to TCP, which always uses a large packet header.

The 1-RTT packet is made up of the following fields:

- **Header Form (1 bit):** Value is not set, indicating a short header is used
- **Fixed Bit (1) bit:** Always set to identify QUIC, aiding protocol multiplexing per RFC 7983
- **Spin Bit (1 bit):** "Spins" values with every round trip, used for measuring end-to-end latency on a per-connection basis without having to decrypt packets
- **Reserved Bits (2 bits)**
- **Key Phase (1 bit):** Indicates which set of keys is used to protect the packet's payload, allowing the protocol to signal key updates.
- **Packet Number Length (2 bit)**
- **Packet Number (8-32 bits)**
- **Destination Connection ID (0-160 bits):** Once connection is established, there is no need to specify the CID length, as it is determined during the connection establishment phase
- **Packet Payload (variable):** Frames carrying app data and/or control

2.3.3.3.3 Stateless Reset Packets

This packet type is used by an endpoint to signal the abrupt closure of a connection when the endpoint has lost state, typically due to a crash or restart. It mimics a short header packet because it doesn't include actual extensive header information, so it sets the "Header Form" field to "0".

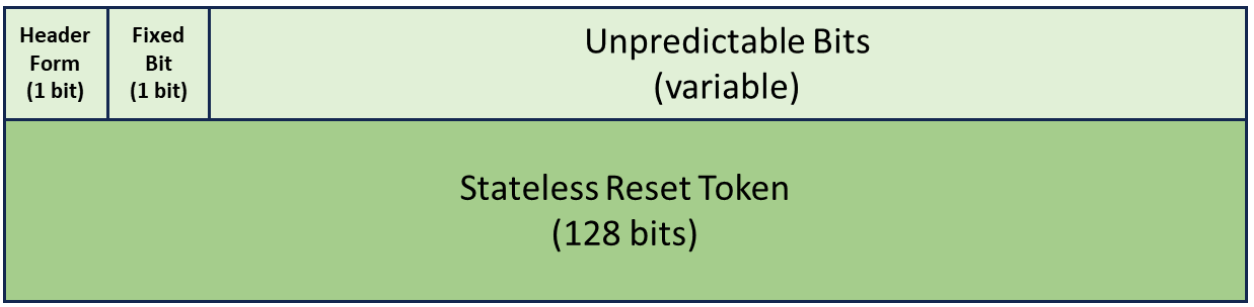

Figure 2.25 QUIC Stateless Reset Packet Format - It resembles a regular 1-RTT packet, so that network observers cannot easily distinguish between stateless reset packets from other packets

Format:

- **Header Form (1 bit):** Value is set to "0", as it mimics a short header.
- **Fixed Bit (1 bit):** Always set to identify QUIC, aiding protocol multiplexing per RFC 7983
- **Unpredictable Bits (min 38 bits)**: Random bits to make the packet appear like a regular packet to an observer, adding a layer of privacy to the connection.
- **Stateless Reset Token (128 bits):** A token which was established during the connection phase by the server. Clients store that token securely, and upon receiving a packet that ends with that token, know that the packet is a stateless reset packet.

2.3.3.4 Connection Establishment

QUIC integrates the TLS 1.3 handshake in its connection establishment, to ensure encrypted communication, leveraging the ability to chain together multiple packets within a single UDP datagram (Thomson & Turner, 2021). This method not only accelerates the handshake process but also enhances security, making it ideal for modern internet applications where speed and security are paramount.

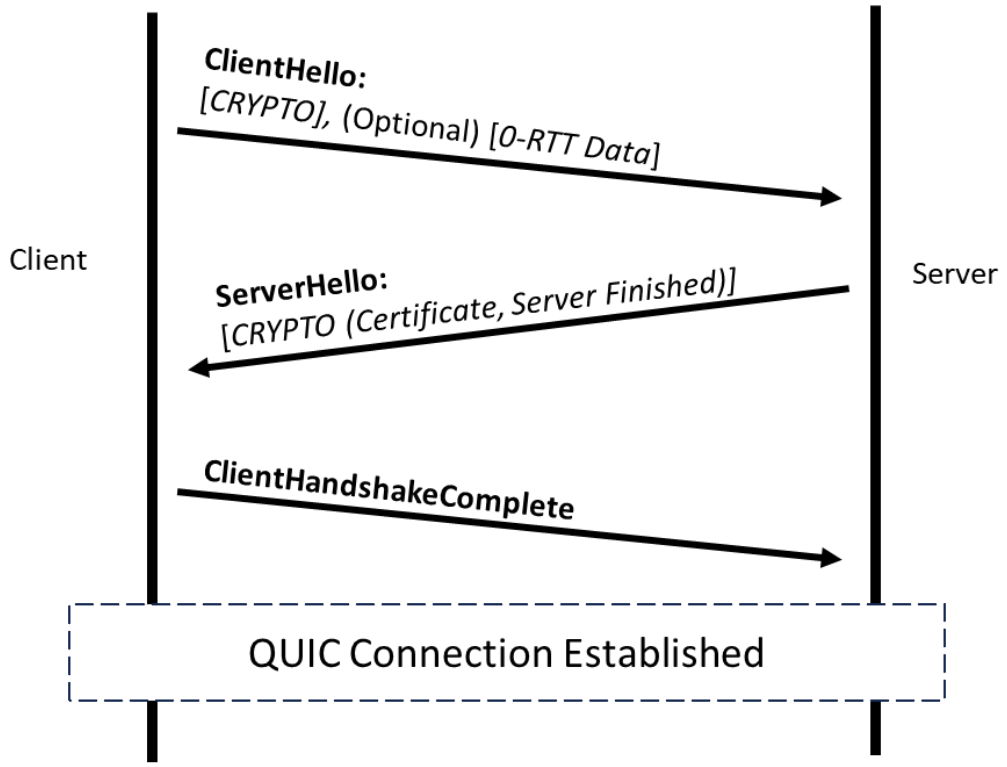

Figure 2.26 QUIC Connection Handshake - QUIC streamlines the connection process by integrating the TLS1.3 handshake. All the information required is packed into one packet sent by each side.

ClientHello:

- The client initiates the connection by sending an Initial packet, which contains a *CRYPTO* frame with a TLS 1.3 Client Hello message. This frame proposes cryptographic parameters, supported cipher suites, and might include the session resumption token needed for 0-RTT data.

0-RTT Data (Optional):

- If applicable, the client sends 0-RTT packets immediately after the Initial packet, carrying application data in *STREAM* frames.
- This data is encrypted with keys derived from a previous session with the server, and if the server still recognises the keys, will accept the data, enabling immediate data transmission

ServerHello:

- The server responds with its own Initial packet containing a *CRYPTO* frame with a TLS 1.3 Server Hello message, which selects cryptographic parameters and provides the server's connection ID.
- This packet also includes *CRYPTO* frames carrying the server's certificate, CertificateVerify, and ServerFinished messages, completing its part of the TLS handshake.

ClientHandshakeComplete:

- The client sends a series of Handshake packets that include *CRYPTO* frames with the ClientFinished message, encrypted with temporary handshake keys.
- This step completes the TLS handshake, transitioning to encrypted communication.

QUIC Connection Established:

- The secure QUIC connection is now established, allowing for fully encrypted bidirectional communication.
- Both parties can exchange 1-RTT packets containing application data and control messages

2.3.3.5 Connection Resumption

QUIC streamlines subsequent connections to a server after the initial handshake has been successfully completed, by leveraging "next session tokens", which the server would have established during the last connection.

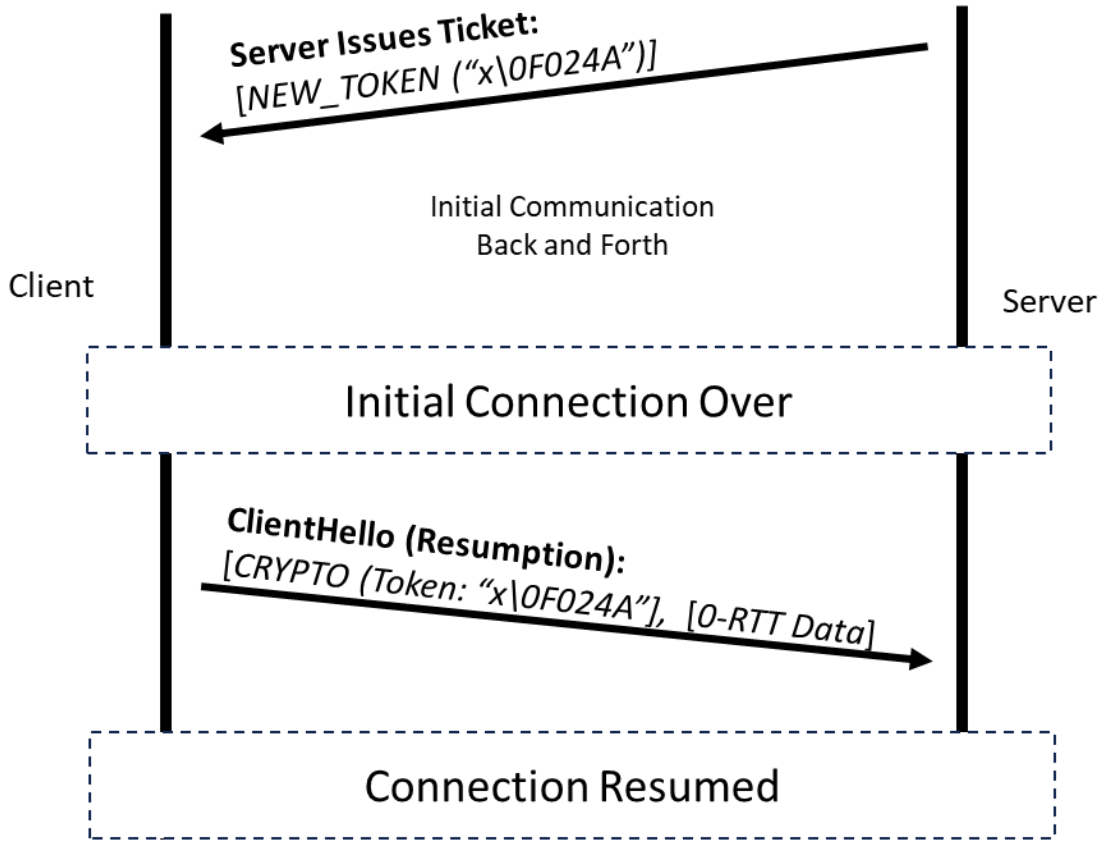

Figure 2.27 0-RTT Connection Resumption Process - When reconnecting after a previous session, the client uses a server issued session ticket within a "ClientHello" message. This facilitates immediate data transfer with 0-RTT data packets, and streamlines the handshake for subsequent connections by reusing cryptographic parameters.

Session Issues Ticket (During Initial Connection):

- **Server's Role:** At the end of a successful initial connection, the server generates a session ticket that includes the necessary information for resuming the session. This is sent to the client in a *NEW_TOKEN* frame and also includes cryptographic parameters, application state, and other session-specific data. The client stores this session ticket for future use.

- Eventually the session ends, but the client has a session resumption ticket now.

Client Initiates Connection Resumption:

- **Using the Session Ticket**: When the client wishes to reconnect to the server, it initiates a new QUIC connection. As part of its Initial packet, the client includes a *CRYPTO* frame carrying a TLS 1.3 Client Hello message that references the session resumption token from the session ticket.

- **0-RTT Data**: Alongside the Initial packet, the client can immediately send 0-RTT packets containing *STREAM* frames with application data, encrypted using 0-RTT keys derived from the previous session's cryptographic parameters.

Server Processes Resumption Request:

- Token Validation: If the token is valid, the server accepts the 0-RTT data. It then proceeds to use the previously established cryptographic parameters to decrypt and process this early data. The server also generates a new session ticket for future resumptions.

Handshake Completion:

- The connection is resumed, both parties switch to using 1-RTT keys again for subsequent communication, with STREAM frames carrying application data fully encrypted under the new keys.

2.3.3.6 Connection Migration

Once a connection is established, every packet sent contains a Connection ID (CID). As opposed to TCP, which identifies connections based on their IP and port number, QUIC uses this CID to identify the connection. Because of this, "connection migration" can be implemented on QUIC servers and clients. This is when an active connection can continue with minimal interruption, even when one of the endpoints changes its IP address or port, making the connection independent of the underlying network.

However, one issue is that the CID is not an encrypted field of the QUIC packet, and it is visible to observers. This on its own is not a major issue, as not much can be done with just the CID, but it could permit tracking of connections when switching between networks. This is why alternative, "backup" CID's are established over the course of a connection, using an encrypted

NEW_CONNECTION_ID frame. When a client switches, they simply use one of these, secret CID's, and only the server and client know what happened, preventing tracking.

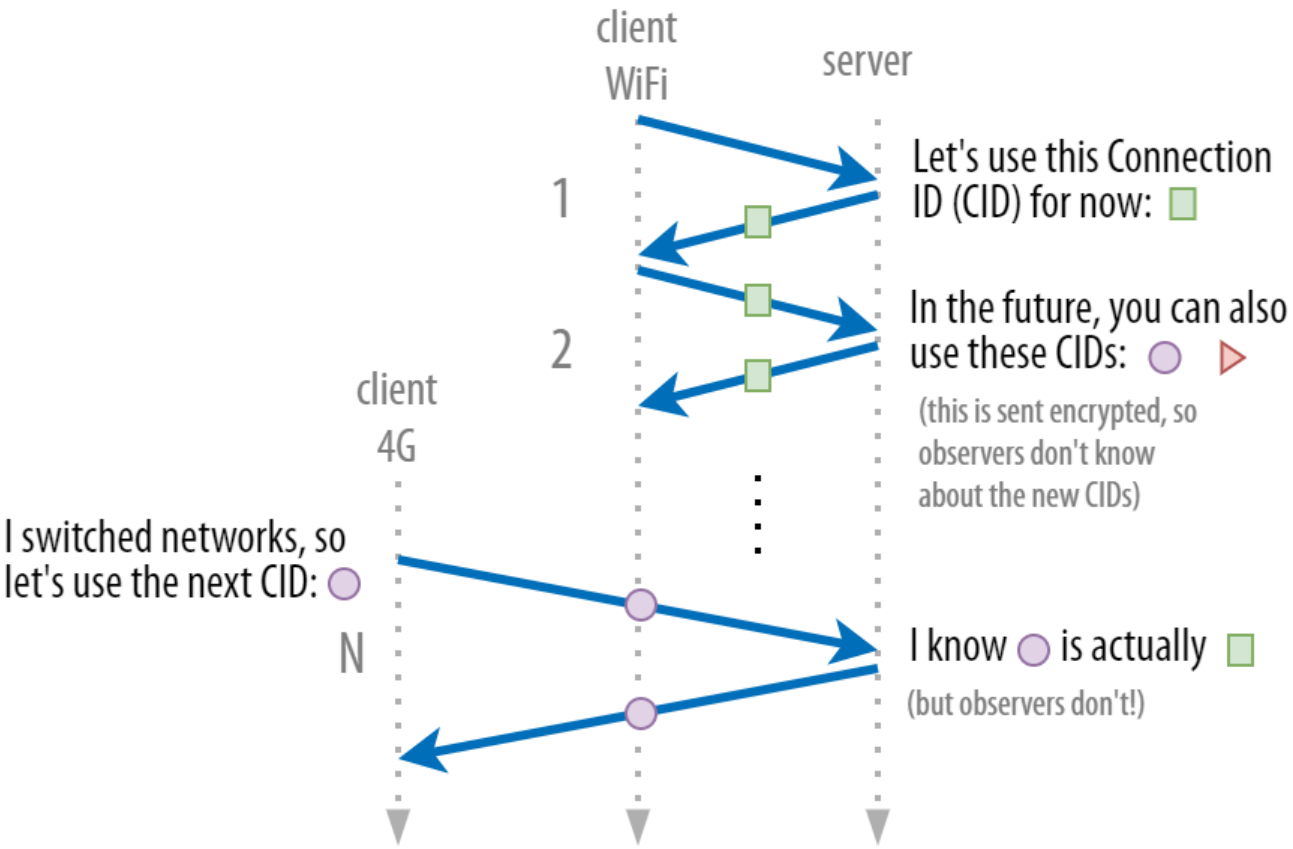

Figure 2.28 QIUC Connection Migration (Marx, 2023) - *The underlying network can change without interrupting the ongoing communication, as the server recognizes the client by the CID's established, rather than IP address, ensuring a persistent connection despite changes in the underlying network.*

Connection migration is particularly useful on mobile networks and for a microservices architecture (Puliafito & Conforti, 2022), as users move around and are constantly changing their IP addresses. By maintaining the same CID, when it sends a request to a server it already had an ongoing connection to, despite changing IP addresses and port, the server recognises the CID, and the connection is not ended.

2.3.3.7 Reliability & Flow Control

As outlined in RFC-900 (Iyengar & Thomson, 2021) and Dellaverson et al. (2022), QUIC employs an ACK mechanism for confirming data receipt, which is crucial for maintaining transmission reliability. It also integrates flow control features tailored to both the connection overall and to the discrete streams within that connection. These strategies allow for precise regulation of data flow, which is critical in preventing congestion and ensuring the smooth delivery of packets.

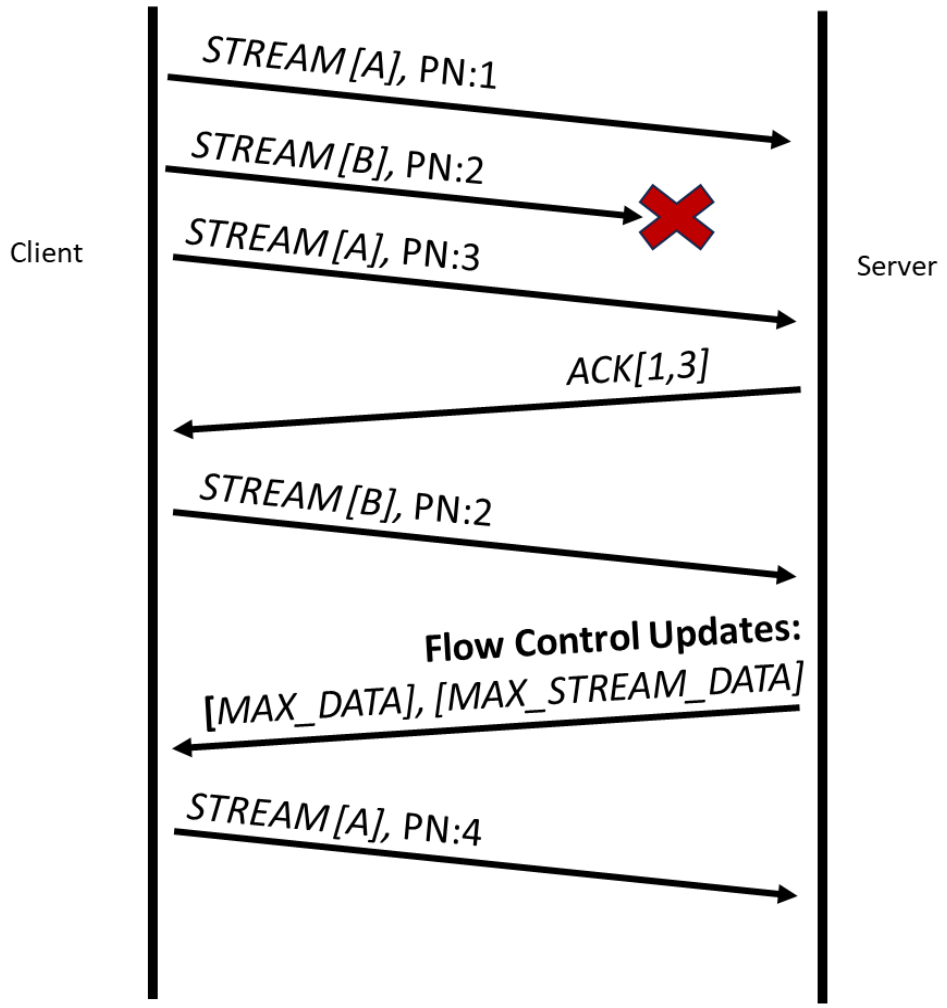

Figure 2.29 QUIC Flow Control - Packets are sent without waiting for individual ACKs. When a cumulative ACK is missing a packet number, it indicates that a certain packet was lost, so it is retransmitted. Flow control updates can be sent by either party, to directly set the size of the transmission window on either the whole connection or on the individual stream level.

Data Transmission:

- The client sends out packets with *STREAM* frames containing data for Stream A and B
- Packet 2, containing Stream B data, gets lost because over server stream buffering issues

ACKs for Reliability:

- The server sends an ACK frame back to the client, acknowledging the receipt of Packets 1 and 3.
- The lack of acknowledgment for Packet 2 signals the client to consider this packet lost.

Retransmission:

- The client retransmits the lost data for Stream B in Packet 4.

Flow Control Updates:

- The server might choose to send flow control updates, using *MAX_DATA* to update the connection-wide window and *MAX_STREAM_DATA* to update the window for a specific stream, to prevent data loss again.

2.3.3.8 Connection Termination

Once the communication is complete, either the client or the server can initiate the connection closure process. The connection closure process in QUIC is designed to be concise and clearly communicated between endpoints, ensuring that both parties are aware of the connection state and can clean up resources appropriately.

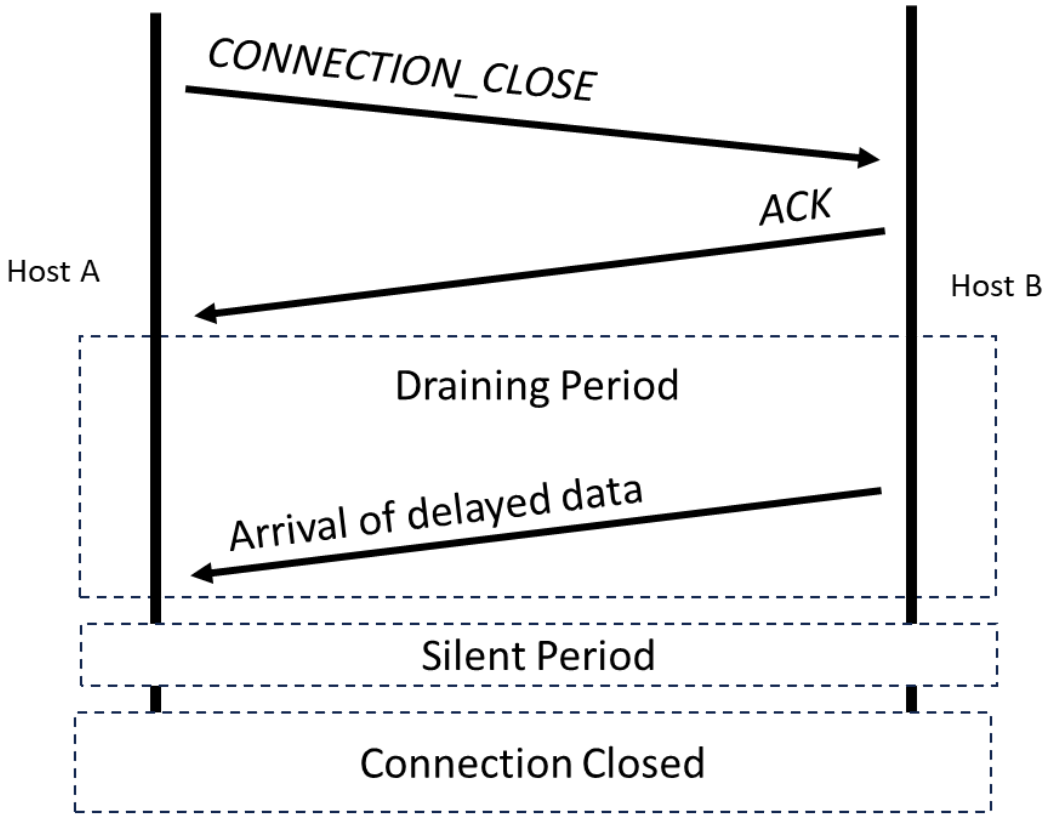

Figure 2.30 QUIC Connection Termination Process - On arrival and acknowledgement of a CONNECTION_CLOSE frame, the connection enters a "Draining Period" which allows data which is already in transit to be received. After a "Silent Period" the connection is officially closed.

Sending CONNECTION_CLOSE Frame:

- The closing endpoint sends a CONNECTION_CLOSE frame within a 1-RTT packet. This frame can indicate a normal connection close or a close due to an error.
- It includes a reason for the closure and any additional information for debugging if it's an abnormal closure.

Receiving End Acknowledges:

- The receiving endpoint processes the CONNECTION_CLOSE frame, acknowledges it, and initiates the closing of the connection on its side.

Draining Period:

- QUIC recommends a "draining period" after a CONNECTION_CLOSE frame is sent. During this period, the closing endpoint should not send any further packets but should remain available to read packets from its peer.

- This is to handle any delayed or retransmitted packets on the network that may still arrive.

Silent Period and Final Closure:

- After the draining period, the endpoint enters a silent period, during which it discards any additional incoming packets and completes the closure of the connection.
- The silent period ensures that both sides have ample time to observe the connection termination, reducing the chances of lingering or spurious transmissions that could otherwise lead to confusion.

2.3.3.9 Evaluation Of QUIC

By addressing the limitations of previous protocols, QUIC introduces a more efficient, secure, and adaptable foundation for internet communication. The use of UDP as a base allows QUIC to implement and add its own layer for reliability, flow control, and connection management, in a manner that surpasses the TCP implementation.

Unlike TCP's three-way handshake, QUIC can establish a secure connection with fewer exchanges, minimising the delay before data transmission begins, crucial for applications demanding low latency such as live video streaming or online gaming. This is through its integration of transport and cryptographic features from TLS1.3 into a single protocol, which not only streamlines the connection by taking advantage of TLS1.3's faster handshake, but also significantly enhances reconnection processes.

A significant advancement in QUIC is its approach to data transmission reliability and flow control. By adopting independent stream multiplexing, QUIC circumvents TCP's head-of-line blocking problem, whereby a single lost packet can delay the delivery of all subsequent packets within the same connection. Each stream in QUIC operates independently, so packet loss in one stream does not block the progress of others. This is particularly beneficial over networks experiencing intermittent losses, typical in mobile or wireless environments.

QUIC also incorporates advanced congestion control mechanisms that are designed to be more reactive to changing network conditions than those typically found in TCP, and can be used to configure congestion settings at the connection level, as well as at the individual stream level. These mechanisms allow QUIC to adjust quickly to bandwidth fluctuations, improving throughput in scenarios with high latency or packet loss. This adaptability makes QUIC well-suited for high-speed networks where TCP's traditional slow-start and congestion avoidance algorithms may underutilise available bandwidth.

In conclusion, QUIC brings substantial improvements in speed, security, and reliability. Its ability to reduce latency, avoid head-of-line blocking, and handle network transitions effectively addresses many of TCP's drawbacks, positioning it as a potent solution for the future of internet communications, particularly in environments where performance and efficiency are critical.

2.4 The Application Layer & HTTP

The HyperText Transfer Protocol (HTTP) is used one layer above the Transport Layer, in the Application Layer, and is a foundational technology of the internet, designed for transferring documents such as HTML. Over the years, HTTP has evolved through several versions, each aiming to improve efficiency, speed, and security of data transfer (Babmberg, 2023). Below is a brief explanation of how each version of HTTP works, to see its evolution.

2.4.1 HTTP/0.9

HTTP/0.9 is exceedingly simple, supporting only "GET" requests. There are no headers in this version of HTTP, and requests are just sent as plain text.

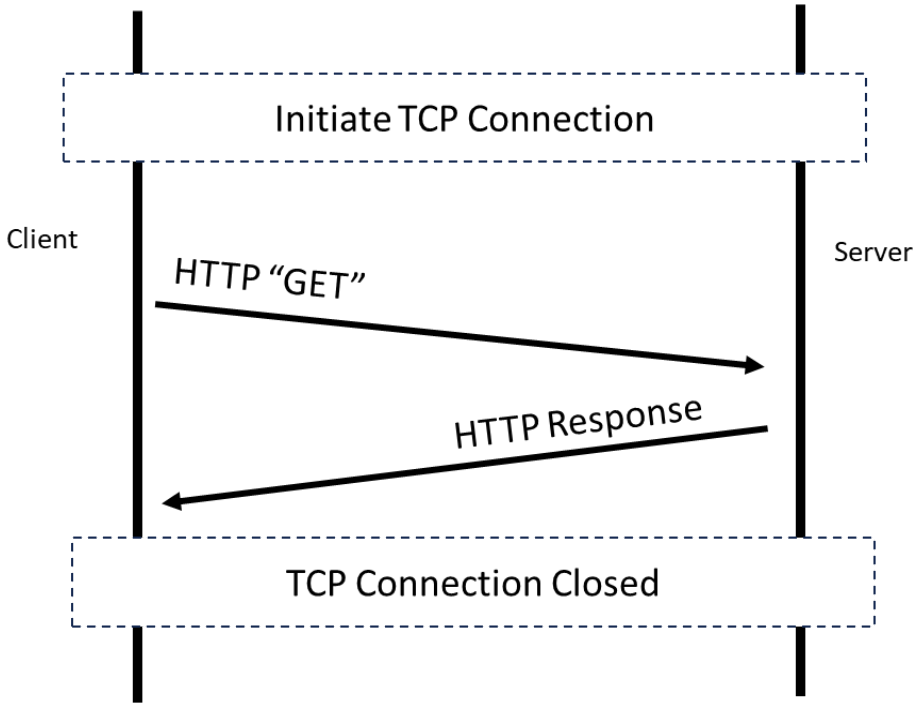

Figure 2.31 HTTP/0.9 Request & Response - a client initiates a TCP connection and issues a simple "GET" request. The server responds with the requested document and terminates the connection, reflecting the protocol's design for basic document retrieval without headers or metadata.

Client Request:

- A client initiates a TCP connection to a server.
- It sends a single line request to get a document.

Server Response:

- The server responds with the content of the document (only HTML supported) and closes the connection, meaning every request had to to initialise a new TCP connection

.

While HTTP/0.9 was introduced to solve the problem of simple document retrieval over the internet, the lack of headers means no metadata or status codes can be communicated effectively, in a standardised manner.

2.4.2 HTTP/1 & HTTP/1.1

HTTP/1 introduced foundational methods such as “PUT” and “DELETE”, and basic headers for content types, status codes, and caching. However, every request needed a new TCP connection. HTTP/1.1 fixed this by introducing support for persistent connections, but performance was hindered due to head-of-line blocking at the HTTP layer, caused by how the pipelining was implemented. HTTP/1.1 expected ordered delivery, meaning that if an earlier request in the pipeline was blocked, all other requests would also be blocked, even if they are ready.

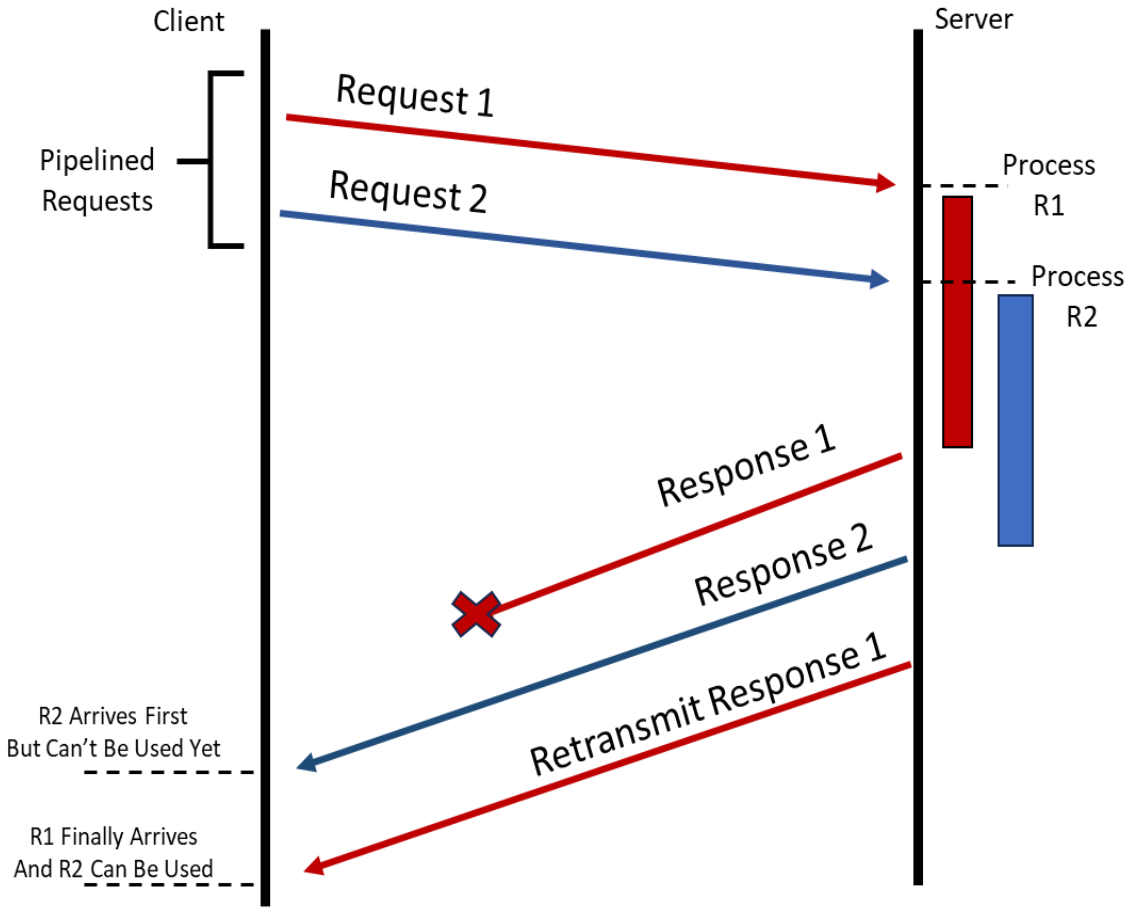

Figure 2.32 Inefficiencies of HTTP/1.1 Pipelining - multiple requests are sent sequentially without waiting for the corresponding responses, but the delay in response to “Request 1” hinders the utilisation of “Response 2”, despite its readiness, underlining the limitations of ordered delivery

One way to overcome this problem and to achieve some form of concurrency was for clients to maintain multiple TCP connections which would execute the requests in parallel. But this obviously increases the network overhead, and more resources are needed to establish and maintain the extra connections.

It was clear that HTTP needed to implement a concurrency model, like request multiplexing, to be able to handle concurrent requests and deal with the head-of-line blocking issues that were manifesting.

2.4.3 HTTP/2

HTTP/2 added header compression to minimise overhead, and tries to solve the head-of-line problem by introducing independent concurrent streams. However, this only fixes the problem at the HTTP/2 layer.

Because of TCP's implementation and its guarantee for in order delivery, the head-of-line problem persisted, as segments that arrive after a segment are not returned until the missing segment is retransmitted.

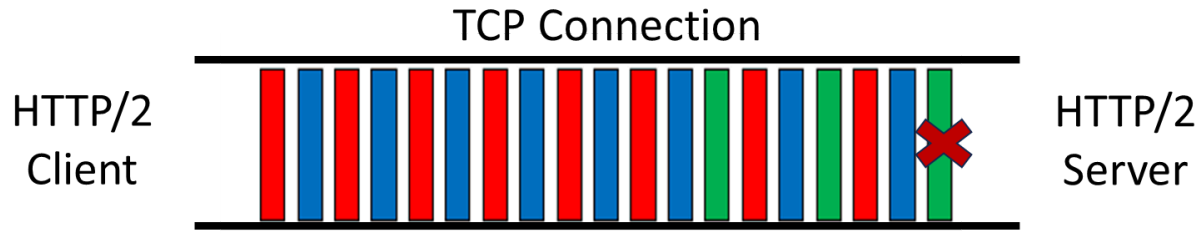

Figure 2.33 HTTP/2 Head-of-Line Blocking - multiplexing independent concurrent streams at the HTTP level is still constrained by TCP's in-order delivery guarantee.

2.4.4 HTTP/3

HTTP/3 represents a significant leap forward in the evolution of the HTTP, primarily due to its switch from TCP to QUIC as its underlying transport layer. This change addresses many of the inefficiencies and limitations previously encountered, offering a more optimised web experience.

Unlike its predecessors, HTTP/3's reliance on QUIC enables a much more efficient handling of connections, effectively sidestepping the notorious head-of-line blocking issue that plagued HTTP/2 by allowing multiple streams of data to be multiplexed over a single connection without waiting for lost packets to be resent.

By having TLS within the QUIC protocol itself, HTTP/3 also ensures that all communications are encrypted by default, providing a stronger security posture against eavesdropping and tampering attacks compared to earlier versions. This built-in encryption is particularly beneficial in today's internet ecosystem, where security concerns are paramount. Additionally, the improved connection migration feature of QUIC, allows a HTTP/3 connection to persist even when the underlying network changes.

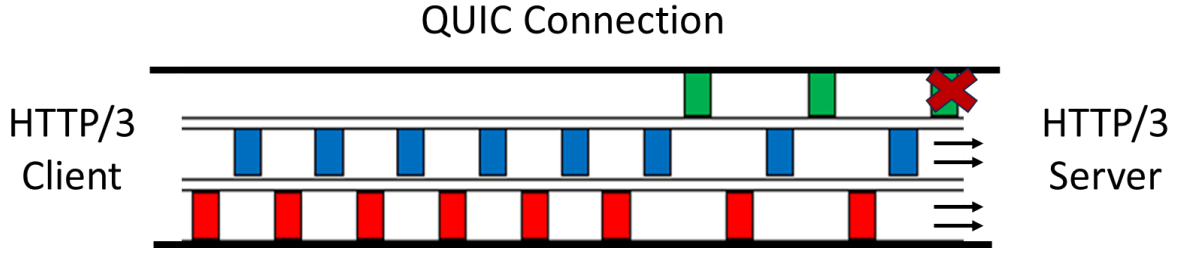

Figure 2.34 HTTP/3 Overcoming Head-of-Line Blocking - QUIC's independent stream management allows the HTTP/3 client and server to continue processing other streams even if one stream is blocked.

2.5 Containerisation

Containerisation is a technology used to encapsulate an application's code, configurations, and dependencies into a single object, or "container". At its core, containerisation relies on the kernel of the host operating system to run multiple isolated user-space instances, known as containers. These containers are a lightweight, efficient form of virtualisation that enables the deployment of distributed applications without launching an entire virtual machine (VM) for each app.

Unlike traditional virtualisation, which requires the use of hypervisors to manage multiple VMs each with their own OS, containerisation involves encapsulating an application in a container with its own operating environment. This method significantly reduces the overhead of running multiple operating systems, leading to better utilisation of system resources (da Silva et al., 2018).

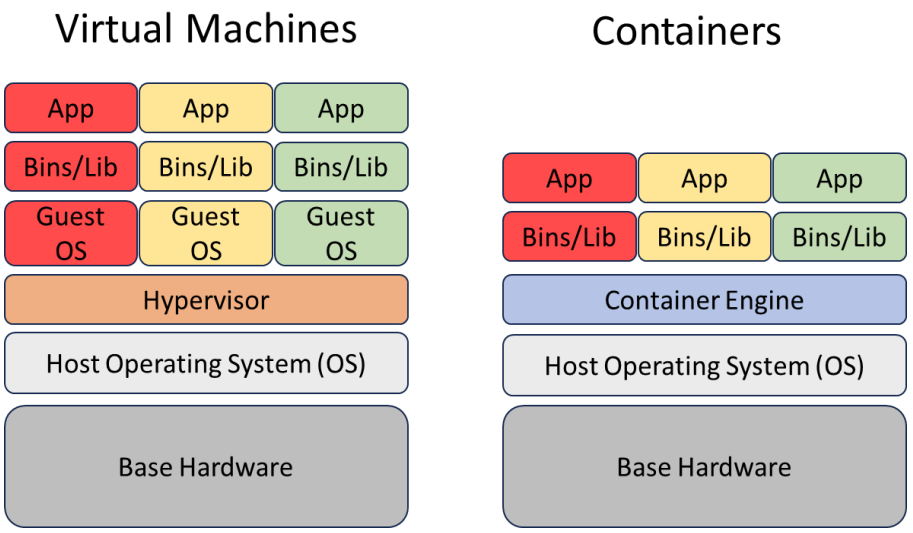

Figure 2.35 Comparing VMs and Containers - As opposed to VMs that need full copies of the guest OS and a hypervisor, containers are more efficient as they share the host OS and run on top of a container engine

Containers share the same kernel, but can be constrained to use a defined amount of resources like CPU, memory, and I/O. The key technologies behind containerisation include namespaces and cgroups. Namespaces ensure process isolation, effectively making each container appear as a standalone system to its processes. On the other hand, Control Groups ("cgroups") limit, account for, and isolate the usage of resources like CPU, memory, disk I/O, network.

2.5.1 Docker

Docker is an open-source platform that automates the deployment, scaling, and management of applications using containers. Docker evolved from earlier container technologies like Linux containers (LXC), building on its capabilities with easier configuration and deployment, making it a pivotal tool for developers and sysadmins aiming for efficiency and scalability in application deployment (Merkel, 2014).

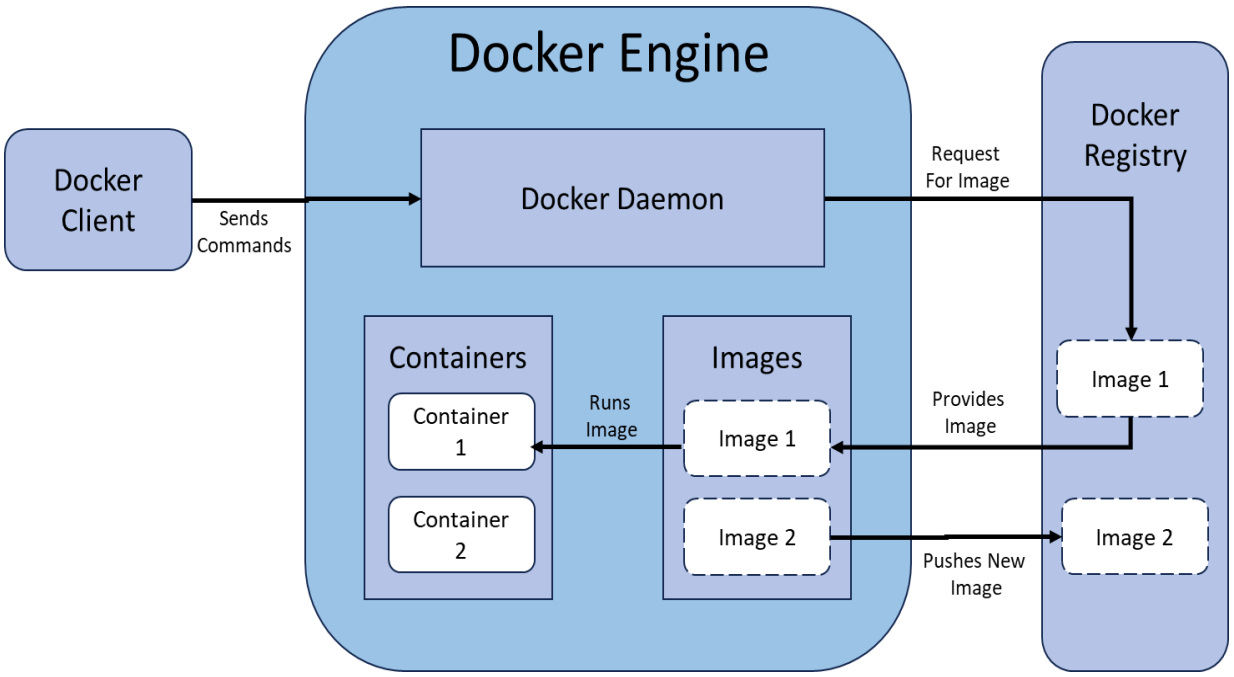

Figure 2.36 Underlying Docker Components - The Docker Engine runs the runtime environment that builds and runs the containers and images using the Docker Daemon. The Docker client sends commands to the Daemon, which executes these commands. The Docker Daemon can push or pull images from the Docker Registry, which it uses to build and run containers.

The core components of Docker are:

- **Docker Engine:** The Docker Engine is the runtime environment that builds and runs containers. It operates as a client-server application and communicates with the Docker Daemon, which is responsible for creating and managing Docker images and containers.

- **Docker Daemon:** The Docker daemon is the background service that manages the state of Docker containers and images. It handles the tasks associated with building, running, distributing, and monitoring containers. The daemon accepts commands from the Docker CLI or API, performs the requested operations, and maintains the lifecycle of containers.

- **Docker Images:** Docker images are read-only templates used to build containers. They contain the source code, libraries, dependencies, tools, and other files needed for an application to run. A Docker image is built from a Dockerfile, which is a plaintext file that specifies all the commands, steps, and environment necessary to build the image. Images are stored in a Docker Registry, such as Docker Hub, from where they can be downloaded and used to create containers.

- **Docker Containers:** A Docker container is a runnable instance of a Docker image. When an image is run, the result is a container running that image. Containers encapsulate the application, its environment, and a minimal runtime to manage the application's lifecycle. Containers can be started, stopped, moved, and deleted. Each container is isolated from others and from the host system, sharing only the kernel and, optionally, portions of the file system.

2.5.2 Why Use Containers

Containerising applications brings numerous benefits that address both technical and operational challenges in software development and deployment.

Consistency Across Environments: One of the foremost advantages of containerisation is the promise of consistency. Containers encapsulate an application and its dependencies into a single executable package. This encapsulation ensures that the application runs the same way, irrespective of where it's deployed, be it on a developer's local machine, a test environment, or a production server in the cloud. This consistency eliminates the infamous "it works on my machine" syndrome, thereby reducing bugs and discrepancies that arise from environment-specific configurations.

Rapid Deployment and Scaling: Containers are lightweight compared to traditional virtual machines, as they share the host system's kernel and do not require an OS per application. This characteristic enables faster start-up times, making deployment rapid and scalable. Applications can be quickly scaled in and out in response to demand, and load balancing can be more effectively managed.

Improve Microservice Architectures : Containerisation supports a microservices architecture (Nadareishvili et al., 2016), where applications are broken down into smaller, independently deployable services that communicate over well-defined APIs, applying the principle of "separation of concerns". This architecture allows development teams to work in parallel, improving productivity and accelerating development cycles

Isolation: Containers provide process and namespace isolation, which improves security and allows for more granular control over resources. Each container has its own filesystem, CPU, memory, and network resources, preventing applications from interfering with each other. This isolation also simplifies dependency management, as each container can have its own set of dependencies, unaffected by those of other containers.

2.5.3 Container Orchestration

Container Orchestration goes beyond the basics of running individual containers, by addressing the need for efficient operation, maintenance, and scalability of containers at scale.

Seamless Updates and Rollbacks: Orchestration facilitates the deployment of new containers with minimal downtime, leveraging rolling updates and blue-green deployment strategies that gradually transfer user traffic from a previous version of an app or microservice to a newer one. This ensures that applications remain available even as updates are being deployed. If an update does not perform as expected, orchestration tools enable quick rollback to a previous version, minimising impact on the application's availability and user experience.

Automated Health Checks & Self-healing: Through continuous monitoring of container health and application performance, orchestration systems can identify failing containers and automatically restart them on the same or different hosts. This "self-healing" mechanism ensures that applications remain reliable and resilient, even in the face of unexpected failures.

Efficient Resource Utilisation Through Scheduling: Orchestration systems optimise the use of underlying hardware resources by intelligently scheduling containers based on their resource requirements and the current load of cluster nodes. This dynamic resource allocation helps in maximising resource efficiency, reducing costs, and ensuring that applications have access to the resources they need when they need them.

Security and Compliance: Orchestration tools often incorporate security policies and compliance checks as part of the container lifecycle management. This includes managing who can access and deploy containers, encrypting data in transit and at rest, and ensuring that containers are updated with the latest security patches. By automating these processes, orchestration helps maintain a secure container environment.

Simplified Network Configuration and Service Mesh Integration: As applications grow in complexity, so does their internal and external communication architecture. Orchestration tools simplify network configurations, manage ingress and egress traffic, and facilitate service mesh integrations. These capabilities enable secure, efficient, and reliable communication between microservices and with the outside world.

Persistent Storage Management: Stateful applications require persistent storage that outlives container restarts. Orchestration systems manage persistent volumes and storage classes, providing containers with consistent and reliable access to data. This management includes provisioning, attaching, and detaching storage as containers are moved or scaled across the cluster.

2.5.3.1 Available Orchestration Tools

Several such tools have been developed to manage the complexity of deploying, managing, and scaling containerised applications. These tools each offer unique features tailored to different use cases, performance needs, and operational complexities. Docker Swarm is Docker's native orchestration tool, designed for simplicity and ease of use. It uses the standard Docker API, making it straightforward for those already familiar with Docker to adopt. Swarm allows for easy deployment, scaling, and management of container clusters, offering a seamless way to turn a pool of Docker hosts into a single, virtual Docker host.

Its primary advantage lies in its integration with Docker; however, Docker Swarm is often seen as less flexible for complex, multi-container app deployments primarily because it lacks advanced features for service discovery, load balancing, and auto-scaling.

Apache Mesos is another open-source cluster manager that abstracts compute resources such as CPU, memory, and storage across a network of machines, facilitating the efficient operation of distributed systems on a large scale. Marathon, running atop Mesos, is a framework for container orchestration, adding crucial functionalities like service discovery, load balancing, and application health management, making it well-suited for managing long-running, containerised applications that require high scalability, resource optimisation, and fault tolerance. However, their complexity and the operational overhead involved make them less appealing for smaller projects or teams. Additionally, the Mesos and Marathon ecosystem, while robust, does not match the vibrancy and support network of some alternative solutions.

These factors, combined with the steep learning curve associated with deploying and managing a Mesos-Marathon cluster, often lead users to consider other orchestration systems that prioritise simplicity, specific feature sets, or have stronger community backing, despite Mesos and Marathon's proven capabilities in resource management and scalability.

2.6 Kubernetes

Kubernetes, often referred to as K8s, is an open-source platform developed by Google using GoLang, and has become the leading container orchestration tool, distinguishing itself from alternatives like Docker Swarm and Apache Mesos, through its comprehensive management capabilities, scalability, and flexibility.

Specific ways Kubernetes excels:

- **Self-healing Capabilities**: Kubernetes continuously monitors the state of applications and automatically restarts containers that fail, replaces and reschedules containers when nodes fail and terminates containers that don't respond to user-defined health checks

- **Service Discovery and Load Balancing**: Kubernetes automatically assigns containers their own IP addresses and a unique DNS name for a set of containers, which can balance load between them. This is achieved without additional configuration, unlike systems like Docker Swarm, where load balancing requires extra setup.

- **Rich Ecosystem and Community Support:** Being the standard for container orchestration, Kubernetes benefits from a vast ecosystem of tools and integrations developed by a large and active community.

- **Extensibility and Customisation**: Kubernetes allows for high degrees of extensibility and customisation through Custom Resource Definitions (CRDs) and the aggregation layer, enabling users to extend the Kubernetes API or integrate with existing infrastructure, a contrast to more rigid or less customisable systems.

2.6.1 Kubernetes Cluster Architecture

Kubernetes architecture is structured around a cluster model, centralising around Nodes, Pods, and the Control Plane (*Kubernetes Components*, 2024). Nodes serve as the hardware layer, facilitating pod execution. Pods, encapsulating one or more containers, represent the minimal deployment units, enabling application scaling and management.

The Control Plane orchestrates cluster operations, using its components for interface access, state storage, pod placement, and background task handling. This architecture enables efficient container orchestration, resource allocation, and system state management within a distributed computing environment.

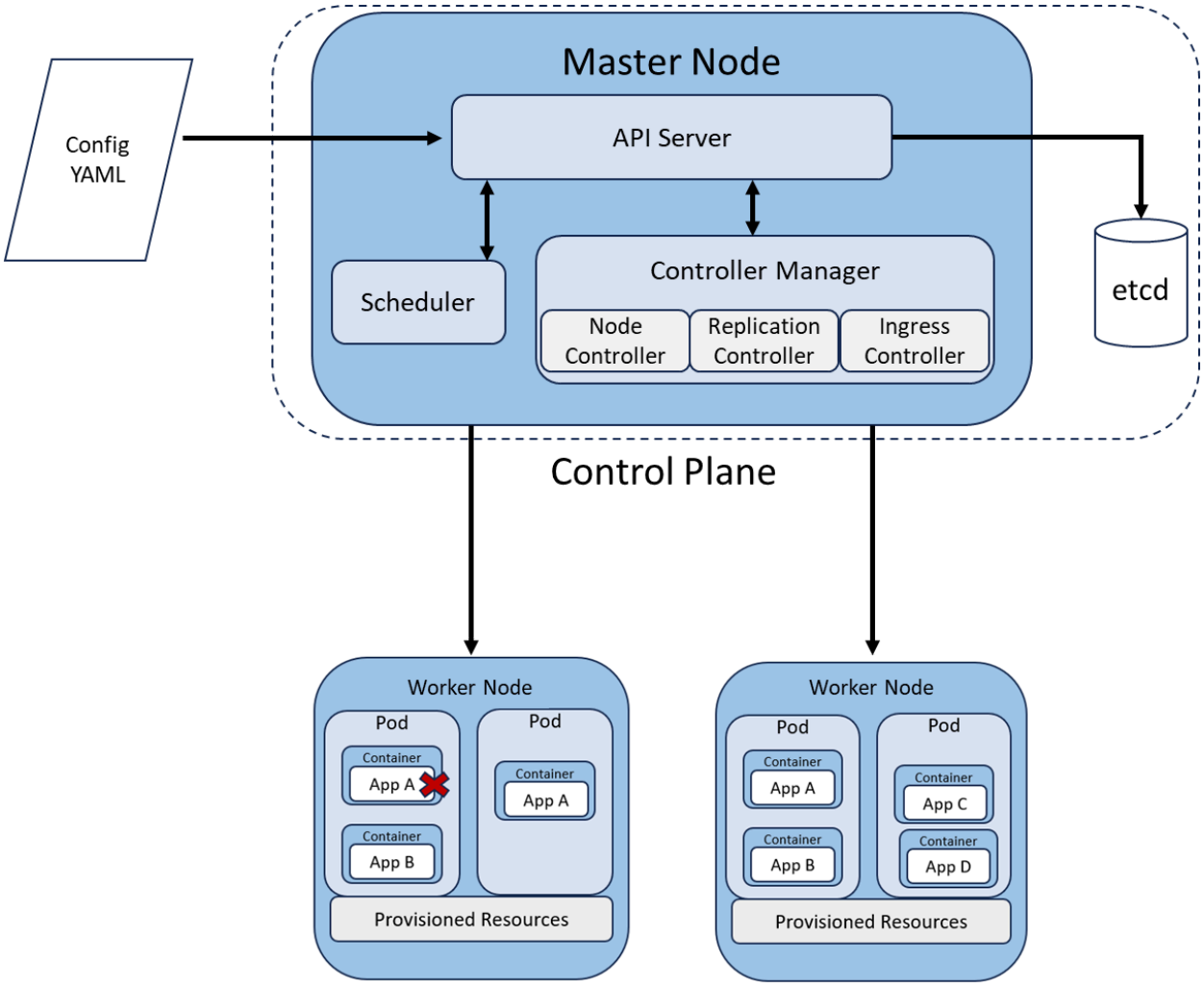

Figure 2.37 Overview of Kubernetes Architecture - The configuration passed into the Control Plane defines a desired state, and is used by the API Server to communicate the changes that need to be made to to match this desired state.

2.6.1.1 Nodes & Pods

Nodes and Pods are fundamental to understanding the Kubernetes Architecture. Each node in a Kubernetes cluster can host multiple pods, and the Kubernetes control plane manages the orchestration of pods across these nodes, taking into account the resources available on each node and the requirements of each pod.

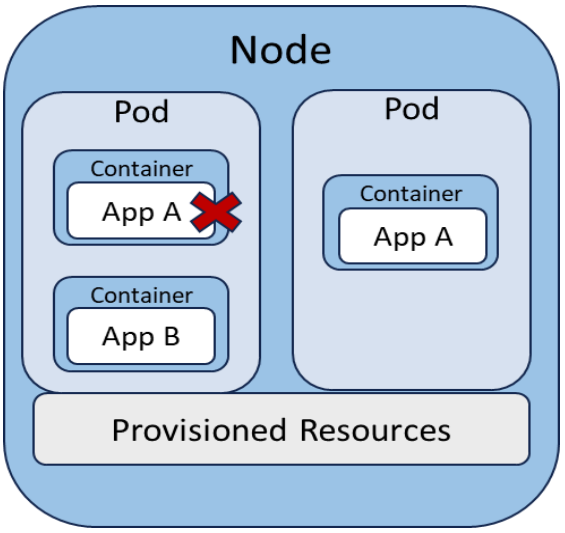

Figure 2.38 Kubernetes Node Structure - A Node can run Pods inside of it, depending on its provisioned resources. These Pods run the actual application containers.

Nodes:

- A node is the smallest unit of computing hardware in Kubernetes. It represents a single machine in a cluster, and provide the environment in which pods are scheduled and run.
- Each node provides the necessary resources to run applications, such as computational power, memory, and networking.
- There are two types of nodes: master nodes, which host the Control Plane components, and worker nodes, where the actual application containers run.
- The worker nodes communicate with the master node to receive instructions on which containers to run and where to run them.

Pods:

- A pod is the most basic, user deployable object in Kubernetes. It represents a group of one or more containers, with shared storage and network resources, and a specification template for how to run the containers. When an application is deployed on Kubernetes, it is actually being deployed in a pod.
- Each pod is designed to run a single instance of a given application, which may need more than one container to run properly. If the application needs to be scaled, Kubernetes creates more pods to match the desired state.
- Pods can be thought of as a wrapper around containers, providing a logical grouping of one or more containers that are scheduled on the same node.
- As containers in the same pod share the same network IP address, port space, and storage, it allows them to communicate and share data with each other easily.

2.6.1.2 The Control Plane

The Kubernetes control plane is the governing entity for a Kubernetes Cluster, providing the core logic for managing its state by managing the worker nodes. The control plane itself runs inside a master node.

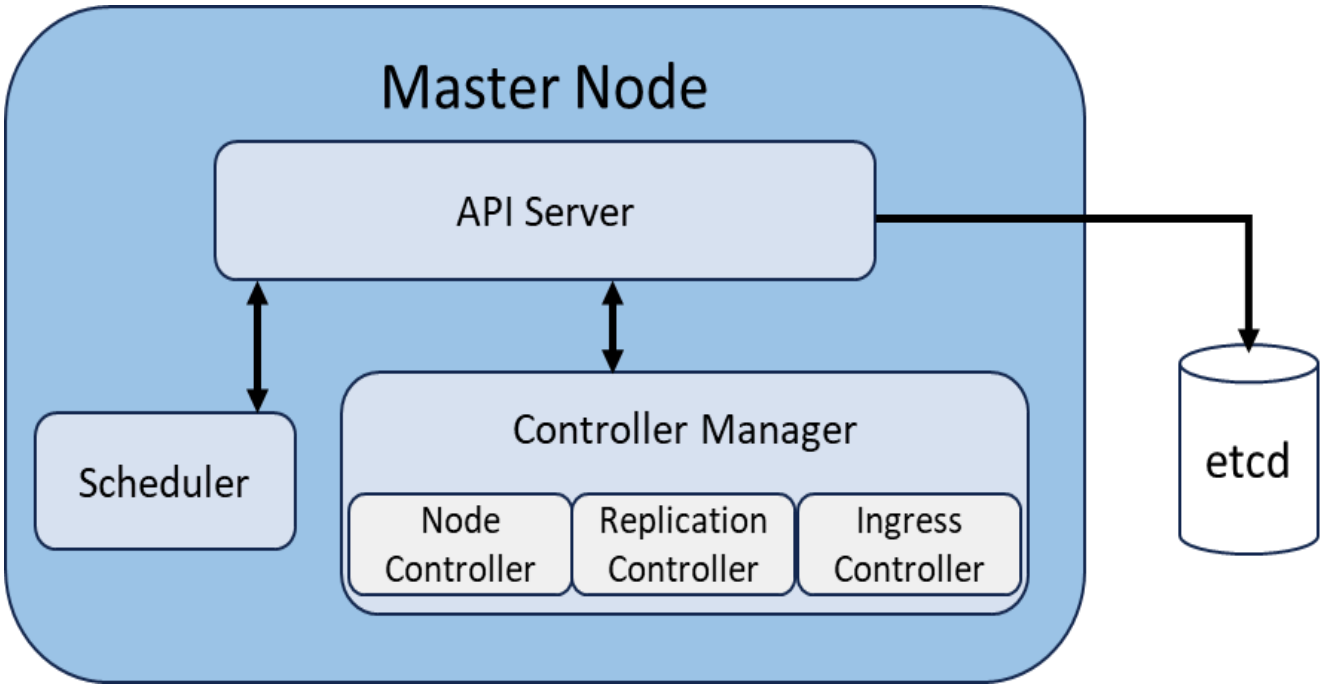

Figure 2.39 The Control Plane - The API server communicates with the Scheduler, Controller Manager and "etcd" to control the state of the cluster by managing worker nodes.

API Server:

- This acts as the front-end for the Kubernetes control plane.
- It exposes the Kubernetes API, which is the primary management interface of the cluster.
- The API server processes REST commands, validates them, and updates the corresponding objects in etcd, Kubernetes' backing store.

The "etcd":

- It stores the cluster configuration data of the cluster in a key-value store, and represents the state of the cluster at any given point in time.
- It includes information like secrets, service definitions and deployment declarations

Scheduler:

- Watches for newly created pods with no assigned node, and selects a node for them to run on based on several factors such as resources requirements, hardware/software/policy constraints, and other user-specified constraints.
- The scheduler takes into account individual and collective resource requirements, quality of service requirements, and hardware/software constraints.

Controller Manager:

- Runs controller processes, the background threads that handle routine tasks in the cluster.
- The job of a controller is to match the current state with the desired state of the system.
- Key controllers include the node controller, responsible for noticing and responding when nodes go down, the replication controller, responsible for maintaining the correct number of pods for every replication controller object in the system, and most relevant to this paper, the ingress controller, which manages incoming traffic to the cluster.

2.6.2 Kubernetes Resources

In practice, Kubernetes uses a range of Kubernetes Resources to configure and manage a Kubernetes cluster. These resources are objects in the Kubernetes API which Kubernetes uses to represent the state of a cluster. From basic units like Pods to more complex abstractions like Deployments and Services, each resource plays a crucial role in the cluster's operation. Understanding these resources is fundamental to effectively using Kubernetes, but Pods, Deployments, and Services are often considered the backbone of Kubernetes workloads.

Key Kubernetes Resources:

- **Pods:** Each pod encapsulates one or more containers, providing them with shared storage resources and a singular network IP. This grouping allows containers within the same pod to share their state and communicate directly.

- **ReplicaSets**: These ensure that a desired number of pod replicas are continuously running at any given time. ReplicaSets are often used in tandem with Deployments to provide high availability and facilitate the desired state management of pods.

- **Deployments:** Focused on the automation of deploying and updating instances of application replicas. A Deployment utilises a Pod template, which outlines specifications for the Pods it manages, allowing for systematic updates through rolling update mechanisms that replace old Pods with new ones as defined in the template.

- **Services:** An abstract way to expose an application running on a set of Pods as a network service. Services provide a persistent endpoint to access a dynamic set of Pods, ie, a consistent way to communicate with applications. By providing a persistent endpoint for accessing Pods, Services decouple access to these applications from the underlying pod configuration, which might change due to scaling or updates.

- **Ingress (Resource):** This resource manages external access to the services in a cluster,and can provide load balancing, SSL termination, and name-based virtual hosting, based on its configuration. It acts as an entry point for external traffic destined for the services within the cluster.

- **ConfigMaps and Secrets:** ConfigMaps provide storage for non-confidential data in key-value pairs, while Secrets store sensitive information, such as passwords, OAuth tokens,

and SSH keys, ensuring that these details can be used securely in the cluster. These are key for separating configuration details from application images, thereby enhancing portability.

- **Volumes**: Offer a method for persisting data generated by and used by the containers in Pods, across restarts or rescheduling of these Pods. Volumes can be backed by various storage backends including local storage or cloud-based solutions, depending on the setup and requirements.

- **Namespaces**: Provide a scope for names in a cluster, allowing the partition of cluster resources into logically named groups. Namespaces are typically used for environments with many users spread across multiple teams or projects.

- **StatefulSets:** These are used for managing stateful applications and provide guarantees about the ordering and uniqueness of these Pods. They make it easier to deploy and scale sets of Pods while maintaining a stable network identity and storage.

- **DaemonSets**: Ensure that each Node in your cluster runs a copy of a specified Pod. This is crucial for deploying system services that need to run on every Node, such as log collectors and monitoring agents.

2.6.3 Kubernetes Networks

Kubernetes ensures that each pod has a unique IP address, resulting in a "flat network model" (*Kubernetes Networking: The Complete Guide*, n.d.), which simplifies the networking as there is no need to explicitly manage port mappings between containers and the host. This approach ensures that containers within a pod can communicate with each other using "localhost", and that pods can communicate with each other across the cluster without the use of Network Address Translation (NAT).

Kubernetes itself primarily relies on TCP and HTTP for its internal operations, especially for critical communications that require reliable transmission, such as the interactions between the Kubernetes control plane components. This reliance on TCP ensures that commands and state information are consistently and reliably communicated across the cluster. However, Kubernetes also supports UDP for applications running within the cluster, as part of its flexible networking model. The support for UDP allows developers and operators to deploy and manage applications that benefit from UDP's characteristics, such as lower latency and less overhead than TCP, which can be advantageous for certain types of applications like media streaming, real-time analytics, or gaming servers. This means that QUIC can be used inside of Kubernetes, as it is built on top of UDP, and Kubernetes already supports it.

The management of external traffic that needs to reach services within the cluster is facilitated through the use of an Ingress Controller, which serves as the gatekeeper for incoming communications. Ingress Controllers are responsible for implementing a set of rules for routing external HTTP and HTTPS traffic to internal Kubernetes services (*Ingress Controllers*, 2023).

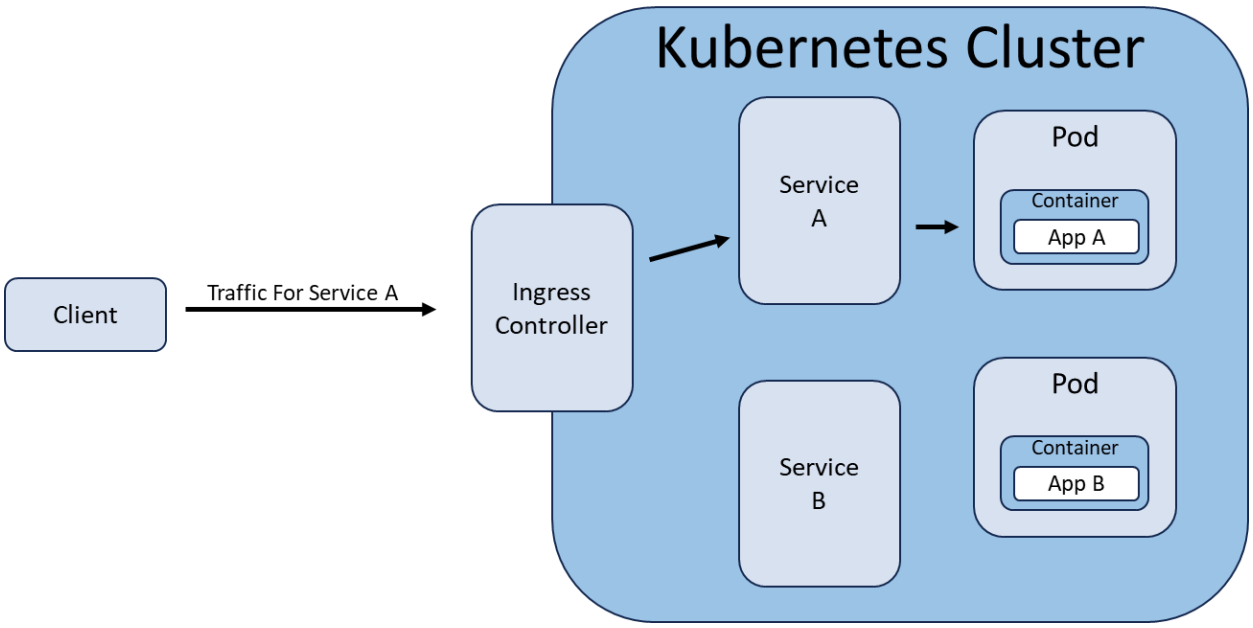

Figure 2.40 Ingress Controller Routing - External traffic enters the Cluster through the ingress controller, which routes it appropriately to Services that expose pod applications.

Ingress Controller Functions:

- **Load Balancing:** Ingress Controllers distribute incoming traffic across multiple backend services, enhancing the performance and reliability of application deployments.
- **SSL/TLS Termination:** Handling SSL/TLS at the Ingress level centralises certificate management and removes the need for individual pods to decrypt traffic, thereby optimising resource utilisation.
- **Host and Path-Based Routing:** Ingress rules allow routing decisions based on the URL path and the host, directing traffic to different backend services based on the content of the request.

Two popular Ingress Controller implementations are Traefik and Nginx.

- **Traefik:** Traefik is designed with a focus on automated and dynamic configuration. It excels in environments where services are frequently updated, as it automatically detects changes and adjusts its routing rules without manual intervention. This makes it particularly suited for microservices architectures that are highly dynamic.

- **Nginx:** Nginx, on the other hand, is renowned for its performance and stability. Its event-driven architecture allows it to manage thousands of concurrent connections on a single thread without significant memory overhead. This efficiency is crucial in high-load environments where performance is a key concern. It handles a vast amount of traffic with minimal resource consumption, making it ideal for high-load environments. Nginx also has a larger and more established support community, as it has been in the industry for longer.

Chapter 3
Problem Statement

The TCP/IP model has been instrumental in shaping internet communications, by streamlining the complex OSI model into four functional layers. TCP, the main transport layer protocol under this model, ensures reliable data transmission and has been the backbone of network communication. However, TCP's inherent limitations, such as head-of-line blocking and slow connection establishment, have prompted the need for newer, more efficient protocols.

The introduction of QUIC represents a significant leap forward. Built on top of UDP, QUIC addresses TCP's shortcomings by reducing latency through minimised handshake times and eliminating head-of-line blocking by allowing multiple streams of data to be multiplexed over a single connection. This is particularly advantageous for real-time applications where delays caused by packet loss and retransmission can degrade performance. The integration of TLS within QUIC also provides end-to-end encryption at the transport layer, enhancing the security posture of communications.

The rise of container technologies, led by Docker, has revolutionised application deployment by encapsulating applications in lightweight, portable containers. Kubernetes extends these capabilities by providing a framework for automating the deployment, scaling, and management of containerised applications. It supports complex, distributed system architectures and enables microservices to scale dynamically and operate independently. Kubernetes not only automates various aspects of container management but also integrates with network protocols to enhance application delivery and performance. Its ability to handle service discovery, load balancing, and automatic scaling makes it an essential tool for modern microservices architectures that require rapid responsiveness and high availability.

3.1 Challenges With Multiplexed Connections

The evolution of network protocols and the widespread adoption of containerised microservices, through the use of Kubernetes, outlined above, have enhanced the scalability and flexibility of web applications.

However, to maximise the advantages of Kubernetes' microservice architecture, a significant problem needs addressing: **the efficient handling of streams within a multiplexed connection**.

Until recently, handling a multiplexed connection in Kubernetes was inefficient, because of TCP and HTTP limitations, those specifically being: head-of-line blocking, limitations of TCP's unified congestion control, and the latency and overhead associated with TCP connection establishment.

Head-of-Line Blocking: One of the persistent issues with TCP, even when used with HTTP/2, is head-of-line blocking. This bottleneck is particularly detrimental in a microservices architecture where different services may be communicating concurrently over the same connection. For instance, if a video stream encounters a packet loss, not only is the video stream affected, but so are other streams, such as API calls or data uploads, which are independent yet stalled due to TCP's in-order delivery requirement. This limitation severely impacts the overall system throughput and increases latency, making the performance of the entire microservices architecture unpredictable and inefficient. The dependency of stream progress on the successful transmission of all other streams leads to increased response times and can degrade user experience and service reliability.

Stream Management Limitations: In microservices architectures that utilise HTTP/2, multiple streams share a single TCP connection, subjecting them to unified congestion and flow control mechanisms. This configuration significantly limits the ability to manage streams effectively according to their specific needs. A high-priority video stream might be delayed due to congestion caused by lower-priority text data, thus affecting the quality of service and responsiveness. This "one-size-fits-all" approach to congestion management does not align with the dynamic requirements of microservices architectures, where different services might have varying priorities and need isolated control over their network performance to operate efficiently.

Connection Establishment and Overhead: The process of establishing a secure TCP connection involves a lengthy handshake mechanism and a slow-start congestion control phase, both of which introduce considerable latency. This setup is particularly problematic in microservices environments that demand rapid scaling and responsiveness. Each new instance of a service or a microservice component might require the establishment of new TCP connections. Frequent scaling up and down, typical in cloud-native environments responding to real-time demand or service health, exacerbates this issue. The latency incurred during the handshake and slow-start phases can lead to delays in service provisioning and responsiveness, hindering the ability to scale services quickly and efficiently.

3.2 Proposed Solution

The integration of QUIC into Kubernetes offers a solution to the stream handling challenges detailed above, addressing the issues of head-of-line blocking, stream management limitations, and the inefficiencies of TCP connection establishment.

The proposed solution is to develop an approach to distribute individual streams within a multiplexed connection, **based on their content**, which is connected to a group of microservices inside a Kubernetes Cluster, and to demonstrate how QUIC can be effectively used with Kubernetes to reduce past pain points of TCP. Doing so would implement a new way of routing within microservices architectures, better aligning with the principles of the microservice architecture, and setting a path for future enhancements in network communication within distributed systems.

QUIC is specifically designed to overcome these hurdles, making it an ideal protocol for enhancing the networking capabilities within microservices architectures facilitated by Kubernetes.

Overcoming Head-of-Line Blocking: QUIC inherently solves the problem of head-of-line blocking by implementing its own independent stream multiplexing within a single connection. Unlike TCP, where packet loss in one stream can delay all others due to sequential segment processing, QUIC allows each stream to operate independently. This independence means that a packet loss in one stream does not stall others; each stream continues to receive or send data as available, significantly reducing delays and improving overall latency.
This feature is particularly beneficial in microservices environments, where multiple services communicate simultaneously, often with varying criticality and performance requirements. By ensuring that communication in one service does not affect others, QUIC enhances the resilience and efficiency of the entire system.

Enhanced Stream Management: QUIC provides more granular control over stream management and prioritisation. It enables the setting of priorities on a per-stream basis, allowing critical services such as live video feeds to be prioritised over less time-sensitive data like background synchronisation tasks. This capability aligns well with the dynamic nature of microservices architectures, where resource allocation and prioritisation can directly impact service performance and resource utilisation. By allowing for individual stream management, QUIC makes it possible to optimise network traffic according to the specific needs and operational demands of each service within a Kubernetes cluster.

Efficient Connection Establishment: QUIC reduces the latency associated with establishing connections by combining the handshake and data transfer steps, significantly cutting down the time required to initiate communications. The reduced latency in connection establishment provided by QUIC is therefore a critical advantage in maintaining high responsiveness and agility in microservices operations.

Chapter 4
Design

This chapter outlines an abstract solution that can effectively manage streams in a multiplexed HTTP/3 connection to a Kubernetes Cluster by leveraging QUIC and its capabilities. The proposed design is crafted to be independent of specific implementations, ensuring its applicability across different technology stacks, Kubernetes deployments, and network settings.

4.1 High-Level Architecture

The architecture is composed of three main components:

- **Client:** Responsible for initiating multiplexed HTTP/3 connections.
- **Ingress Controller:** Manages the reception of these connections within the Kubernetes cluster, and responsible for splitting the data streams.
- **Backend Services:** Process the data streams routed to them by the ingress controller.

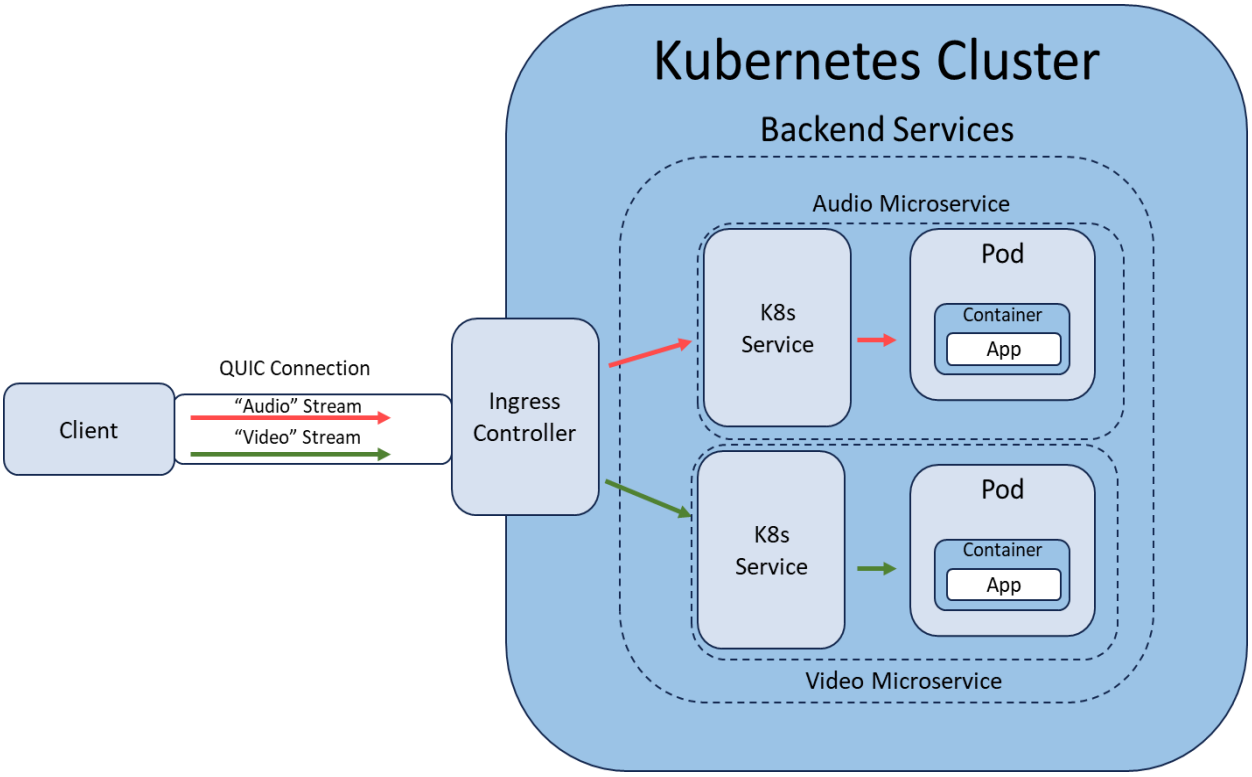

Figure 4.1 High Level Overview Of Proposed Design - Client QUIC connections have their streams split and routed individually by the Ingress Controller, to facilitate efficient stream handling.

4.2 Addressing Stream Splitting & Routing

The proposed solution is to split the streams of a HTTP/3 connection at the Ingress controller. This approach is challenged by the encryption characteristics of QUIC (Thomson & Turner, 2021), where the packets are almost completely encrypted. This encryption shields most packet information from inspection, leaving minimal unencrypted fields such as packet numbers and some flags, which are insufficient for stream identification purposes.

TLS Termination

To facilitate effective stream splitting, TLS termination is employed at the Ingress Controller. TLS termination is a standard feature in many Ingress Controllers, primarily used for URL inspection, which can be leveraged to address the constraints posed by QUIC's encryption. By terminating TLS, the Ingress Controller decrypts incoming data, thereby gaining access to the entire content of QUIC packets. This decryption is crucial as it exposes the stream ID of each Stream Frame and the HTTP/3 headers encapsulated within.

A Practical Solution

A practical and efficient method for stream identification and routing involves the use of the HTTP "content-type" header. This header is set by the client when creating requests and indicates the nature of the data in the stream (e.g., application/json, text/html). Upon decryption and inspection at the Ingress Controller, the "content-type" header is analysed to determine the appropriate routing for each stream.

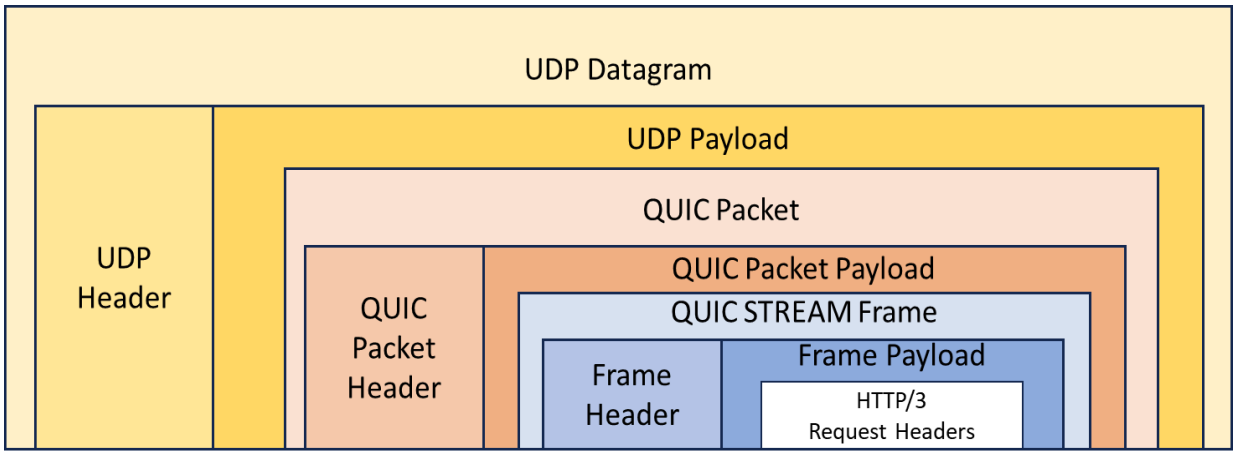

Figure 4.2 HTTP3 Over QUIC - HTTP/3 packets are encapsulated in a QUIC STREAM Frame.

This approach leverages the existing capabilities of Ingress Controllers to enforce security rules and manage traffic based on URL paths and extends this capability to manage data types through HTTP headers. The implementation of this method means that when a client sends a request, the corresponding "content-type" header explicitly informs the Ingress Controller of the content type, enabling the controller to route the stream to the designated backend microservice that specialises in handling that specific type of data.

4.3 System Components

This section will discuss the main system components that make up the High-Level architecture highlighted in Section 4.1.

4.3.1 The Client

As the initiator of the communication process, the client is responsible for formatting and sending HTTP/3 requests via QUIC to the Kubernetes
The following requirements ensure that the client can interact properly with the Kubernetes cluster via QUIC, supporting the multiplexed handling of HTTP/3 streams.

- **Functions to Establish and Maintain a QUIC Connection:** A stable connection is crucial for the reliable delivery and management of multiplexed streams, ensuring consistent communication without interruptions. The client must be capable of establishing a stable QUIC connection to the Kubernetes cluster's ingress controller. This includes handling the QUIC handshake, managing connection state, and maintaining the connection throughout the session.

- **Security and Authentication:** As TLS is integrated within QUIC itself, the client must also support standard security practices, such as server validation and being able to reply to client authentication mechanisms required by the server.

- **Support for HTTP/3 Requests:** The project's main aim requires that the client should fully support HTTP/3. This includes proper formatting of HTTP/3 requests and understanding HTTP/3 responses.

- **Multiplexing Capability:** Multiplexing is a core advantage of QUIC, allowing multiple independent HTTP/3 requests and responses to be interwoven on the same connection, improving resource utilisation and reducing latency. The client must be capable of multiplexing multiple request streams over a single QUIC connection. This involves managing several streams simultaneously, each carrying different types of data.

- **Stream Content Identification:** Correct identification of stream content is critical for the ingress controller to route each stream to the appropriate backend microservice, ensuring that each type of content is handled by the most suitable service. Before sending, the client should appropriately tag or identify the content type of each stream in some way, so that it can be recognised and interpreted by the ingress controller for accurate routing.

- **Responsive to Server Feedback:** Responding appropriately to server feedback allows for adaptive stream management, enhancing overall communication efficiency and responsiveness. The client should be responsive to control messages and feedback from the server, such as congestion control advice, stream cancellation, or priority adjustments

4.3.2 The Ingress Controller

As the gateway to the Kubernetes Cluster, the Ingress Controller is the main component of this system. Its primary function is to manage all inbound traffic, acting as a smart router that directs incoming HTTP/3 streams to the appropriate backend services based on their content.

The Ingress Controller performs TLS termination, which enables the inspection and identification of stream content. By doing so, it can apply predefined rules to route each stream to the correct microservice.

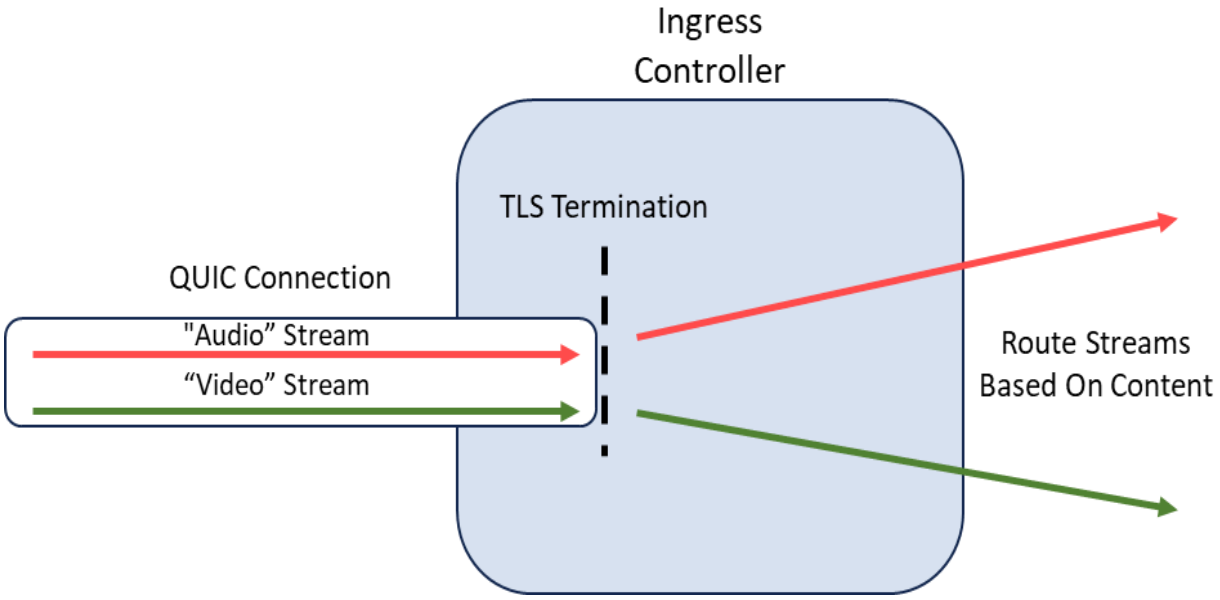

Figure 4.3 Ingress Controller Overview - Incoming QUIC connections are terminated using TLS termination so that their contents can be inspected. Then they are individually based on their content type.

These requirements are critical for ensuring that the Ingress Controller performs its roles of traffic management, security enforcement, and efficient resource distribution.

- **Effective Management of QUIC Connections:** The Ingress Controller must be capable of receiving, maintaining, and managing persistent QUIC connections. This includes handling the complexities of the QUIC protocol such as connection setup, state management, and graceful termination.

- **HTTP/3 Protocol Support:** Full support for HTTP/3 is required to handle and understand the specifics of incoming requests from the client, and to format appropriate responses. This includes the ability to parse HTTP/3 headers and handle data compression.

- **TLS Termination:** As the first point of contact for incoming traffic, the Ingress Controller is responsible for terminating TLS connections. This is crucial for stream content identification.

- **Intelligent Stream Routing:** The Ingress Controller should have routing capabilities to direct each stream to the appropriate backend service, based on the stream's content type. It must be able to interpret tags or headers associated with each stream for accurate routing decisions.

4.3.3 The Backend Microservices

The backend microservices serve as critical endpoints within the Kubernetes deployment, playing a pivotal role in verifying the functionality and precision of the Ingress Controller's data routing capabilities. In this project, the services are specifically designed to act as validators, ensuring that only the expected types of data streams, as defined by the content type, are received by each service. This setup is essential for demonstrating the achievement of the main aim of the project; stream separation and routing based on content.

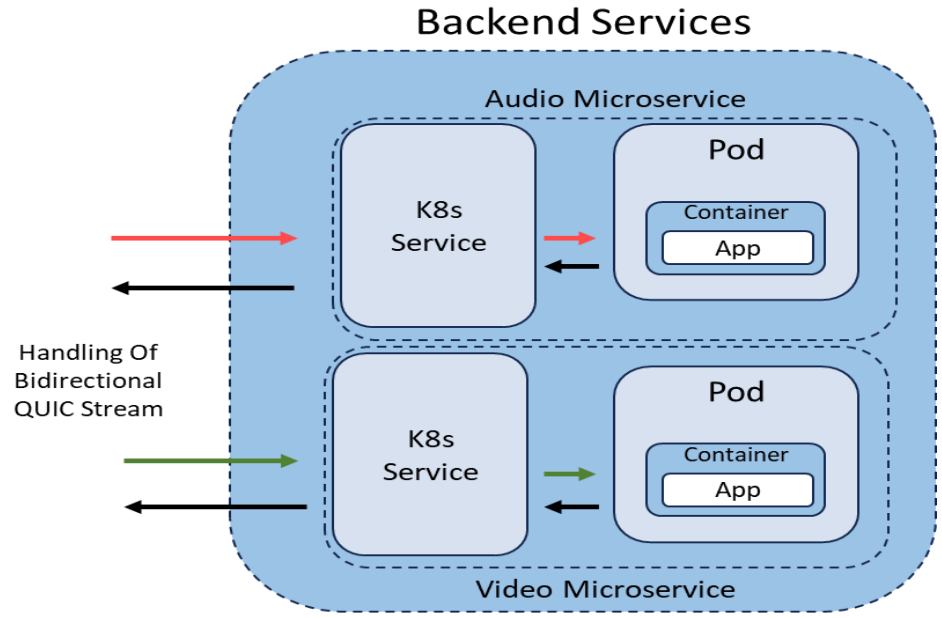

Figure 4.4 Backend Services Overview - The Backend comprises multiple specialised microservices, such as an audio and a video microservice. Each microservice application is running inside of a Pod, and is exposed on the network through a Kubernetes Service resource. Each microservice is capable of receiving and responding to requests.

The microservices have just two, straightforward functional requirements.

- **Accurate Reception and Handling of Data:** Microservices must receive and process data streams routed to them by the Ingress Controller. They should only accept data that matches their pre-configured content type specifications to confirm the accuracy of routing, and raise exceptions when an incorrect data content type is received.

- **Response Handling:** As the endpoints of the Cluster, the microservices should be capable of sending responses to the client for requests that expect a reply. This requires handling bidirectional streams effectively, including the ability to process incoming requests and generate appropriate responses through the same connection.

4.4 Data Flow

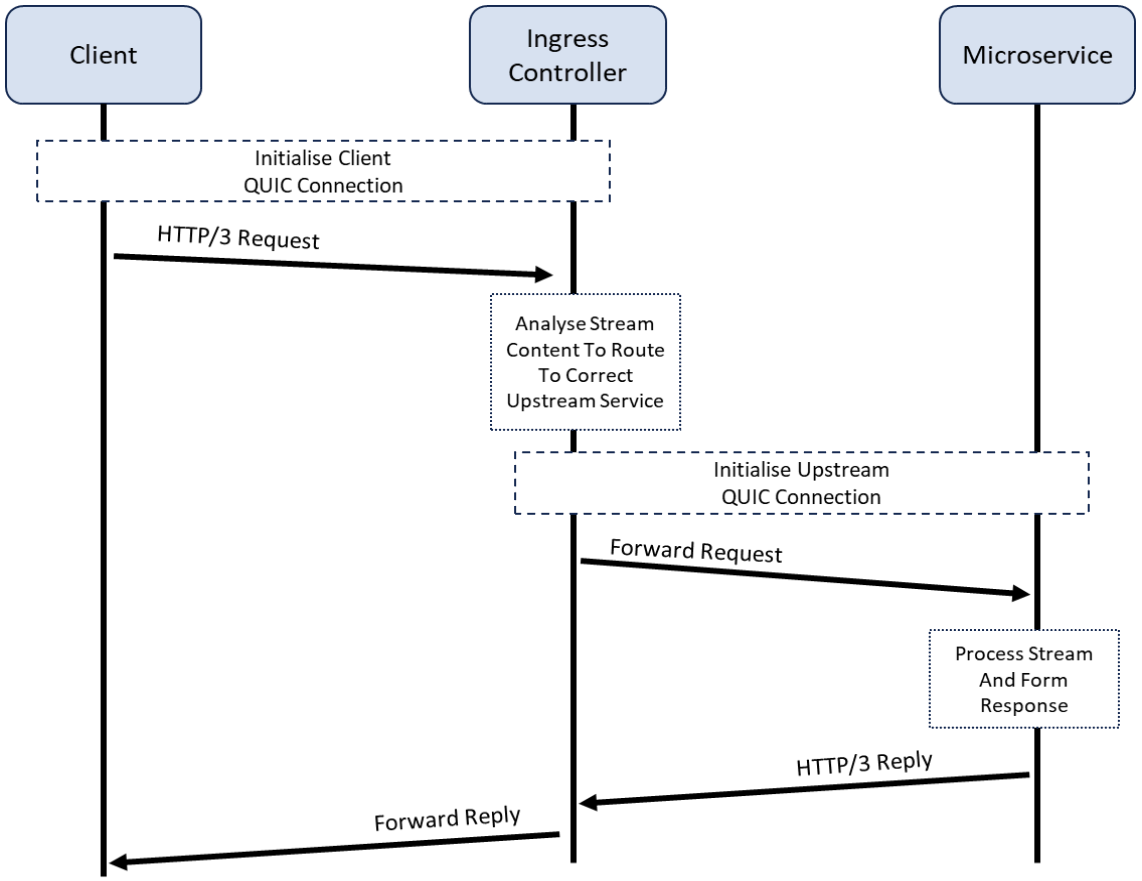

Figure 4.5 Flow Of Data - Client requests are forwarded by the ingress controller to microservices

Client Initialisation:

- The client starts the process by establishing a secure QUIC connection with the Kubernetes cluster's ingress controller.
- It sends HTTP/3 requests, each tagged with a "content-type" header to indicate the nature of the data in the stream.

Ingress Controller Processing:

- Upon receiving the streams, the ingress controller performs TLS termination. This step decrypts the data, allowing the ingress controller to access the full contents of each packet, including stream IDs and headers.
- The controller analyses the "content-type" header of each stream to determine the appropriate backend service for routing.
- Based on the "content-type", the ingress controller routes each stream to the designated backend service that specialises in that particular type of data.

Backend Services Handling:

- The backend service processes the streams sent to them, verifying that they received only the data types they are configured to handle, and sends a response.

Chapter 5
Implementation

This chapter presents an implementation methodology that builds on the theoretical frameworks and design principles discussed in previous chapters. It focuses on the deployment of a system architecture adept at efficiently managing multiplexed connections, addressing the issues outlined in Chapter 3.

The chapter begins by conducting an analysis of options for deploying Kubernetes, discussing the advantages and disadvantages of local versus cloud-based solutions. This leads to the selection of a local setup for development and testing, providing insights into the tools and configurations used. The selection process of the underlying Ingress Controller implementation is extensively discussed, culminating in the choice of ANGIE, an NGINX fork, as the preferred solution. This comprehensive approach ensures a deep understanding of the system's operational dynamics and architectural integrity.

Then the chapter continues by detailing the development of three integral components of the system, beginning with the ingress controller, which handles incoming connections and splits stream traffic to appropriate backend services based on the data type. Then the implementation of the client is outlined, which is capable of establishing a QUIC connection to a Kubernetes cluster, designed to initiate communication effectively. Lastly, the backend services are covered, explaining how they are tasked with processing the incoming data streams. Together, these components validate the system's ability to handle high-volume, multiplexed network traffic, crucial for real-time applications that demand minimal latency and maximum data integrity.

5.1 Running Kubernetes

There is no one best solution to setting up and running a Kubernetes Cluster. Each project has its own individual requirements that should be accommodated for. Within the broader Kubernetes ecosystem, deployment options can broadly be categorised into local and cloud-based solutions. Each category offers distinct advantages and is geared towards different types of projects

5.1.1 Local vs Cloud Solutions

When setting up a Kubernetes cluster, developers have the option to choose between local and cloud-based solutions. Both solutions offer unique advantages depending on the project requirements, budget, team size, and intended use case.

Understanding the fundamental differences between these environments is crucial for making an informed decision about the most suitable deployment method for specific project needs.

Local Solutions

Local solutions allow developers to run Kubernetes clusters on their own hardware. This approach is often favoured for development and testing purposes due to its accessibility, ease of setup, and cost-effectiveness.

Two popular local Kubernetes solutions are Docker Desktop and Minikube:

- **Docker Desktop:** This integrated development environment makes it easy to create and manage both Docker containers and Kubernetes clusters. Docker Desktop includes Kubernetes as part of its installation, providing a seamless experience for developers who already use Docker for containerisation.

- **Minikube:** Designed to run a single-node Kubernetes cluster on a personal computer. It supports various VM drivers and can also run on top of Docker, making it versatile across different operating systems.

Cloud Based Solutions

Cloud-based solutions are hosted and managed by third-party providers. These platforms offer reliability, scalability, and extended services that are difficult to match with local setups.

There are two prominent cloud-based Kubernetes services:

- **Google Kubernetes Engine (GKE):** A managed environment within Google Cloud Platform, GKE allows users to deploy, manage, and scale Kubernetes applications using Google's infrastructure. GKE provides automatic scaling, monitoring, and maintenance, enabling teams to focus on their applications rather than managing the underlying infrastructure.

- **Amazon Elastic Kubernetes Service (EKS):** This service from AWS automates much of the heavy lifting involved in running Kubernetes, including patching, node provisioning, and updates. EKS is deeply integrated with AWS services, providing a rich set of features to enhance and scale Kubernetes applications.

For this project, local solutions offer several benefits over cloud-based solutions, such as:

- **Cost-Effectiveness:** Local environments eliminate the costs associated with cloud services, such as per-hour billing for compute resources. This is particularly advantageous for development and testing phases, where costs can escalate with the complexity and duration of the project.

- **Rapid Workflow:** Local deployments provide faster setup and iteration cycles. Changes can be tested immediately without the latency associated with deploying to a remote cluster.

- **Simplified Networking:** Local setups typically involve less complex network configurations compared to cloud environments. This simplicity helps in debugging and reduces the overhead of configuring network policies and VPNs to secure cloud communications.

In summary, while cloud solutions offer scalability and ease of maintenance, local Kubernetes deployments like Docker Desktop and Minikube provide the flexibility, cost control, and simplicity needed for this project.

5.1.2 Why Docker Desktop

While it is acknowledged that Minikube offers greater flexibility and is preferable in scenarios requiring detailed configuration customisations or multi-platform compatibility, the choice of Docker Desktop for deploying Kubernetes in this project was based on its simplicity, highlighted with the following:

Installation and Initial Configuration

- **Docker Desktop:** Setting up Kubernetes with Docker Desktop involves simply enabling the Kubernetes option within the Docker Desktop application settings. This process automatically configures the cluster to use the same Docker daemon as containerised applications, requiring no additional installation steps.

- **Minikube:** Installing Minikube typically involves downloading the Minikube binary and setting up a virtual machine using a chosen driver (e.g., VirtualBox, VMware, or Docker itself). Each driver may require specific configurations, such as allocating CPU resources, memory, and storage. The command to start Minikube “minikube start” attempts to automatically select a VM driver based on the system configuration. The Docker driver is usually the simplest choice, as it will leverage Docker containers instead of an actual VM.

Networking Configuration and Integration

- **Docker Desktop:** In Docker Desktop, Kubernetes services automatically utilise Docker’s network settings because the Kubernetes nodes are Docker containers themselves. This setup inherently simplifies network management as there is no need to establish and configure a separate network bridge or routing rules. Services deployed in Kubernetes can readily communicate with each other and the host system, using Docker’s native network configuration, which includes automatic DNS resolution and IP address management.

- **Minikube:** Networking in Minikube, involves setting up a separate virtual network within the VM. To expose services to the host machine, Minikube provides the “minikube tunnel” command, which sets up a route between the host and the Minikube VM, allowing LoadBalancer services to receive an external IP address.

Volume Management and Data Persistence

- **Docker Desktop:** Volume management in Docker Desktop benefits from direct integration with Docker's volume driver. This integration facilitates consistent handling of data persistence across container restarts and pod migrations. Specifically, Docker manages volumes created for Kubernetes, ensuring that data stored on these volumes persists across pod restarts and survives even if the pod moves across different nodes within the Docker-managed Kubernetes cluster.

- **Minikube:** In contrast, Minikube handles volumes using a variety of storage classes, which may require manual configuration to ensure compatibility with the host system's storage solutions. This could involve setting up hostPath volumes that map directly to specific directories on the host machine, or configuring persistent volume claims that are appropriately supported by Minikube's underlying VM. These steps add complexity and can introduce variability in how data persistence is managed, depending on the Minikube setup and the storage drivers used.

5.2 The Ingress Controller

Central to the proposed system is the ingress controller, a pivotal component designed to enhance the system's functionality and security. Its primary responsibility is the intelligent splitting of streams based on content type, ensuring that each data stream is accurately routed to the appropriate backend services for processing.

5.2.1 Analysing Ingress Controller Options

Choosing the right ingress controller involves selecting an implementation that fulfils as many of the requirements in Section 4.3.2 as possible. The closer an ingress controller is to meeting these functionalities out-of-the-box, the less customisation and additional work will be required.
An analysis of the two most popular Ingress Controller implementations, Traefik and Nginx, was conducted. This analysis delved into the technical specifics of their latest versions, Traefik 2.11 and NGINX 1.23.1, focusing on their implementations of QUIC connections, HTTP/3 support, TLS termination, and their capabilities to proxy to upstream services.

5.2.1.1 Traefik

QUIC Capabilities: With the release of version 2.11, Traefik has solidified its support for QUIC, transitioning from experimental to more stable support. This includes enhanced session resumption capabilities, which are critical for maintaining performance and reliability over mobile networks where connections may be intermittent. Traefik's implementation of QUIC also supports zero-round-trip time (0-RTT) resumption, reducing latency significantly during session re-establishments.

HTTP/3 Features: Traefik's integration of HTTP/3 fully supports HTTP/3 features on the client side, and benefits from QUIC, enabling faster content delivery by reducing the overhead associated with connection and transport-level negotiations.

TLS Termination & Features: Traefik supports TLS termination, and simplifies TLS configuration, automatically handling certificate renewal and secure traffic management through integrations with Let's Encrypt. This automation is beneficial in dynamic environments like Kubernetes.

Proxy Protocols: Traefik can not proxy over HTTP/3 traffic to upstream services; it supports HTTP/3 only from the client side to the ingress point. Instead, it reverts to HTTP/1.1 or HTTP/2 when communicating with internal services.

5.2.1.2 NGINX

QUIC Capabilities: NGINX has integrated experimental support for QUIC. NGINX's QUIC implementation includes connection establishment, 0-RTT session resumption, stream multiplexing, and allows configuration to turn on features like "quic_rety", which would make the ingress always issue a Quic Retry challenge to connecting clients.

HTTP/3 Features: It also has support for HTTP/3 on the client side, meaning that it can support HTTP/3 connections, which involves decrypting HTTP/3 traffic at the ingress and then handling it internally.

TLS Termination & Features: It can manage multiple SSL/TLS certificates and configurations, enabling secure and encrypted connections. NGINX supports various SSL features such as SSL/TLS protocol tuning, and client certificate validation.

Proxy Protocols: Nginx does not currently support proxying these requests to upstream servers using HTTP/3. Instead, requests are proxied using HTTP/1.1 or HTTP/2, which means the benefits of HTTP/3 end at the termination point.

5.2.1.3 Conclusion Of Analysis

In concluding the analysis of choosing between Traefik and NGINX as Kubernetes ingress controllers, it becomes apparent that both options are well-suited for the task, meeting the primary requirements of managing QUIC connections and supporting HTTP/3 at the client ingress point.

However, an important implementation question arose during the analysis concerning the protocol used to connect to upstream servers. Notably, neither Traefik nor NGINX currently supports proxying QUIC to upstream servers, which means TCP is the fallback protocol. This situation can undermine some of the advantages of using QUIC at the client side, as the internal traffic would not benefit from QUIC's improved performance.

Given these considerations, the decision was made to use NGINX for this project *initially*.
This choice was influenced by several factors:

- **Popularity**: NGINX is widely used and trusted in the industry, which speaks to its reliability, and results in extensive community support.

- **Extensive Documentation**: NGINX provides comprehensive documentation that aids in effective implementation and troubleshooting, essential for complex configurations.

- **Familiarity with Implementation Language:** Personal experience with NGINX and its implementation in C++ is a decisive advantage, as it allows for potential customisation of the source code to meet specific needs that may arise during the project.

5.2.2 The Proxy Protocol Problem

This section will involve a detailed analysis of which protocol should be used when connecting to the upstream servers. This decision is crucial as it will determine the extent to which customisation of NGINX might be necessary to optimise the performance and efficiency of the network architecture. This analysis will aim to align the choice of protocol with the overarching goals of maximising throughput, minimising latency, and ensuring secure connections throughout the network, to ensure efficient management of streams as per the project goal defined in Section 3.2.

5.2.2.1 Proxy Over TCP

When NGINX receives a multiplexed HTTP/3 connection, the scenario becomes a bit complex because HTTP/3 inherently operates over QUIC, which itself is built on top of UDP. Despite the ingress controller's ability to handle HTTP/3 client connections, the internal workings require a fallback to HTTP/1.1 or HTTP/2 which operate over TCP when proxying to upstream servers.
This is what is currently done by NGINX to support proxying HTTP/3 connections:

- **Connection Termination:** Initially, NGINX terminates the HTTP/3 connection at the ingress point, decrypting the QUIC packets that encapsulate HTTP/3 requests.

- **Protocol Translation**: After termination, NGINX translates these requests from HTTP/3 into HTTP/1.1 or HTTP/2. This step is crucial as it adapts the incoming HTTP/3 requests into a protocol format that is compatible with TCP.

- **Forward The Request:** NGINX forwards the by establishing new TCP connections to the required upstream service, for each translated request. Connection pooling or keep-alive strategies can be employed to reuse connections and improve efficiency.

The reliance on TCP for forwarding requests from the ingress to upstream services introduces several limitations that can negatively impact the efficiency and performance of a microservices architecture, negating many of the benefits of using QUIC at the client side.

There are several reasons why this approach is not optimal:

- **Connection Overhead:** Each TCP connection introduces significant overhead due to the necessity of establishing and tearing down connections for each session. This process involves a time-consuming handshake (three-way plus TLS negotiation) which can become a bottleneck, especially in systems that handle a large number of concurrent connections. The cumulative effect of this overhead can lead to increased latency and decreased throughput, impacting the overall performance of the system.

- **Latency:** The inherent characteristics of TCP, which include the extensive handshake procedure and the sequential nature of connection establishment, contribute to higher latency. This is particularly disadvantageous in a dynamic microservices environment where rapid and frequent connections to various services are common. The latency introduced during the setup and teardown of these connections can significantly delay the processing of requests, resulting in slower response times.

- **Resource Utilisation:** In environments where resources are a constraint, the inefficient use of network and server resources under TCP can be particularly problematic. TCP requires separate connections for each client to server interaction, which not only consumes more server resources but also utilises more network capacity. This contrasts sharply with the capabilities of HTTP/3 over QUIC, where a single connection can handle multiple streams of data. The inability to multiplex streams over individual TCP connections leads to a higher number of connections being maintained, which escalates resource consumption and complicates connection management.

5.2.2.2 Proxy Over QUIC

This is how an ingress that supports QUIC on the server side would work:

- **Connection Maintenance:** The ingress server maintains the QUIC connection integrity from the client all the way to the upstream services. This direct handling preserves the end-to-end benefits of QUIC, including reduced connection establishment time and improved security.

- **Protocol Consistency:** Unlike scenarios where protocol translation is necessary, this ingress server does not need to translate HTTP/3 to older protocols for internal communication. Instead, protocol consistency throughout the traffic route.

- **Forwarding The Request:** The ingress server forwards the request over QUIC, by establishing and then maintaining existing QUIC connections to the upstream service. QUIC's connection model supports extensive multiplexing capabilities This allows the server to handle numerous requests concurrently across the same connection, enhancing

throughput and reducing latency.

Due to the TCP's limitations outlined in the previous section, and the improvements brought by QUIC, it was concluded that using TCP to forward traffic to the microservices in the Kubernetes Cluster should be avoided if possible. Supporting QUIC on both the client and server sides of the Ingress, the system would leverage the full potential of HTTP/3 and QUIC. This added a new functional requirement for the ingress server: it should support QUIC on the server facing side. As NGINX did not support this, it would have to be modified to support forwarding requests over QUIC.

5.2.3 Modifying NGINX

To enable NGINX to support HTTP/3 proxying, significant modifications are required in its architecture and source code to accommodate the specifics of the HTTP/3 protocol and its underlying transport protocol, QUIC. This section details some of these changes, focusing on integrating QUIC handling and updating the proxy module, and ensuring overall system compatibility with the new standards.

5.2.3.1 Extending QUIC Connection Management

To implement effective QUIC connection management in NGINX, modifications to the existing connection handling modules, or the development of a new module, would be required. This module would need to interact with NGINX's core event processing system, which handles network events in an efficient, non-blocking manner.

- **Connection Tracking:** The module would track each QUIC connection with its unique connection ID, managing the lifecycle from initiation to closure, including handling of any connection ID changes.

- **Session Resumption:** It would handle session tickets for TLS 1.3, crucial for quick 0-RTT resumption of connections.

- **State Management:** It would maintain state information necessary for QUIC's flow control and congestion control mechanisms, which are integral to QUIC's performance improvements over TCP.

- **Integration with NGINX's Event Loop:** The module would need to integrate seamlessly with NGINX's existing event-driven architecture. This involves registering and handling new types of events specific to QUIC, such as immediate acknowledgment requests, connection migration events, and stream creation and termination events.

- **Error Handling and Teardown:** Proper mechanisms for gracefully tearing down QUIC connections, handling timeouts, and responding to errors (such as decryption failures or protocol violations) should be implemented. These mechanisms ensure the stability of the web server under various conditions.

5.2.3.2 Integration With The Proxy Module

Modifications need to be made to NGINX's "ngx_http_proxy_module", to forward requests to upstream services using QUIC. The following changes need to be made so that NGINX can initiate, maintain, and close QUIC connections to upstream servers:

- **QUIC Stream Management:** Functionality to manage the lifecycle of QUIC streams, are needed to create, manage, and close QUIC streams that are associated with a single request-response cycle. This requires extending NGINX's existing connection handling in the proxy module to support QUIC's multiple streams per connection.

- **HTTP/3 Frame Processing:** Integrating frame processing functions within the proxy module that can encode and decode HTTP/3 frames as per the protocol specifications is another crucial modification. HTTP/3 uses a binary framing layer that is fundamental to its operation. Proper handling of these frames is necessary for the accurate transmission of headers and body data over QUIC.

- **Configuration and Adaptation:** Introduce configuration directives such as proxy_http_version which can be set to 3 to enable HTTP/3, to allow users to configure and enable HTTP/3 proxy forwarding. Additional QUIC-specific settings to fine-tune performance and compatibility would be useful, although perhaps beyond the scope of this project, would allow NGINX to adapt dynamically to different upstream capabilities and client requirements, maximising compatibility and performance.

5.2.3.3 The Harsh Truth

In addition to this, additional modifications such as modifications to NGINX's buffering systems to support the non-blocking, stream-based nature of QUIC, made it clear that integrating HTTP/3 and QUIC functionality into NGINX was no trivial task.

The modifications span from deep within the core networking stack, through the SSL/TLS layers, up to the user-facing configuration syntax. This is not a task that a single developer or even a small team could undertake lightly, and due to the limited time associated with this project, a better solution was needed.

5.2.4 ANGIE As The Ingress

Following further investigation, a fork of NGINX named ANGIE was discovered. It had only recently introduced HTTP/3 proxy support in ANGIEv1.4, by implementing critical changes such as those described in Section 5.2.3.

This discovery provided a timely solution well-suited to the project's constraints.

5.2.4.1 ANGIE Useful Features

The ANGIE ingress controller includes several enhancements that are particularly relevant to this dissertation project:

- **Complete HTTP/3 Support:** Angie integrates and fully supports HTTP/3 out-of-the-box. This support crucially allows the parsing of HTTP/3 header frames, enabling the identification of the "content-type" header for stream splitting.

- **Proxying to Upstream Servers Over QUIC:** Unlike standard NGINX, which typically terminates HTTP/3 connections at the proxy and forwards requests to upstream servers over HTTP/1.x or HTTP/2, Angie can maintain the integrity of HTTP/3 all the way to compatible upstream servers. This capability ensures that the latency and performance benefits of HTTP/3 are not lost in proxy scenarios.

ANGIE's support of HTTP/3 and QUIC on both the client and server sides allows forwarding requests to upstream services over QUIC, and makes ANGIE a compelling alternative, so it was decided that ANGIE will be used as the Ingress Controller in the project's implementation.

5.2.4.2 Containerisation With Docker

The deployment of ANGIE began with the creation of a Docker container, which encapsulates all necessary components and configurations for the service to run independently of the underlying infrastructure. The image can be pushed to an image repository like Docker Hub, and it would then be ready to be pulled and used by Kubernetes deployments.

```
FROM debian:12
     ...
RUN apt-get update && apt-get install ...
     ...
COPY angie.conf /etc/angie/angie.conf
COPY cert.crt /etc/angie/cert.crt
COPY cert.key /etc/angie/cert.key
     ...
EXPOSE 443
```

Code Snippet 5.1 Containerising ANGIE with a Dockerfile

Base Image:

- The Dockerfile starts with Debian 12 as the base image, as recommended by the documentation.

Installing ANGIE & Relevant Modules:

- Essential packages like ca-certificates, curl, and lsb-release are installed to enable secure communications and repository management.

- ANGIE is then installed, along with additional modules such as “angie-module-geoip2” for geographical IP handling, and “angie-module-njs” for extended JavaScript support. Cleanup steps remove unnecessary files and packages to reduce the Docker image size and remove potential security vulnerabilities.

Import Configuration Files & Certificates:

- Custom configuration files and SSL certificates are copied into the Docker image. This includes angie.conf for server configuration, and SSL certificate files necessary for HTTPS communications over QUIC.

Exposing QUIC Port:

- The container exposes port 443 for QUIC traffic, as this is universally accepted as the “secure” port for HTTPS.

5.2.4.3 Deploying On Kubernetes

With ANGIE containerised, the next step involved deploying the service using the Kubernetes Deployments and Services resources.

```yaml
kind: Deployment
metadata:
  name: angie-ingress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: angie-ingress
  template:
    metadata:
      labels:
        app: angie-ingress
    spec:
      containers:
      - name: angie-ingress
        image: corneljonathan/angie-ingress:latest
        ports:
        - name: quic
          containerPort: 443
          protocol: UDP
```

Code Snippet 5.2 The ANGIE deployment resource declaration

- A Kubernetes Deployment is defined to manage the lifecycle of ANGIE pods. The deployment specifies that a replica of the ANGIE pod should be maintained at all times.

- A match label is applied to tag all pods running in this deployment as “angie-ingress”

- The deployment configuration ensures that the container image is pulled and run within the cluster. The container is configured to expose UDP port 443, to listen for QUIC traffic.

```
kind: Service
metadata:
  name: angie-ingress
spec:
  type: LoadBalancer
  selector:
    app: angie-ingress
  ports:
    - protocol: UDP
      port: 443
```

Code Snippet 5.3 The ANGIE service resource declaration

- A Kubernetes Service of type LoadBalancer is defined to distribute incoming traffic across the available ANGIE pods. The Service targets UDP port 443, aligning with the deployment’s listening configuration.

- The LoadBalancer type is chosen to expose the service externally. It provides a public IP address that external clients can use to access the service, making this the entry point for the cluster.

5.2.4.4 Configuring ANGIE

```
ssl_certificate      cert.crt;
ssl_certificate_key  cert.key;
proxy_ssl_trusted_certificate angie_cert.pem;
ssl_protocols TLSv1.3;
```

Code Snippet 5.4 Loading the needed SSL/TLS files into ANGIE

- **ssl_certificate:** Specifies the path to the SSL certificate file
- **ssl_certificate_key:** Specifies the path to the SSL certificate key file
- **proxy_ssl_trusted_certificate:** Specifies a trusted CA certificate for verifying the backend service’s certificate. As they will all be signed by ANGIE in this implementation, ANGIE itself is the CA
- **ssl_protocols:** Configures ANGIE to use TLS 1.3, as required by QUIC

```
resolver 10.96.0.10 valid=10s;
```

Code snippet 5.5 Specifying the IP of the Kubernetes resolver to ANGIE

- **resolver:** Sets the Kubernetes DNS Service's IP as the resolver for domain names, allowing ANGIE to resolve dynamic Kubernetes IPs

```
# Content Based Routing
map $http_content_type $destination {
    default "invalid";
    audio https://audio-svc.default.svc.cluster.local:8888;
    video https://video-svc.default.svc.cluster.local:8888;
}
```

Code Snippet 5.6 The "Map" block is used to set the correct destination, based on the content type

- **map $http_content_type $destination:** this block checks the value of the "content-type" HTTP header which ANGIE can access, and sets the corresponding user defined "destination" value, which is used in the next block.

```
# Listen for QUIC and HTTP3
listen 443 quic reuseport;
proxy_http_version 3;
# ...
# Route based on destination value
location / {
    if ($destination = "invalid") {
        return 400;
    }


    if ($destination != "invalid") {
        proxy_pass $destination;
    }
}
```

Code Snippet 5.7 Setting up the QUIC listener and configuring the forward routing rules

- **listen 443 quic reuseport:** Sets the Kubernetes DNS Service's IP as the resolver for domain names, allowing ANGIE to resolve potentially dynamic Kubernetes IPs
- **proxy_http_version 3:** Directs ANGIE to use HTTP/3 when proxying to upstream services.
- **location / { ... }**: Defines routing rules for the "/" path within this block
- **if ($destination = "invalid") { return 400; }:** Checks if the destination set earlier by the "map" block is "invalid", and returns a 400 Bad Request error
- **proxy_pass $destination:** If the destination is valid, ANGIE proxies the request to IP in the destination variable, which is resolved by sending a DNS query to the previously mentioned resolver

5.2.4.5 Ingress Implementation

With ANGIE containerised, deployed, and configured, the Ingress Controller is finally ready for use. Below is an outline of all the previous components working together.

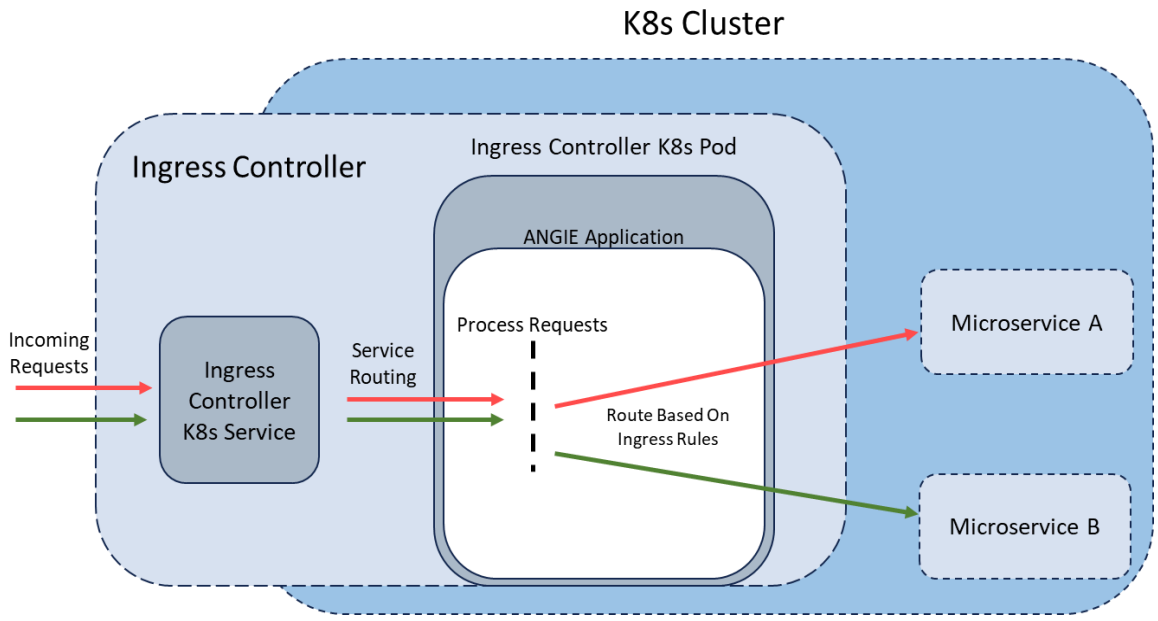

Figure 5.1 Final Ingress Implementation - The Ingress Controller Service receives requests, which get routed to the Ingress Controller Pod running the ANGIE Ingress Controller Application. This in turn processes the requests and forwards them to the appropriate microservice

Incoming Requests:

- An external client sends a QUIC request to the public IP address of the cluster.

Service Routing:

- The service receives the request on port 443 and routes it to one of the available pods based on the selector configuration, which matches the labels of the pods running ANGIE.

Pod Processing:

- The request arrives at the designated pod's container, where ANGIE is running and listening
- ANGIE processes the request based on the ingress rules specified in the configuration file.

Ingress Rules Execution:

- ANGIE evaluates the ingress rules defined within its configuration file to determine the appropriate action for the request, such as forwarding it to an appropriate internal service, applying transformations, or handling security checks.
- Based on these rules, ANGIE routes the request to the correct internal application or service endpoint within the Kubernetes cluster.

The above implementation results in a functional Ingress Controller that has all the functionalities outlined in section 4.3.2.

5.3 Implementing The Client

The client's primary function is to establish a connection with the cluster's ingress controller, manage data streaming, and ensure accurate delivery through checksum validation. This chapter delves into the technical implementation of the client, built on the "aioquic" library, a Python framework that facilitates QUIC protocol handling. By leveraging aioquic, the client handles the complexities of QUIC connections, stream management, and HTTP/3 communications, which are critical for interacting with the modern web infrastructure provided by Kubernetes.

The client is designed to operate asynchronously, by also using Python's "asyncio", which excels at handling asynchronous I/O-bound applications (Solomon, 2019.). This asynchronous model is essential for managing multiple data streams efficiently, particularly when dealing with real-time data such as video and audio streams.

We begin by exploring the aioquic library's capabilities and how the client utilises these to interact with the Kubernetes cluster. Following this, the connection process to the cluster is outlined, emphasising the setup and configuration required for secure and reliable communication. Finally, the mechanism of sending HTTP/3 requests is detailed, demonstrating the client's role in a practical scenario involving the manipulation of video and audio data streams. Each section aims to provide a deep understanding of the client's functionality and its integration within the overall system architecture.

It's important to note here that since the client is outside of the Kubernetes Cluster, it does not get containerised using docker, or deployed using Kubernetes. It is simply run as a script.

5.3.1 "Aioquic" Client Programming Features

The "aioquic" library is a key tool in the Python ecosystem for implementing the QUIC protocol, providing capabilities that align with the project's requirements for the client component. This section explores how the client leverages the aioquic library's features to manage communication with a Kubernetes cluster.

The foundation of the client's interaction with the Kubernetes cluster begins with the instantiation of a "QuicConnectionProtocol" derived class, specifically tailored to handle QUIC connections and HTTP/3 communications. This subclassing allows the client to encapsulate all the logic related to connection management, stream handling, and data transmission, ensuring a modular and clear structure.

```
class UploadClientProtocol(QuicConnectionProtocol):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.h3_connection = H3Connection(self._quic)
```

Code Snippet 5.8 Showing the inheritance of the "UploadClientProtocol"

The “UploadClientProtocol” class plays a fundamental role in managing QUIC connections by extending “QuicConnectionProtocol”, a base class provided by the aioquic library. This inheritance is crucial as it allows the client to integrate and extend the functionalities required to handle QUIC and HTTP/3 protocols effectively. By inheriting from “QuicConnectionProtocol”, “UploadClientProtocol” gains several capabilities such as QUIC event handling and the ability to manage streams, as well as access to methods to maintain and close the existing connection gracefully.

The “super()” call initialises the base class, setting up the necessary environment for the QUIC connection to function. This includes configurations for the QUIC protocol itself, such as transport parameters, cryptographic configurations, and event handling mechanisms. Once the base class is initialised, the “UploadClientProtocol” class specifically creates an H3 Connection instance. This instance is crucial as it represents the bridge between the QUIC transport layer and the HTTP/3 application layer. The self._quic object passed to the H3Connection is a reference to the underlying QUIC connection. This connection manages all the lower-level transport details, such as packet transmission, flow control, and congestion control, while the H3Connection handles higher-level protocol features specific to HTTP/3.

One of the standout features of aioquic is its stream management capabilities, which the client utilises to handle concurrent data streams efficiently. When the client sends HTTP requests, it retrieves a unique stream ID for each new request using the QUIC connection’s “get_next_available_stream_id()” method. This ensures that each HTTP/3 request is correctly multiplexed over the same QUIC connection.

```
stream_id = self._quic.get_next_available_stream_id(
            is_unidirectional=False)
self.h3_connection.send_headers(
           stream_id=stream_id,
           headers=headers,
           end_stream=False)
self.h3_connection.send_data(
            stream_id=stream_id,
            data=stream,
            end_stream=True)
```

Code Snippet 5.9 Using “aioquic” to create a new stream, by getting the next stream ID and sending data

The use of “send_headers()” and “send_data()” methods from the H3Connection object allows for precise control over the HTTP headers and the flow of data. The ability to flag the end of headers and the end of data with the “end_stream” flag provides flexibility and control in how data is transmitted.

```
def quic_event_received(self, event):
    if isinstance(event, StreamDataReceived):
        # Process stream data...
```

Code Snippet 5.10 The beginning of the QUIC event processing block, checking for Stream events

Aioquic also provides mechanisms for handling various events that occur during the lifecycle of a QUIC connection. By overriding methods like "quic_event_received()", the client can customise its response to different events, such as data reception or connection termination, enhancing the responsiveness of the application. This event-driven model is essential for high-performance networking applications, allowing the client implementations to handle asynchronous network events efficiently. This model is particularly effective in environments where quick response times and non-blocking operations are critical.

In summary, aioquic provides a comprehensive suite of tools through classes like the "QuicConnectionProtocol" class, that will enable the client to establish and manage a QUIC connection, handle multiple data streams, and respond to network events dynamically. By extending the "QuicConnectionProtocol", the implementation of "UploadClientProtocol" can make use of these capabilities, and can be tailored to meet the specific requirements of generating and sending HTTP/3 requests to the Kubernetes Cluster.

5.3.2 Connecting To The Cluster

To establish a secure and efficient connection to a Kubernetes cluster, the client must first configure the QUIC protocol parameters. This setup is encapsulated within the QuicConfiguration class provided by the aioquic library, which serves as a blueprint for the QUIC connection. The configuration of each parameter plays a vital role in ensuring that the connection adheres to required security standards and performance expectations.

```
configuration = QuicConfiguration(
        is_client=True,
        alpn_protocols=H3_ALPN,
        max_data=900000000,
        max_stream_data=90000000,
        secrets_log_file=open(".\\pmsl.log", "a"),
        quic_logger=QuicFileLogger(".\\qlog")
    )
    configuration.load_verify_locations(cafile='./root-cert.crt')
```

Code Snippet 5.11 Configuring the QUIC connection with the correct CA and setting the log file paths

- **is_client=True:** Specifies that the configuration is for a client-side connection.
- **alpn_protocols:** This list defines the Application-Layer Protocol Negotiation protocols the client supports, crucial for ensuring compatibility with the server's protocols.

- **max_data and max_stream_data:** These settings control the maximum amount of data that can be sent over the connection and per stream, respectively, important for flow control and avoiding congestion.
- **secrets_log_file and quic_logger:** These parameters are used for debugging and logging the QUIC connection, aiding in troubleshooting and network visualisation with "qvis" (Marx et al., 2020).
- **load_verify_locations:** This method is critical for security, as it loads the certificate authority's certificate to verify the server's identity. The certificate used designates the ANGIE ingress controller as a CA, which means the client can trust certificates signed by it.

```
async with connect(host, port, configuration=configuration,
create_protocol=UploadClientProtocol) as protocol
```

Code Snippet 5.12 Using the aioquic "connect()" function with the custom "UploadClientProtocol"

The "connect()" function is a high-level convenience method provided by aioquic that abstracts the QUIC connection establishment process. The client uses this function to connect to the Kubernetes Controller.

It involves:

- **DNS Resolution:** If a hostname is provided, aioquic resolves it to an IP address. This step is essential for determining the correct server location in the network.

- **The QUIC Handshake:** The most critical phase of the connection setup, where cryptographic parameters are exchanged between the client and the server. The handshake process involves:
  - **ClientHello:** The client sends a message with supported cipher suites, a randomly generated number for session keys, and other cryptographic parameters.
  - **ServerHello:** The server responds with its selection of cryptographic parameters, including the cipher suite and session keys.
  - **Cryptographic Establishment:** Using the parameters exchanged, both client and server generate session keys for encrypted communication.

- **Stream Initialisation:** Simultaneously with the handshake, aioquic begins initialising streams. This is where the "max_data" and "max_stream_data" configurations play their roles in setting the bounds for data transmission.

With the connection securely established and all protocol parameters configured, the client can proceed to utilise the full capabilities of HTTP/3, such as initiating multiple concurrent data streams, and is well-positioned to interact with the Kubernetes cluster effectively and securely. This completion sets the stage for the next crucial phase of the project: sending HTTP/3 requests.

5.3.3 Sending Client HTTP/3 Requests

Once a QUIC connection is successfully established and configured, the client utilises this secure and efficient channel to send HTTP/3 requests to the Kubernetes cluster. This process involves the transmission of various types of data, which, in this implementation, includes splitting and sending separate video and audio streams. This approach exemplifies the advanced capability of HTTP/3 to handle multiple data types concurrently over a single connection.

Before the client can send HTTP/3 requests to the Kubernetes cluster, it must prepare the data it intends to transmit. This preparation involves taking video and audio files from local storage, converting them into binary streams, and then using these streams to create HTTP/3 requests. This process is critical as it sets the foundation for effective stream management and data segregation, ensuring that each stream is correctly identified and routed within the Kubernetes architecture.

The "get_split_video_streams()" is a function that handles the extraction and conversion of video and audio content into binary streams. This function is designed to separate the video and audio components of a media file, enabling independent handling and transmission of each stream.

Once the streams are prepared, the client begins creating the HTTP/3 request streams. Each stream is tagged with a specific content-type header that corresponds to the nature of the data (e.g., "video" for video streams and "audio" for audio streams). This tagging is crucial as it fulfils the client's requirement to correctly label streams in a multiplexed connection, enabling the ingress controller's to accurately split and direct these streams to the appropriate microservices.

```
await asyncio.gather(
            self.send_http_request('video', video_stream),
            self.send_http_request('audio', audio_stream),
        )
```

Code Snippet 5.13 Using the asyncio "gather()" function to concurrently send the HTTP requests

The client uses asyncio's "gather()" to send both video and audio streams concurrently, leverages the multiplexing capabilities of HTTP/3, allowing multiple data streams to be transmitted over the same QUIC connection concurrently, without interference.

The "send_http_request()" function orchestrates HTTP/3 request transmission over a QUIC connection by first constructing necessary HTTP headers, setting method to "POST", scheme to "https", authority as "angie-ingress", path as the root location"/", and "content-type" to either "video" or "audio". It then retrieves a unique stream ID for sending headers and the binary data stream sequentially, with the final transmission marked by end stream being set. This function also sets up asynchronous response handling to manage server feedback and potential errors efficiently, ensuring the integrity and responsiveness of the data exchange process.

After sending the requests, the client handles responses from the Kubernetes cluster. These responses should confirm the receipt of the data and provide checksum validations. The client manages these responses asynchronously, ensuring each stream's lifecycle is effectively handled from initiation through to successful completion.

5.3.4 Validating Responses

After sending HTTP/3 requests through the established QUIC connection, the client engages in a response validation process. This round-trip validation is crucial in the context of this project, for demonstrating that the cluster's routing and processing mechanisms are functioning as expected, providing a check beyond QUIC's inherent data integrity features.

```
async def stream_handler(self, reader: asyncio.StreamReader, writer,
stream_id):
        try:
            await reader.readuntil(b'{"checksum":"')
            checksum = await reader.readuntil(b'"')
            checksum = checksum[:-1].decode()
            print("CHECKSUMS MATCH:",
                checksum == self.expected_result[stream_id])
           self.response_futures[stream_id] .set_result(checksum)
        except asyncio.IncompleteReadError as e:
            print("Incomplete Read:", e.partial)
            print("STREAM ID:", stream_id)
```

Code Snippet 5.14 The client stream handler verifying the server correctly received the packets by checking the servers calculated checksum of the content

The client's response handling mechanism involves:

- Listening for StreamDataReceived events to capture and analyse incoming stream responses. One response should be received per request, containing a checksum generated by the microservice, which is checked against an expected checksum, which was calculated by the client before sending the request. The checksum is then used to mark the handler's future as complete, for the sake of program termination.

- Managing errors and anomalies that might occur during data transmission, such as incomplete reads of a multipacket response. The client logs detailed information about any discrepancies in data or unexpected behaviours in the transmission process. This facilitates debugging and helps maintain high reliability and performance standards.

5.3.5 Summary Of Client Implementation

The client's implementation is a comprehensive approach to establishing and managing efficient communication with a Kubernetes cluster's ingress controller, using the QUIC protocol implementation facilitated by the aioquic library. Through the use of asynchronous programming with Python's asyncio library, the client effectively manages multiple data streams, ensuring non-blocking operations and optimal use of network resources. By focusing on the specific requirements set out in the Design chapter, the client implementation is able to perform essential functions such as connection setup, data streaming, and checksum validation.

5.4 Implementing The Microservices

The microservices' primary function is to efficiently process and respond to routed data packets within a Kubernetes cluster, utilising the advanced capabilities of HTTP/3 and QUIC protocols for high-performance communication. This chapter delves into the technical implementation of the microservices, built on the foundation of the aioquic library and Starlette, a Python framework which provides the Asynchronous Server Gateway Interface (ASGI) programming interface necessary for building responsive web applications.

Similarly to the client implementation, the microservices also leverage the asynchronous programming features of Python's asyncio library, enabling the microservices to handle I/O-bound and network-intensive operations efficiently.

This section will explore the capabilities of aioquic and Starlette and how they interact within the Kubernetes environment to handle HTTP/3 communications. The architecture of the microservices, designed around the Starlette framework, allows for the scalable and resilient handling of incoming requests and routing of packets to appropriate services. This setup ensures that the microservices can maintain high performance and reliability even under high loads. The final section will outline the detailed process of containerising the microservices using Docker and their deployment using Kubernetes, aligning with the project goals.

5.4.1 Libraries & Technologies Used

In this section, we will delve into how each library contributes to the system's architecture, focusing on their specific roles and functionalities in the context of QUIC server programming and ASGI application development.

5.4.1.1 Aioquic For QUIC Management

As outlined in Section 5.3.1, the choice of aioquic for server-side programming stems from its extensive support for QUIC features, including connection migration, stream multiplexing, and full TLS 1.3 encryption, which are essential for maintaining secure and efficient client-server connections. These capabilities extend to the server side as well, making aioquic a valuable tool to use for programming the server side.

Aioquic's efficient management of connection overhead not only reduces latency but also conserves server resources, allowing for higher throughput. Additionally, the server-side implementation benefits from aioquic's comprehensive handling of QUIC's built-in security features, like TLS 1.3, which simplify the complexities involved in securing communications. Moreover, aioquic's support for HTTP/3 server push technologies allows servers to preemptively send responses to clients, a significant advantage in optimising web content delivery.

5.4.1.2 Starlette For ASGI Programming

Starlette's primary appeal for use in microservices lies in its simplicity and direct approach to handling both synchronous and asynchronous requests efficiently. This characteristic is crucial for services that operate under high concurrency, ensuring that server resources are used optimally without unnecessary overhead.

A key feature of Starlette is its routing system, which is intuitive and easy to use. This is essential for microservices where different endpoints might be handling varied types of data transactions or user requests. Furthermore, Starlette supports WebSocket connections out of the box, enabling real-time, bidirectional communication between the clients and the server, a necessity for applications like live data feeds or interactive platforms.

These utilities make Starlette a practical choice for microservices architectures, especially when combined with its ability to handle a large number of simultaneous connections efficiently. Its lightweight execution model ensures that microservices built with Starlette can quickly respond to client requests, managing data processing tasks without blocking critical operations.

5.4.2 ASGI Architecture

The implementation of the microservices utilises the ASGI architecture, which is specifically designed to handle asynchronous applications and is fundamental in enabling efficient, non-blocking communication. ASGI serves as the standard interface between asynchronous Python web servers and applications. By leveraging ASGI, the microservices can manage and process multiple requests simultaneously, making the architecture highly scalable and responsive.

Utilising the ASGI architecture not only facilitates efficient and non-blocking communication but also significantly eases the development process. ASGI inherently supports the separation of concerns, a principle critical in software engineering designed to organise code in a way that each module or layer handles a specific, independent function.
This approach is especially beneficial in microservices architecture, where each service's ASGI application can be developed, maintained, and scaled independently.

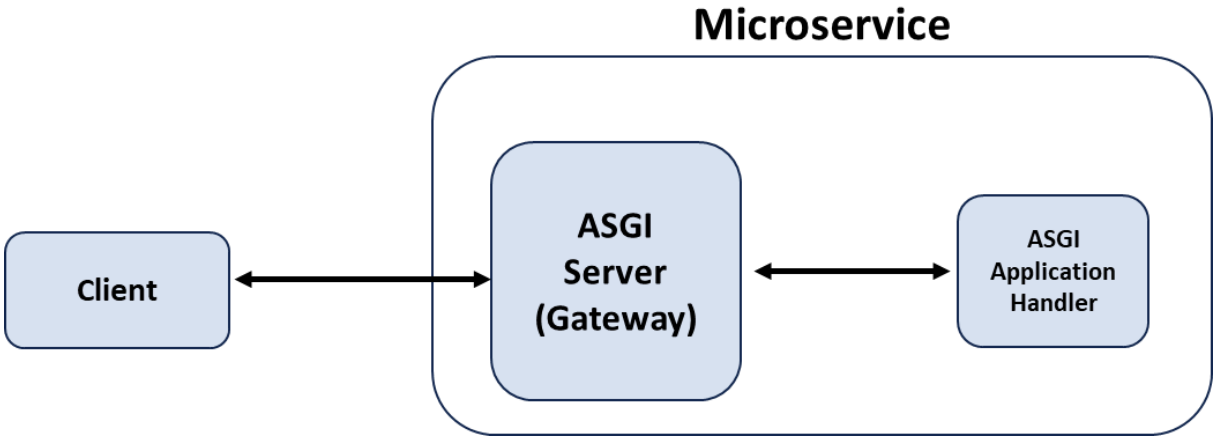

Figure 5.2 Overview Of ASGI Architecture - The ASGI Server relays requests to the ASGI Application Handler, which processes them and returns the response

5.4.2.1 The ASGI Server

The ASGI server plays a pivotal role as the central coordinator for all incoming and outgoing web communications. Utilising asyncio and aioquic, the server asynchronously handles HTTP/3 requests leveraging the QUIC protocol's capabilities. This setup allows for handling multiple, concurrent client interactions.

5.4.2.1.1 Receiving Requests

The handling of incoming requests is efficiently orchestrated through a structured process within the implemented HttpServerProtocol class, which also extends the "QuicConnectionProtocol" class, giving it access to useful aioquic functions, like the "quic_event_received()" function.
This process leverages the aioquic library's capabilities to manage each request as a discrete stream, ensuring that requests are handled promptly and efficiently. Here's the detailed mechanism through which the server processes incoming HTTP/3 requests:

Identification of New Requests:
Each incoming HTTP/3 request is recognised as a stream, uniquely identified by a stream_id. When the HeadersReceived event is triggered by aioquic, it signals the arrival of new headers for a particular stream. The server first checks if this stream_id is already associated with an existing request handler. If not, it indicates a new incoming request, prompting the server to initiate processing.

Header Processing and Request Setup:
Upon identification of a new request, the server parses the headers accompanying the HTTP/3 request. Key HTTP headers such as :method, :path, :authority, and :protocol are extracted. Query strings are also extracted if present.

These headers are crucial as they:

- Define the HTTP method being used, which influences how the request is handled.
- Indicate the path being requested, which is necessary for ASGI routing
- Provide the authority or host information, which would be necessary for resolving virtual hosts or handling requests across multiple domains, but is not needed in this implementation

Scope Definition and Handler Initialisation:
A scope dictionary is constructed, encapsulating all essential data about the request, including client address details, headers, and HTTP version. An instance of "HttpRequestHandler" is then initialised with this scope, along with the connection and protocol context. This handler is responsible for the lifecycle of the request, managing everything from further data reception, to sending the responses.

Asynchronous Task Execution:
Finally, the request handler is scheduled as an asynchronous task using asyncio.ensure_future. This function ensures that the handler's run_asgi method is executed, allowing it to interact with the ASGI application to process the request and generate an appropriate response.

When subsequent events are received for an existing handler, the handler adds the event to an asyncio Queue, where it will be retrieved from by the ASGI application when it is ready to process that event.

5.4.2.2 The ASGI Application

The ASGI application is where the business logic of the microservices resides. The application utilises Starlette to set up two simple routes that directly correspond to the project's requirements. The primary functionality needed for this project involves responding to basic health checks and processing uploads to generate and return checksums.

5.4.2.2.1 Responding To Requests

The ASGI application sets up two basic endpoint handlers:

```
async def healthcheck(request):
            if   (request.headers['content-type']   ==   "video")   or
(request.headers['content-type'] == "audio")  :
        return PlainTextResponse('OK', status_code=200)
    return PlainTextResponse("Bad Request", status_code=400)
```

Code Snippet 5.15 The "healthcheck()" endpoint handler responds with '200' for valid content types

Health Check Handler: The application defines a "healthcheck()" function that responds to GET requests. This endpoint checks if the incoming request's content type is either "video" or "audio". If so, it returns a 200 OK response, indicating that the service is operational and ready to handle relevant media types. If the content type does not match, it responds with a 400 Bad Request, indicating an unsupported request type.

```
async def upload_handler(request: Request):
    hash_obj = hashlib.sha256()
    async for chunk in request.stream():
        hash_obj.update(chunk)
    checksum = hash_obj.hexdigest()
    print(f"checksum: {checksum}")
    return JSONResponse({"checksum": checksum}, status_code=201)
```

Code Snippet 5.16 The "upload_handler()" asynchronously calculating a checksum of the received data

Upload Handler: The upload_handler() function manages POST requests aimed at handling data uploads. It processes the data stream asynchronously, calculating a SHA-256 checksum of the received data. This checksum acts as a unique identifier for the data, ensuring integrity and successful upload. Once the data stream is completely processed, the checksum is returned to the client in a JSON response, confirming the receipt and integrity of the uploaded data.

5.4.3 Containerisation & Deployment

This section describes the steps involved in containerising and deploying the audio microservice using Docker and Kubernetes, ensuring the application is scalable and manageable within a practical environment.

5.4.3.1 Containerising With Docker

The first step in deploying the microservice involves creating a Docker container, which encapsulates all necessary components, including the application code and its dependencies. The Dockerfile uses python:3.9-slim base image, as this choice offers a balance between having a small image size and retaining essential functionalities required by Python applications. The working directory within the container is set to /app. This directory serves as the default location for all operations performed in the container.

```
COPY server_cert.pem server_cert.pem
COPY server_key.pem server_key.pem
COPY server.py server.py
COPY asgi_app.py asgi_app.py

RUN pip install aioquic[http3] wsproto starlette jinja2
```

Code Snippet 5.17 Containerising the microservice code with a Dockerfile

Essential files including the ASGI's server's code, the ASGI application, and SSL certificates (server_cert.pem and server_key.pem) are copied into the container. This ensures that the server has all the files it needs to operate. Afterwards, necessary Python packages that provide support for HTTP/3 and ASGI web functionalities are installed.

Once the Docker image is built and pushed to a registry such as Docker Hub, the next step involves deploying it within a Kubernetes environment.

5.4.3.2 Deploying In Kubernetes

A deployment named audio-svc-deployment is defined to manage the lifecycle of the pods running the audio service. The deployment ensures that one replica of the pod is maintained at all times, supporting the audio microservice's availability. It also exposes the chosen application port, 8888, over UDP.

```
volumeMounts:
        - name: qlogs-volume
          mountPath: /app/qlog/
```

Code Snippet 5.18 Configuring the microservice deployment to mount the .qlog files

The deployment configures volume mounts to facilitate the storage and management of QUIC log files, qlogs, which are crucial for analysing and visualising QUIC networks using tools like qvis. The microservices generate these qlogs at /app/qlog/ as they receive network data. This path is mounted to a volume that maps to a physical location on the host machine, ensuring that log files are preserved beyond the lifecycle of individual pods and can be accessed directly from the host for analysis.

The service declaration for the audio service is defined to facilitate internal communication within the Kubernetes cluster. This service is created as a ClusterIP service, which restricts access to the service within the cluster network, making it invisible to the external network. This setup is ideal since the audio service only needs to communicate with the ingress controller and other internal components. The service configuration specifies that it listens on UDP port 8888, which corresponds to the port exposed by the containers running the audio service. This ensures that all traffic intended for this service is appropriately routed to the containers based on the labels specified in the selector. By using a ClusterIP, the configuration promotes better security and resource management, reducing exposure to external threats and unnecessary resource allocation.

5.4.4 Summary Of Microservice Implementation

The core of the system's functionality resides in the ASGI server, where incoming HTTP/3 requests are processed and routed using aioquic. This setup allows for concurrent handling of multiple data streams, enhancing throughput and responsiveness. Each request is managed independently, ensuring that high loads do not hinder system performance. The server setup includes advanced features such as connection migration, stream multiplexing, and full TLS 1.3 encryption, ensuring secure and efficient data handling. The ASGI application component, built with Starlette, simplifies the routing and handling of requests with minimal overhead, supporting real-time data processing with capabilities like WebSocket integration.

The microservice application specifically caters to the project's needs by implementing two main functionalities: health checks and data upload handling. Health checks ensure the service's readiness to handle specified media types, while the upload handler processes incoming data streams to generate and return checksums, confirming the integrity of uploads. In summary, this implementation meets the requirements set out in Section 4.3.3.

Chapter 6
Evaluation

This chapter assesses the implementation's operational functionality, to provide a comprehensive understanding of how the system meets its design goals and performs under simulated conditions. It also compares it against other potential solution models, and analyses the proposed solutions scalability and reliability.

6.1 Demonstrating The System

Demonstrating the operational functionality of the system will be helpful in objectively evaluating the implementation's success, as well as limitations. To do this, we will explain the setup used and then verify the successful operation of the system by verifying results through the logs of the various components.

6.1.1 Setup Overview

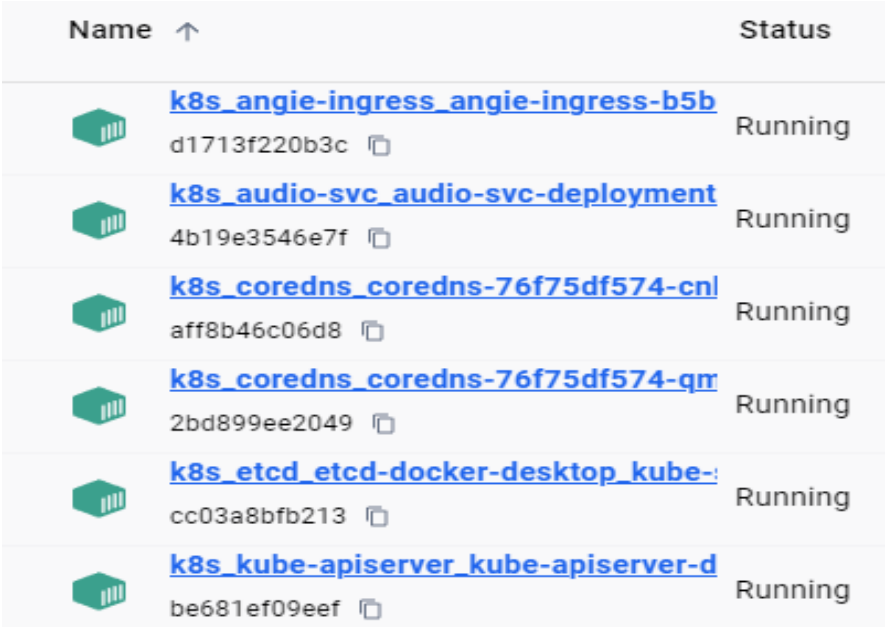

Image 6.1 The Running System - The Kubernetes Cluster components running inside of Docker Desktop, using "k8s" naming scheme, along with the user defined services and deployments.

The Kubernetes cluster is running locally on Docker Desktop, which simulates a production environment for development and testing purposes. We can see containers with the name “k8s_POD_X-deployment”, which run the Pods managing the deployment of the “X” application, ie, the audio-svc or angie-ingress. We can also see the containers using the “k8s_X_-[unique-identifier]” naming pattern, which are the actual containers running “X” application. This indicates that all the cluster components are healthy and running.

- **Ingress Controller Deployment:** The ingress controller, specifically ANGIE, is deployed within the Kubernetes cluster and configured to act as the entry point for all incoming traffic. It is exposed externally via a LoadBalancer service, facilitating access from the internet.

- **Microservices Deployment:** Dedicated microservices for handling distinct data types, specifically video and audio services, are deployed. These services are accessible within the cluster through a ClusterIP service, which ensures that they can communicate internally without exposure to the external network.

- **Logging and Monitoring:** Each microservice is configured to log its network activity via qlog, which captures detailed information about the QUIC network transactions. This setup is critical for later analysis and verification of the network operations.

- **System Readiness:** The Kubernetes dashboard shows all pods in a running state, indicating that the system setup is complete and operational.

6.1.2 Analysing System Logs

Starting the developed Client script initiates the communication process. During execution, the client script outputs logs that detail each stage of its operation, providing real-time feedback on its progress and actions.

```
Uploading:  video
Done uploading: video
Uploading:  audio
Done uploading: audio
Stream ID: 4 - CHECKSUMS MATCH: True
Stream ID: 0 - CHECKSUMS MATCH: True
TERMINATING
```

Image 6.2 Successful Client Program Execution - Logs outlining correct uploading of streamed

- **Data Stream Upload and Response:** The client successfully uploads the data streams to the ingress controller, which should route them to the appropriate microservices based on the content type. Each microservice processes its respective stream and sends a response back to the client

- **Checksum Verification**: Upon receiving responses, the client verifies the checksums provided by the microservices, confirming the integrity and accuracy of the data transmission and processing.

To verify the behaviour of the network, tools like Wireshark are usually used to inspect network traffic. However, using qlogs generated from both the client and the services, the network traffic can be visualised using qvis, providing a better analysis of network behaviour (Marx , 2020).

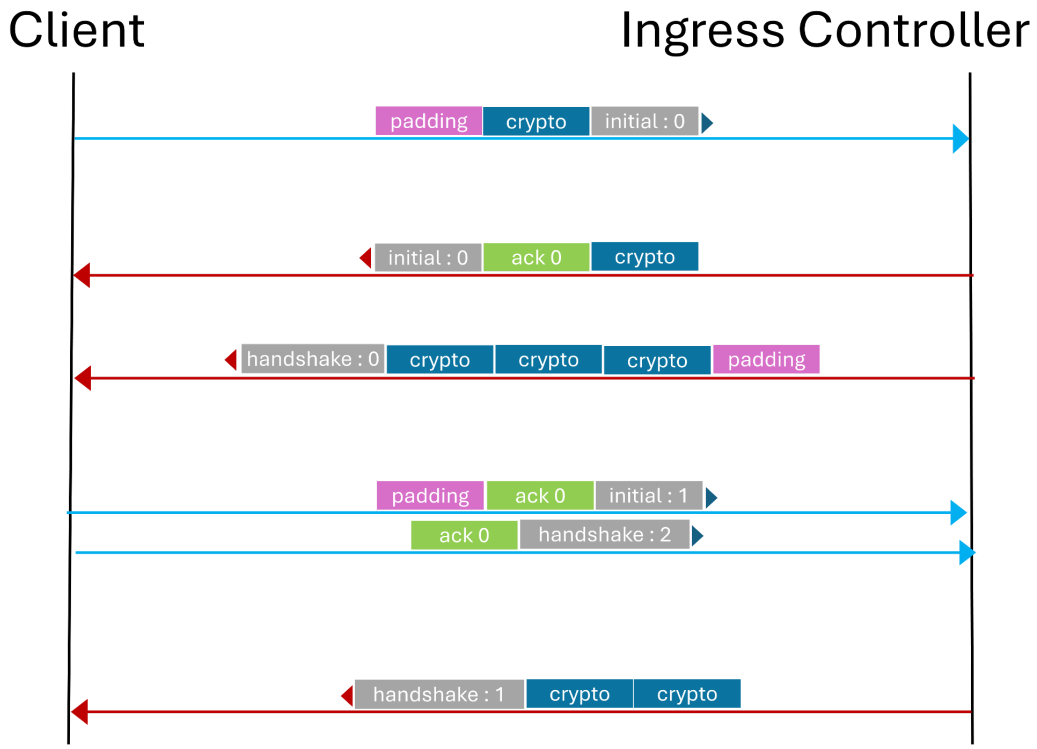

Figure. 6.1: Client-Ingress QUIC Handshake - Initial packets are sent, followed by several crypto packets containing version negotiation and cryptographic information to establish the session keys

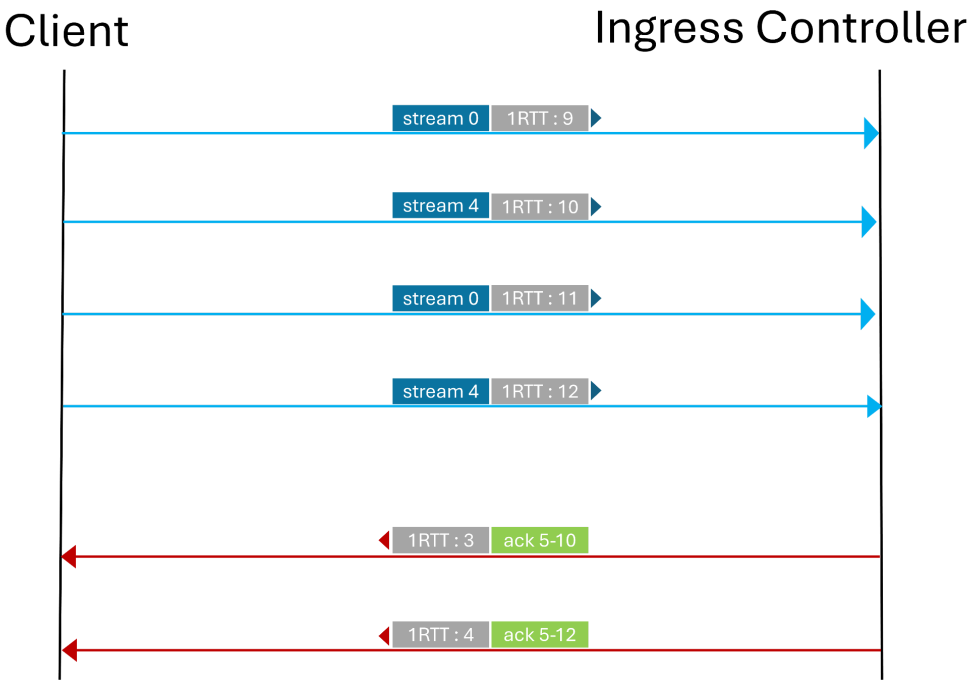

Figure 6.2 Independent Client Streams - The client sends packets for different streams, and the server is sending cumulative ACKs

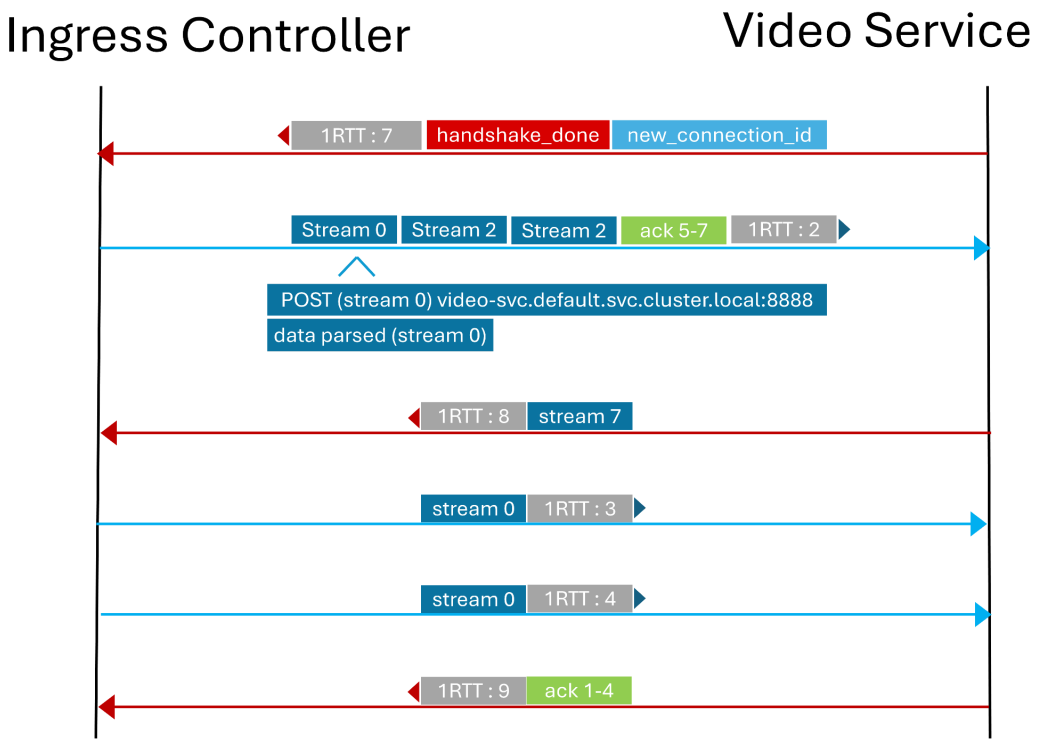

Figure 6.3 Video Microservice Receiving Data - A video POST request arrives on stream 0, which was stream 0 on the client side

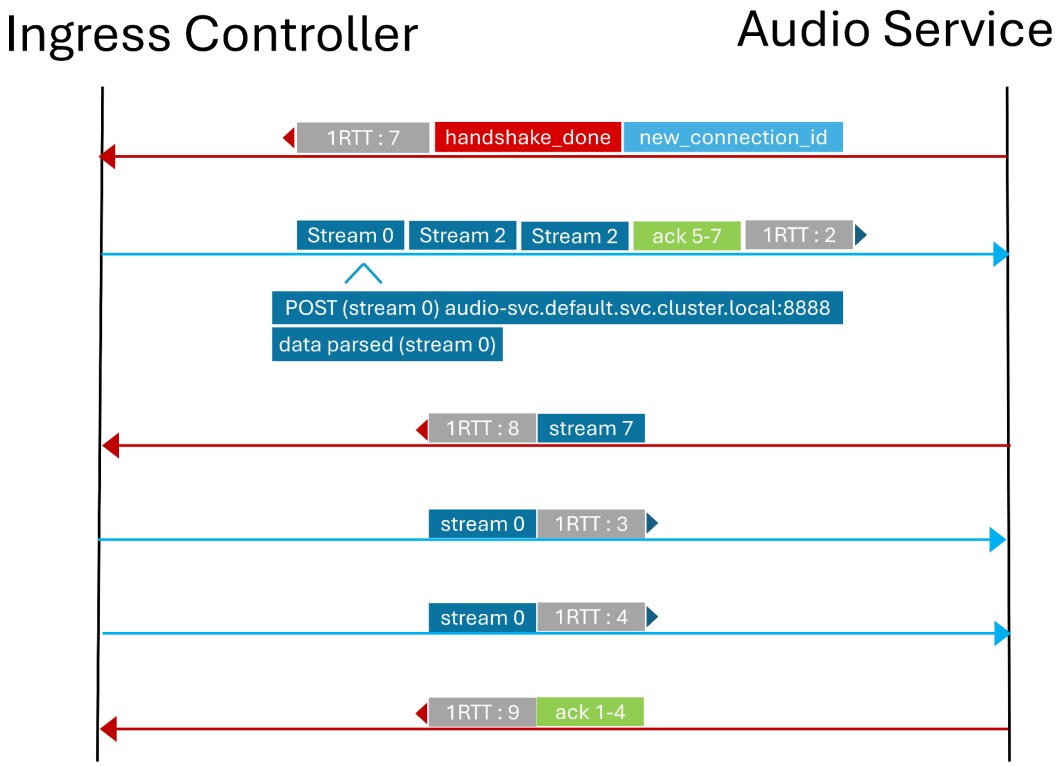

Fig. 6.4: Audio Microservice Receiving Data - An audio POST request arrives on stream 0, which was stream 4 on the client side, but this is a new connection between the ingress and the microservice, so the stream numbering is reset

The ANGIE Ingress Controller can also have its logs checked, which record the responses that are sent by the ingress controller for forwarded requests. It would be expected that two “201” responses are returned, as the client is uploading a video and audio stream to be stored by the server. Checking the ANGIE logs verifies this behaviour, and depending on the log level ANGIE is configured to use, can provide further evidence that the system is working as required and expected.

```
2024-04-15 09:45:25 192.168.65.3 - - [15/Apr/2024:08:45:25 +0000] "POST / HTTP/3.0" 201 79
2024-04-15 09:45:33 192.168.65.3 - - [15/Apr/2024:08:45:33 +0000] "POST / HTTP/3.0" 201 79
```

Image 6.3 Successful Responses - ANGIE logs outlining successful “201” responses to the client, indicating successful request processing by the microservices

```
2024/04/15 08:53:01 [debug] 7#7: *140 http3 header: "content-type: application/json"
2024/04/15 08:53:01 [debug] 7#7: *140 http3 header len:94
2024/04/15 08:53:01 [debug] 7#7: *140 http3 output header: ":status: 201"
2024/04/15 08:53:01 [debug] 7#7: *140 http3 output header: "server: Angie/1.5.0"
2024/04/15 08:53:01 [debug] 7#7: *140 http3 output header: "date: Mon, 15 Apr 2024 08:53:01 GMT"
2024/04/15 08:53:01 [debug] 7#7: *140 http3 output header: "content-type: application/json"
2024/04/15 08:53:01 [debug] 7#7: *140 http3 output header: "content-length: 79"
2024/04/15 08:53:01 [debug] 7#7: *140 write new buf t:1 f:0 0000564D26F39B70, pos 0000564D26F39B70, size: 2 file: 0, size: 0
2024/04/15 08:53:01 [debug] 7#7: *140 write new buf t:1 f:0 0000564D26F39AC0, pos 0000564D26F39AC0, size: 61 file: 0, size: 0
2024/04/15 08:53:01 [debug] 7#7: *140 write new buf t:1 f:0 0000564D26F39BC8, pos 0000564D26F39BC8, size: 3 file: 0, size: 0
2024/04/15 08:53:01 [debug] 7#7: *140 http write filter: l:0 f:0 s:66
2024/04/15 08:53:01 [debug] 7#7: *140 file cleanup: fd:3
```

Image 6.4 ANGIE In Detail - Detailed log showing ANGIE preparing to forward a reply from a backend microservice to the client, by creating a new HTTP/3 response and setting its headers.

6.1.3 Analysis Of Requirement Satisfaction

Upon analysis, the system’s ability to handle and separate multiplexed HTTP/3 streams has been demonstrably achieved. The implementation facilitates precise routing of data streams at the ingress controller, which effectively identifies stream content types, ensuring that each stream can be dispatched to a backend service designed to process that particular stream type. This strategy showcases a significant enhancement in routing efficiency, which is a critical requirement for microservices which demand swift and reliable communication between services.

While the system meets core requirements, certain enhancements could be explored to address the broader goals of microservice architectures. For instance, scalability tests to validate how the system performs under increased loads are not explicitly demonstrated. Future iterations could benefit from incorporating load balancing tests, resilience checks, and more extensive scalability evaluations.

6.2 Comparative Analysis

While conducting the comparative analysis, it becomes apparent that the utilisation of QUIC for managing multiplexed HTTP/3 connections was not only innovative but also singular in its approach. This uniqueness presented a challenge in finding a directly comparable, pre-existing system that operated under the same premises as the implementation outlined in this project.

More precisely, as previously outlined in Section 5.2.1, current Ingress Controller implementations have yet to fully exploit the benefits of QUIC, such as supporting QUIC on the server facing side. Consequently, a viable comparison of the implementation proposed in this paper, “The Proposed System”, can only be made against two theoretical implementations:

Theoretical Implementation A): “The Legacy System”

- It has the same three basic components as the implementation outlined in this paper: a client, the ingress controller, and a set of microservices.
- This implementation has no support for HTTP/3 or QUIC.
- It therefore uses HTTP/2 (which has limited multiplexing capabilities) over TCP on the client side and on the server side.
- It uses a similar approach for splitting streams as the approach outlined in this paper, ie, checking the HTTP “content-type” header of the HTTP/2 requests.

Theoretical Implementation B): “The Transitioning System”

- It has the same three basic components as the implementation outlined in this paper: a client, the ingress controller, and a set of microservices.
- This implementation has experimental support for HTTP/3 or QUIC, allowing it to accept HTTP/3 connections on the client side.
- It still uses HTTP/2 over TCP on the server side, due to the microservices on the server side not supporting QUIC.
- It uses a similar approach for splitting streams as the approach outlined in this paper, ie, checking the HTTP “content-type” header of the HTTP/2 requests.

Implementation	**Client to Ingress Controller Protocol**	**Ingress Controller to Upstream Protocol**
Legacy System	HTTP/2 over TCP	HTTP/2 over TCP
Transitioning System	HTTP/3 over QUIC	HTTP/2 over TCP
Proposed System	HTTP/3 over QUIC	HTTP/3 over QUIC

Table. 6.1: Protocols Used - Outlining the protocols used by the three implementations

Characteristic	Legacy System	Transitioning System	Proposed System
Stream Splitting Effectiveness	Limited by TCP	Limited by TCP (Server Side)	Effective
Head-of Line Blocking	Present	Reduced (Client side)	Eliminated
Connection Overhead	High	Reduced	Low
Security	Uses TLS over TCP	TLS over TCP (Server Side)	Integrated in QUIC
0-RTT Connections	Not available	Not available	Supported
Connection Migration	Not available	Not available	Supported
Microservices Scalability	Base levels	Undermined by Server Side	Improved

Table. 6.2 Characteristic Comparison - Comparing the characteristics of the three implementations.

The Legacy System:
This system represents a traditional approach, leveraging HTTP/2 over TCP without the advancements of HTTP/3 or QUIC. While HTTP/2 introduced some level of multiplexing, it's limited by TCP's susceptibility to head-of-line blocking and the absence of features like connection migration and 0-RTT connections, which are vital for the rapid scaling required by microservices. Stream splitting is managed at the ingress controller by inspecting the "content-type" header, yet the overall performance is constrained by the latency and throughput limitations of TCP.

The Transitioning System:
This architecture takes a step towards modernisation by introducing experimental support for HTTP/3 and QUIC at the client-facing side of the ingress controller. It signifies progress with the ability to establish faster connections due to HTTP/3's benefits. However, this system does not extend QUIC to the server side, leading to a mismatch in protocol efficiency as the internal communication with microservices is still handled over HTTP/2 and TCP. This disparity results in losing some of QUIC's advantages after the ingress point, particularly affecting internal microservices communication, which is a cornerstone of microservice architecture efficiency.

The Proposed System:
The proposed system fully integrates QUIC with Kubernetes, managing both ingress and egress traffic over HTTP/3. It showcases end-to-end usage of QUIC's capabilities, thereby ensuring minimal latency, enhanced security with TLS 1.3, and efficient stream multiplexing throughout the cluster. This system aligns closely with microservice principles such as service isolation, fault tolerance, and rapid scalability. By eliminating the protocol mismatch present in the Transitioning System, the Proposed System can leverage QUIC's full potential within the cluster, offering a homogeneous and optimised environment for microservices communication.

In summary, while the Legacy System is bound by the constraints of TCP, the Transitioning System represents an intermediary stage that acknowledges the potential of QUIC and HTTP/3. The Proposed System takes full advantage of QUIC, leveraging its capabilities to provide a seamless and highly efficient networking framework. The Proposed System sets a new standard for communicating with microservices within Kubernetes clusters.

6.3 Scalability & Reliability Analysis

The proposed stream splitting mechanism facilitates scalability by distributing loads to microservices based on the content type of incoming data. This allows for the dynamic allocation of resources, where backend services can be scaled independently in response to varying demands, maintaining system performance.

By routing data streams directly to the appropriate processing service, the stream splitting mechanism reduces the travel distance within the network. This minimises the latency typically introduced by multiple service hops, ensuring efficient processing and quicker response times as system demand increases. This is another crucial aspect of scalability that is improved on by the stream splitting mechanism.

However, the stream splitting mechanism is dependent on Kubernetes to enable fault tolerance. Proper configuration of Kubernetes deployments and ingress controllers is needed to ensure that even if a service fails, the mechanism can reroute traffic to operational instances, maintaining service availability. With the proper Kubernetes and Ingress Controller configuration, the mechanism can support quick recovery and redistribution of traffic during outages

In summary, the stream splitting mechanism plays a pivotal indirect role in enhancing the scalability and reliability of systems. It does so by enabling more effective use of Kubernetes' dynamic features and by ensuring that traffic routing aligns with the current state of the system, promoting efficient resource utilisation and maintaining service integrity even under adverse conditions.

6.4 Summary Of Evaluation

The system was effectively demonstrated in a controlled environment, showcasing its capacity to route data streams based on content type. The operational tests, supported by detailed logs and visualisations, verified that the system performs as expected, efficiently managing and routing multiple data types to appropriate services.

When compared to other theoretical implementations, the proposed system is expected to reduce latency, enhance security, and improve data throughput, thanks to the full utilisation of QUIC's capabilities. The stream splitting mechanism enhances the system's scalability by enabling dynamic per service routing, resulting in minimised latency, which is crucial for handling varying traffic volumes effectively.

Chapter 7
Conclusions

To conclude this dissertation, this chapter summarises the limitations and reflects on the work done by identifying potential improvements of the stream splitting mechanism proposed.

7.1 Limitations

The proposed stream splitting mechanism also comes with its own limitations. These limitations concern its dependence on specific conditions like predefined content types, assumption of correct labelling by the client, and protocol reliance. Addressing these limitations is crucial for improving the adaptability of the system.

7.1.1 Limited To Predefined Content Types:

The system's efficacy is contingent on the ability to recognise and differentiate between predefined content types through HTTP headers. This approach restricts the system's flexibility, as it can only handle content types that are explicitly defined in the ingress controllers configuration. Consequently, any new or custom content types not initially configured would bypass the specialised routing logic, leading to less efficient handling.

7.1.2 Dependency On Client Content Labelling:

The mechanism's effectiveness relies heavily on correct content-type declarations by the client, as the system does not verify the accuracy of the declared content types; it assumes that the client has correctly labelled the content type. This assumption can lead to issues at the microservice layer if the content is mislabeled, resulting in inefficiencies and potential errors in data processing. This dependency makes the system vulnerable to issues stemming from client-side errors or malicious data manipulation.

7.1.3 Reliant On HTTP Protocol:

The system's reliance on HTTP protocol headers for stream identification and routing means it is not protocol-independent. This dependency restricts its utility to environments where HTTP is the application layer protocol used. This limitation is not overly concerning, as the majority of web services utilise HTTP, such a limitation can be significant in network environments where different applications and services need to communicate over a variety of protocols. Expanding the system's capabilities to interpret and manage other types of protocols could vastly increase its versatility and applicability across a broader range of technological ecosystems.

7.2 Future Work

To build on the current achievements and address the identified limitations of the proposed stream splitting mechanism, several areas of future work are recommended. These enhancements aim to improve the system's flexibility, performance, and general applicability in diverse network environments.

7.2.1 Further Investigation Of Alternative Ingress Controllers

A detailed investigation into other ingress controller implementations, notably HAProxy, could provide insights into their potential integration with the current system. This study would involve a comparative analysis against the existing ANGIE-based solution, focusing on performance, configurability, and ease of integration with QUIC and HTTP/3 protocols. Understanding the strengths and limitations of different ingress controllers could lead to more optimised and resilient implementations of the stream splitting mechanism.

7.2.2 Implementing Client Labelling Independence

Future work on enhancing the stream splitting mechanism should focus on achieving independence from client-labelled content, transitioning towards a system that autonomously identifies and classifies data streams. Using techniques outlined in (Karresand & Shahmehri, 2006), the approach could involve incorporating binary structure analysis within the ingress controller, or some "helper" service in the backend. Implementing this label-agnostic mechanism would not only improve routing accuracy and efficiency but also bolster the system's resilience against errors and manipulation associated with incorrect content labelling.

7.2.3 Enhanced Evaluation Through Implementation Of Theoretical Models

An important direction for future work is the enhanced evaluation of the proposed solution to determine the extent of the benefits QUIC's integration provides, through practical implementation and standardised testing of the theoretical models outlined in Section 6.2. Although initial analysis and existing research, such as Y, already provide a strong indication of QUIC's performance enhancements, there is significant value in a more thorough comparative analysis. Future efforts should focus on actually implementing the Legacy System and the Transitioning System alongside the current QUIC-integrated system. This will not only validate the theoretical benefits in a controlled environment but also highlight potential areas for further optimisation and refinement in real-world scenarios.

7.3 Closing Remarks

The work done over the course of this project has introduced and detailed the implementation of a header-based stream splitting mechanism, tailored to optimise microservice architectures, and also leverages the full potential of advanced networking protocols such as QUIC and HTTP/3. The overarching conclusion of this work underscores the importance of effective communication as a fundamental component of modern system infrastructures.

By proposing, developing, and demonstrating a header based stream splitting mechanism, this dissertation contributes to industry-wide efforts of addressing efficiency and scalability in network systems. The integration of modern technologies like HTTP/3 illustrates the evolutionary nature of network protocols and also highlights the continuous efforts within the industry to develop and refine solutions that address prevalent challenges in network communications.

As this work concludes, it paves the way for future research to expand upon its findings. Further studies are encouraged to build on this foundation, potentially implementing and testing the theoretical models discussed to provide deeper insights and empirical validation of the proposed solutions.

References

Babmberg, W. (2023, April 10). *Evolution of HTTP*. MDN Web Docs. Retrieved February 2, 2024, from https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP

da Silva, V. G., Kirikova, M., & Alksnis, G. (2018). Containers for virtualization: An overview. *Applied Computer Systems*, *23*(1), 21-27.

Dellaverson, J., Li, T., & Wang, Y. (2022). *A Quick Look at QUIC**.

Ingress Controllers. (2023, December 8). Kubernetes. Retrieved January 22, 2024, from https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

International Organization for Standardization. (1984). *Information processing systems - Open Systems Interconnection - Basic Reference Model* (Issue No. 7498).

Iyengar, J., & Thomson, M. (2021). *QUIC: A UDP-Based Multiplexed and Secure Transport* [RFC 9000]. Internet Engineering Task Force.

Jacobson, V. (1988). Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review*, *18*(4), 314-329.

Jain, S. (2023, July 21). *TCP/IP Model*. GeeksforGeeks. Retrieved March 17, 2024, from https://www.geeksforgeeks.org/tcp-ip-model/

Koblitz, N. (1987). Elliptic curve cryptosystems. In *Mathematics of Computation*.

Kubernetes Components. (2024, January 30). Kubernetes. Retrieved March 16, 2024, from https://kubernetes.io/docs/concepts/overview/components/

Kubernetes Networking: The Complete Guide. (n.d.). Tigera. Retrieved March 16, 2024, from https://www.tigera.io/learn/guides/kubernetes-networking/

Marx, R., Lamotte, W., & Quax, P. (2020). *Visualizing QUIC and HTTP/3 with qlog and qvis*.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. In *Linux j* (Vol. 239, Issue 2, p. 2).

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Incorporated.

Postel, J. (1980). *User Datagram Protocol* [RFC 768]. Internet Engineering Task Force.

Postel, J. (1981). Transmission Control Protocol: DARPA Internet program protocol specification [RFC 793]. In *Defense Advanced Research Projects Agency, Information Processing Techniques Office*.

Puliafito, C., & Conforti, L. (2022). Server-side QUIC connection migration to support microservice deployment at the edge. In *Pervasive and Mobile Computing* (Vol. 83).

Rivest, R., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM*.

Sobolewski, J. S. (2003). *Cyclic redundancy check*.

Solomon, B. (2019). *Async IO in Python: A Complete Walkthrough*. Real Python. Retrieved April 1, 2024, from https://realpython.com/async-io-python/

Thomson, M., & Turner, S. (Eds.). (2021, May). *Using TLS to Secure QUIC* [RFC-9001]. Internet Engineering Task Force (IETF).

Wang, S., Sherry, J., & Han, S. (2013). *A Dual-Channel Approach to Protocol Design in the Presence of Middleboxes* [Technical Report No. UCB/EECS-2013-205]. EECS Department, University of California, Berkeley.