HTTP3 over QUIC Applied for Ingress Network Communication Within a Kubernetes Environment

Pascal Raos, ICS

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Data Science)

Supervisor: Stefan Weber

April 2024

HTTP3 over QUIC Applied for Ingress Network Communication Within a Kubernetes Environment

Pascal Raos, Master of Science in Computer Science

University of Dublin, Trinity College, 2024

Supervisor: Stefan Weber

Kubernetes provides an orchestration framework for micro-services based on containerised solutions. Individual services are instantiated a number of times within a Kubernetes deployment and the orchestration framework manages the network traffic between the individual services to each other and network elements outside a deployment. The communication between the components of a deployment, the control plane managing a deployment and the networks elements outside a deployment is generally based on a very traditional setup of HTTP/1.1 over TCP over IPv4

The evolution of HTTP has resulted in a development towards encrypted protocols that implement stream-based communication over connection-less transport protocols e.g. the current version of HTTP/3 is layered on top of QUIC which in turn is based on UDP and implements TLS 1.3 and individually synchronised streams. This development is in stark contrast to the traditional approach taken in Kubernetes and introduces significant challenges when attempting to connect network elements outside a deployment to services inside a deployment using HTTP/3 and to traffic from individual streams to specific services within a deployment.

The research provides an attempt to apply the QUIC based HTTP3 protocol within a kubernetes environment, for the purpose of ingress based communication. The focus lays on the difficulties which lay therein, and the methods taken to circumvent and ultimately get a working solution. We intend to avoid simplistic implementations which mirror a simplified architecture, as that would defeat the purpose of applying it to the Kubernetes environment. As such the focus is on ingress or ingress controller based implementations. Finally we will compare the ease of setup against global use of the protocol, highlighting the discrepancy within the current open source environment.

Acknowledgments

Thank you to my dissertation supervisor, Professor Stefan Weber for all of the support throughout the project.

Pascal Raos

University of Dublin, Trinity College
April 2024

Contents

Abstract **i**

Acknowledgments **ii**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Structure & Contents 2

Chapter 2 Transport Protocols - Technical Background **4**

2.1 TLS 4

2.2 UDP 5

2.3 TCP 5

2.4 QUIC Protocol 6

2.4.1 Streams 6

2.4.2 TLS integration 6

2.4.3 Handshake 7

2.5 Comparison 7

Chapter 3 Application Protocols - Technical Background **8**

3.1 HTTP 8

3.1.1 HTTP/1 9

3.1.2 HTTP/2 9

3.1.3 HTTP/3 10

Chapter 4 Packet Capturing **11**

4.1 Wireshark 11

4.2 Tcpdump 12

4.3 Qvis & Qlog 12

Chapter 5 Kubernetes - Technical Background 15
5.1 Basic Cluster Components . . . 16
5.1.1 Nodes . . . 16
5.1.2 Containers and Images . . . 16
5.1.3 Pods . . . 17
5.1.4 Deployments . . . 18
5.1.5 Services . . . 18
5.1.6 Secrets and ConfigMaps . . . 19
5.1.7 Namespaces . . . 19
5.2 Advanced Kubernetes Components . . . 19
5.3 Control Plane . . . 20
5.3.1 Kube-apiserver . . . 20
5.3.2 Etcd . . . 20
5.3.3 Kube-controller-manager . . . 21
5.3.4 Cloud-controller-manager . . . 21
5.4 Node Componenets . . . 21
5.4.1 Kubelet . . . 22
5.4.2 Kube-Proxy . . . 22
5.4.3 Container Runtime . . . 22
5.5 Cluster Setup . . . 22
5.6 Cluster Management Tools . . . 23
5.6.1 Kubectx and Kubens . . . 23
5.6.2 K9s . . . 24
5.6.3 Lens . . . 25
5.6.4 Helm . . . 25
5.7 Local Clusters . . . 26
5.7.1 K3d . . . 27
5.7.2 Minikube . . . 27
5.7.3 Kind . . . 27
5.7.4 Comparison . . . 27
5.8 Remote Clusters . . . 28
5.8.1 EKS . . . 28
5.8.2 AKS . . . 30
5.8.3 Cost Model . . . 31
5.8.4 GKE . . . 31
5.8.5 Comparison . . . 33

Chapter 6 State of the Art **35**
6.1 QUIC and HTTP3 Usage Statistics 35
6.2 QUIC Implementations 37
6.2.1 AIOQUIC 37
6.2.2 Quic-go 38
6.2.3 Comparisons 38
6.2.4 Implementation Considerations 39
6.3 Middlebox Ossification 39
6.4 QUIC packet flow 39
6.5 Ingress Controllers 44
6.5.1 Nginx 44
6.5.2 TraefikProxy 45
6.5.3 Comparison 46
6.6 Summary 46

Chapter 7 Problem Formulation **47**
7.1 Problem Formulation 47

Chapter 8 Design **49**
8.1 System Architecture 49
8.2 System Transparency 50
8.2.1 Packet Capturing 50
8.2.2 Packet Visualisation 50
8.3 QUIC Client Connections 50
8.3.1 Web Client 51
8.3.2 CLI Client 51
8.4 Kubernetes Setup 51
8.5 Ingress Server 52
8.6 Backend Server 52
8.7 Summary 52

Chapter 9 Implementation **53**
9.1 Kubernetes Cluster Setup 53
9.2 Host Protcol Testing 54
9.3 Image Building 54
9.4 Kubernetes Yaml Construction 54
9.5 Packet Capturing 55
9.6 TraefikProxy Testing 56

9.7 HTTP3 with TraefikProxy 58
9.8 Cloud Environment 60
9.8.1 Account Creation and CLI setup 60
9.8.2 Cloud TraefikProxy Testing 61
9.8.3 Cloud HTTP3 Testing 61
9.9 Summary 65

Chapter 10 Evaluation **66**
10.1 Results 66
10.1.1 Local AIOQUIC 66
10.1.2 Intra-Cluster Testing 66
10.1.3 Cloud HTTP3 68
10.2 Summary 70

Chapter 11 Conclusions & Future Work **71**
11.1 Future Work 71

Bibliography **73**

Appendix A Appendix **76**
A.1 Browser Testing 76
A.2 Local Testing 76
A.3 Image Building 77
A.4 AIOQUIC-Deployment 77
A.5 Traefik Testing 80

List of Tables

2.1 Transport Protocol Comparison 7

5.1 Kubernetes Cloud Providers 33
5.2 Kubernetes Cloud Providers 33

List of Figures

4.1 Wireshark Packet Capture 11
4.2 TCPdump 12
4.3 Qvis 13

5.1 Basic Cluster Components 16
5.2 Nodes 20
5.3 K9s 24
5.4 Lens 25
5.5 Helm 26
5.6 AKS Cluster Choice 31

6.1 HTTP Usage Share 36
6.2 HTTP3 Browser Share 36
6.3 Local QUIC Pcap 40

9.1 Minikube Start 53
9.2 Local AIOQUIC 54
9.3 Simple TraefikProxy 57
9.4 Pending IP 61
9.5 Multi-port NLB Error 61
9.6 Traefik Values Reference 63
9.7 Cloud HTTP/3 Reply 65

10.1 Intra-Cluster PCAP 67
10.2 Intra-Cluster PCAP 68
10.3 Cloud Qvis 1 69
10.4 Cloud Qvis 2 69

Chapter 1

Introduction

1.1 Motivation

Since the conception of the internet, starting as attempts for cross-university communication between computer systems in 1969 with the ARPANET [9], communication protocols between nodes or servers have played a crucial role.

The original Network Control Protocol kernel-level protocol [27] used was too simplistic, not accounting for homogenous networks or packet dropping. The Transmission Control Protocol (TCP) [7], first introduced in 1974, and later applied within the ARPANET environment in 1983 [9], improved inter-network communication, alongside improved reliability, flow control and packet recovery.

TCP has been deeply rooted in the history of the internet, and despite being over forty years old, is still the most used transport layer protocol on the internet [32]. in its time it has become ingrained in different network layer protocols, such as the popular SSH or HTTP, and as such, has solidified its place in network communication. Other derived or standalone protocols have attempted to iterate on, or extend the use cases of TCP. However they have been unsuccessful in attaining wide-spread adoption.

The QUIC protocol, initially designed by Google in 2013 and standardised in 2021, is built upon the User Datagram Protocol (UDP), with the high level features of TCP built on top of it seems, to indicate a shift in the networking landscape. QUIC improves on some blocking, handshake and general performance over TCP, but its main advantage over other TCP competitors is that it is proposed and used by Google. The sheer scale of Google rivals the implementation biases brought about TCP's early adoption, and provides this new transport layer protocol a stage to shake the global infrastructure.

Currently, within most modern open source softwares, QUIC or HTTP3 is seen as an experimental software, and does not promise proper performance or implementation in many software packages.

The motivation for this dissertation is identifying the level of adoption outside of proprietary software, viewing open source implementations of the QUIC protocol, specifically in conjunction with HTTP3. We want to explore to what level they have been implemented, and how its improvements have been integrated into high performance networking applications. The environment of choice we wish to apply this to is Google's own kubernetes container orchestration environment, which its open source distribution does not natively support QUIC. Highlighting and recording the difficulties in implementing this standard.

1.2 Structure & Contents

The order and value of each of the chapters in the dissertation is as follows.

Chapter 2 - Transport Protocols

Contains technical background regarding the transport layer and transport protocols relevant to the dissertation. Specifically minimum details needed to understand the QUIC algorithm and benefits over its peers.

Chapter 3 - Application Protocols

Provides the reader with basic application layer protocol information. This helps to understand the delegating of certain transport concepts and the relationship between the application and transport layer.

Chapter 4 - Packet Capturing

This chapter introduces standard packet capturing and analysis tools, and how they differ semantically and when deployed in environments.

Chapter 5 - Kubernetes

This section describes in great detail the core components of the Kubernetes environment. It will help the reader understand the complexity of the environment, in the scope of standard protocol application to the environment. It covers basic and advanced cluster components, and local versus remote cluster deployment.

Chapter 6 - State of the Art

This section covers more applied technologies, or implementations of technical background standards. We delve deeper into inference on design decisions and why previous performance research for the QUIC protocol showed discrepancies. This sets up the core them of the project, the broad scope of difficulty in all regards to QUIC adoption.

Chapter 7 - Problem Formation

Focuses on inference from the previous chapter, and what we conclude is a issue to address or topic to explore within the dissertation.

Chapter 8 - Design

Based on previous research, the methods needed to set up a minimum viable HTTP3 testing architecture, and functional requirements our used technologies must follow.

Chapter 9 - Implementation

This chapter focuses on the application of software, following the design principle established prior. It also highlights the issues present in deployment, and discrepancies in performance of certain technologies, ultimately leading to using a cloud environment.

Chapter 10 - Evaluation

The section focuses on evaluating results obtained in the implementation section, to potentially make more inference on results.

Chapter 11 - Conclusions & Future Work

Chapter focuses on final comments regarding the project, and areas that could be viewed or stem from work done.

Chapter 2

Transport Protocols - Technical Background

The Idea of the network stack, defines the encapsulation order of information as it begins from the application creating the data, down to the hardware which transmits the encoded data between two hosts.

Within this project, we focus greatly on the Transport Layer [25]. This is the layer which controls higher level flow control, header and encapsulation of data between two hosts. The User Datagram Protocol (UDP), QUIC and the transmission control protocol (TCP) are some of the most prominent transport layer protocols in the modern age. Before progressing, we will identify some of the most prominent differences between these major transport layer protocols, and how QUIC compares against the most widely deployed standard, TCP. We will also look at how network security is handled at the transport layer.

2.1 TLS

TLS (Transport Layer Security) [30], is an encryption algorithm used to secure data on the transport layer. It does so by allowing hosts first to agree on the version of TLS to be used, and the encryption protocols supported by both hosts, all done through an initial connection handshake. A host or server can register themselves with a third party to have their ownership of a web domain recorded in a root certificate authority (CA). Communication between a client and a server proceeds with the server sending a signed certificate,its public key. The client would confer with the root CA, to see if the server is the rightful owner of what the web domain the client is communicating with. Once

established, the client and server cryptographically generate session keys for encryption and decryption for their singular session.

2.2 UDP

UDP works as a connectionless protocol, which trades off its reliability in favour of speed, and reduced network bandwidth. This minimizes its usefulness outside of noiseless network environments
UDP was initially created by David Patrick Reedm, with its standard defined by the Internet Engineering Task Force (IETF) in their standard documentation Request for Commons (RFC) number 768 [26].

While UDP allows failures and duplicate packet delivery in noisy network conditions, it does not allow the processing of corrupted packets. It handles packet corruption detection using checksums, which are binary outputs from logical operations on the original payload, and sent alongside. if the same operation does not produce the same logical output, it can be said one or more of the bits in the packet were corrupted during transmission.

2.3 TCP

TCP is a connection based transport layer protocol that transmits data packets in an ordered delivery sequence [28]. It can account for lost packets with retransmission and has congestion control to prevent oversaturate the connection bandwidth. TCP can send multiple different data types at once but does not differentiate between them.

TCP includes more advanced transport layer functionality than UDP, to allow it to maintain a strong level of communication, especially in a best effort network environment like the internet. It does this by incorporating sequence numbers for packets, each of which is acknowledged by the peer host after receiving them. This ensures proper tracking in cases where packets can be dropped, corrupted or arrive out of order. It also implements strongly researched congestion controls algorithms [2], alongside a connection oriented bystream to ensure proper data handling.

TCP connections begin with a three way handshake, exchanging initial sequence information between hosts, and ensuring both hosts are in an acceptable state before transmitting data. All of these functions are the basic building blocks of stable internet data transfer.

2.4 QUIC Protocol

The QUIC protocol is a connection based transport layer protocol built on top of UDP. It was created by Google in 2013 and standardised by the IETF in RFC9000. It adds its own implementation of TCP-like packet encapsulation, congestion control, and connection handling to stabilize UDP communication. Its relationship with UDP allows the encapsulation of QUIC packets within UDP datagrams. The use of UDP allows interoperability with older established services which only support older transport layer protocols.

In the following subsections we will highlight the unique features QUIC provides, and improvements against its direct predecessor TCP.

2.4.1 Streams

The main transport level functionality offered by QUIC is its ability to handle multiple byte streams within a single connection, providing a separate logical QUIC stream for each source. flow control, priority and packet handling, can be handled on a per-stream basis.This is different to TCP, which multiplexes multiple byte streams into a single tcp connection stream.

TCP would suffer from a very prominent head-of-line blocking (HOL) issue. If a packet of a specific data type was dropped and must be retransmitted, all other packets across the connection stream must wait, even though not all of them have a dependence on the lost packet. QUIC guarantees in-order delivery of packets in each individual stream, allowing a client to allocate a separate stream for related packets, claiming to circumvent the HOL blocking issue.

However, some points are brought into question, HOL blocking is removed only if multiple QUIC streams are being used concurrently. Taking into account the bursty nature of packets, if a packet is lost from a single stream due to network interference, there is a likely chance it would be lost from other streams too, causing blocking on multiple streams. An interesting discussion follows by Robert Marx on a github post online [21]

2.4.2 TLS integration

TLS is used natively within QUIC, meaning all QUIC communication must be encrypted and authenticated between hosts using the protocol. This differes greatly from TCP, which uses TLS as an additional feature on top of its transport layer functionality. This provides the benefit of encrypted communication at all applications of QUIC, but adds a

level of complexity when applying the protocol in simple use cases, or testing.

2.4.3 Handshake

QUIC follows the standard methods of initiating its connections with a handshake. The native incorporation of TLS within the QUIC protocol itself allows QUIC to send TLS handshake information alongside its own protocol handshake. This allows QUIC to come to an agreement with a server within one round-trip-time (RTT), compared to TCP's 1-RTT without TLS and 2-RTT with.

QUIC also offers a 0-RTT handshake, allowing a client and server to communicate while the initial handshake is occurring. It is done based on remembered values from a prior connection with the server. It incorporates some limitations however, such as the inability to lower flow control parameters beyond the remembered values, or forfeiting a level of security.

2.5 Comparison

Protocol	Connectioned	Reliability	HOL Blocking	RTT	Encryption
UDP	Connectionless	Unreliable	N/A	N/A	Possible w/ sockets
TCP	Connectioned	Reliable	Yes	1-3	Optional
QUIC	Connectioned	Reliable	Reduced	0-1	Mandatory

Table 2.1: Comparison of core transport layer features between protocols, identifying whether the protocol establishes a connection with the server, how reliable is packet delivery, HOL blocking, total RTT for packets, and how it implements security measures

Chapter 3

Application Protocols - Technical Background

Application Layer Protocols [24] are the networking protocols responsible for the end to end communication context between two machines. Transport Layer protocols determine the advanced rules for how these messages travel between hosts. In the context of the dissertation, the HTTP application layer protocol consists of messages which control functional communication between hosts, and the transport protocols QUIC or TCP, control packet level operations when sending and receiving these packets, ensuring they reach their destinations.

An Application protocols design tends to adhere to functional limitations for the underlying transport protocol it will support. For that reason, iteration and improvements on an Application layer protocol could be heavily stunted if it could cause instability with its transport layer.

3.1 HTTP

HTTP is an application layer protocol designed to communicate between a client and server architecture. It follows a Create, Read, Update and Delete (CRUD) philosophy. HTTP has four respective operations for a client to request of the server, POST, GET, PUT, DELETE. HTTP is commonly used for serving web HTML, javascript and stylesheets for web page loading. The HTTP protocol has been iterated on over the years, greatly improving on the former version. HTTP3 is next in line.

3.1.1 HTTP/1

HTTP/1 [22]was a simplistic method of client to server communication, built to be used by the globally accepted TCP transport protocol. It followed the basic CRUD philosophy, which was very appropriate for the early days of the internet. It came with many limitations, the most notable being, it required a separate TCP connection for each HTTP Request. This meant TCP handshakes would occur for every request, causing huge delays.

HTTP/1.1 resolved this by using persistent connections, allowing multiple HTTP requests to be run on a single one, to which either party could signal the close of the connection once finished. However it highlighted another issue, that a single connection only allowed sequential request handling, meaning any delays would cause the next request in line to wait. This is commonly know as Head-Of-Line (HOL) Blocking.

Furthermore, no resource prioritization was implemented, and requests would be handled first-come first-serve, regardless of whether a prior request contained data which needed to be received first. Finally, HTTP/1 stores packet header information as white-space delimited plain-text, vastly underutilizing network bandwidth

3.1.2 HTTP/2

HTTP/2 [6] was introduced in 2015, sixteen years following its predecessor. In the meantime, the average size of web pages would increase, with the development of more complex web designing frameworks, and improved internet speeds facilitating that growth.

While HTTP/2 still depends on TCP for transport layer communication, it improves on HTTP/1 in several ways. It implements a feature known as multiplexing, allowing it to differentiate between separate HTTP requests along a single connection. It is achieved through a feature known as binary framing, the act of splitting data into individual binary encoded frames, associating them with requests and interleaving them across the network connection. This means multiple requests can be created and sent to the server, becoming queued until the server returns a HTTP reply. This eliminated HTTP-level HOL Blocking, however the underlying implementation of TCP does not differentiate between the separate packets it streams while multiplexing. This means if a packet is lost in TCP communication, HOL Blocking still occurs, as it can not determine which other byte streams are associated with the specific packet.

HTTP/2 does add prioritization to requests handling, ensuring the appropriate requests are handled first. It also implements a server push feature, allowing the server to return resources associated with a previous client request, saving round-trip times. Binary framing allowed improvement of network bandwidth usage by eliminating wasteful

cleartext in packet headers.

HTTP/2 introduced a header compression method known as HPACK to further reduce packet header sizes. Header compression is maintained by both the client and server and is represented by indices in dynamic table lists mapping to the corresponding HTTP header. When a client sends a request to the server, it creates an index for this new header and sends it along to the server with the original request. The server can then associate the index with the header in its local dynamic table, allowing further communication between client and server to use the index in place of a lengthy header. Encoding and decoding can be done both ways

3.1.3 HTTP/3

Whilst HTTP/2 heavily improved over HTTP/1, It still had its shortcomings, especially through some TCP specific limitations.

HTTP/3 [15] is the first HTTP built the new QUIC transport layer protocol. The majority of its improvements over HTTP/2 directly mirror QUIC's improvments over TCP discussed previously. The binary framing mechanism incorporated in HTTP/2 is delegated and implemented within the QUIC protocol. HTTP/3 delegates stream control to QUIC natively, and simply maps each request and response pairing to an independant QUIC stream, allowing concurrent queueing.

QUIC implements its own header compression algorithm, QPACK [14], as despite binary framing translating well to QUIC streams, HTTP/2's HPACK header compression relied on the guarantee of TCP delivery header frames in order to the server, to allow approriate build-up of the dynamic table mappings used for compression. QUIC guarantees in order delivery of individual streams, but not packets on the connection as a whole. Therefore QUIC implements its own compression algorithm at the transport layer to handle these changes.

Chapter 4

Packet Capturing

4.1 Wireshark

Wireshark [11] is a packet capturing and analysis tool originally created by Gerald Combs in 1997. Its iteration over the year has been by necessity. When a protocol was found to be unsupported, packet dissection code was written for wireshark, and contributed to the project.

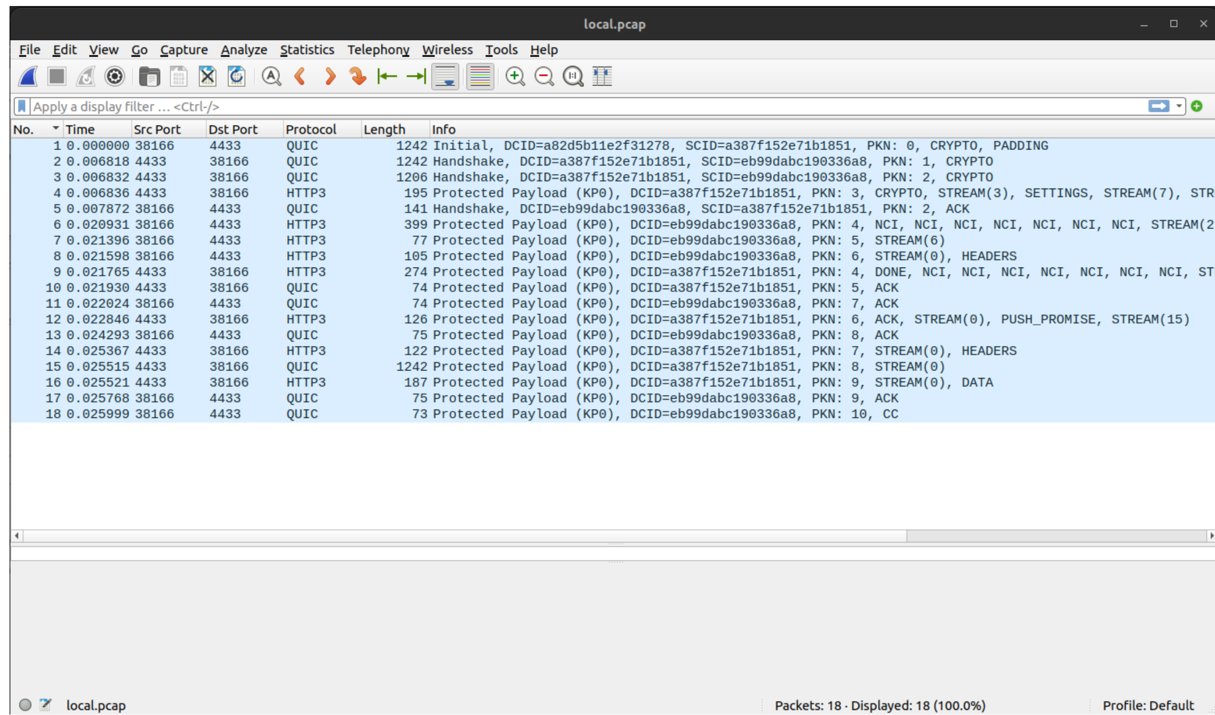


Figure 4.1: Wireshark packet capture, showing the results of a QUIC client-server session. Each individual line number corresponds to a UDP datagram sent or received

Over the course of years, support for many protocols was incorporated into the original software, upgrades to the graphical interface. Wireshark has a command-line alternative named tshark, and works similarly to other command-line packet capturing tools like Tcpdump

4.2 Tcpdump

Tcpdump [33] is an open-source Command-Line-Interface (CLI) based tool. it allows you to listen to network traffic on, or travelling through, ,the host machine on which the tool runs. It was developed and released in 1987 by Van Jacobson, Craig Leres and Steven McCanne, who were workers at the Lawrence Berkeley National Laboratory in California, USA

```
pascal@pascal-Precision-5540:~/Desktop/Dissertation_Repos/Dissertation-Files$ sudo tcpdump udp and dst 34.89.71.52
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on wlp59s0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
23:32:28.628386 IP pascal-Precision-5540.40847 > 52.71.89.34.bc.googleusercontent.com.https: UDP, length 1200
23:32:28.656349 IP pascal-Precision-5540.40847 > 52.71.89.34.bc.googleusercontent.com.https: UDP, length 91
23:32:28.657598 IP pascal-Precision-5540.40847 > 52.71.89.34.bc.googleusercontent.com.https: UDP, length 227
23:32:28.657810 IP pascal-Precision-5540.40847 > 52.71.89.34.bc.googleusercontent.com.https: UDP, length 57
23:32:28.676923 IP pascal-Precision-5540.40847 > 52.71.89.34.bc.googleusercontent.com.https: UDP, length 57
23:32:28.679253 IP pascal-Precision-5540.40847 > 52.71.89.34.bc.googleusercontent.com.https: UDP, length 27
```

Figure 4.2: The figure shows a tcpdump command, acting on the UDP protocol towards a specific IP address, in this case, a public google endpoint. You can see the communication between the host computer, the remote server, the timestamp,and the size of the UDP datagram

The tool is configurable to filter certain criteria to listen on. For example, you can filter by specific port open on the host, specific network protocols, source and destination addresses, and so on. The tool allows you to inspect the packets in real time in the terminal, or allows you to write the output files to pcap format. You can easily use another graphical tool, like Wireshark, to read and do more advanced dissection of the pcap file produced.

4.3 Qvis & Qlog

QVIS is a QUIC protocol visualization tool developed by github user Robin Marx [31]. QVIS is designed to work natively with the qlog packet format. qlog is a trace-based output, which stores the packet information of all QUIC connections between a client and server.

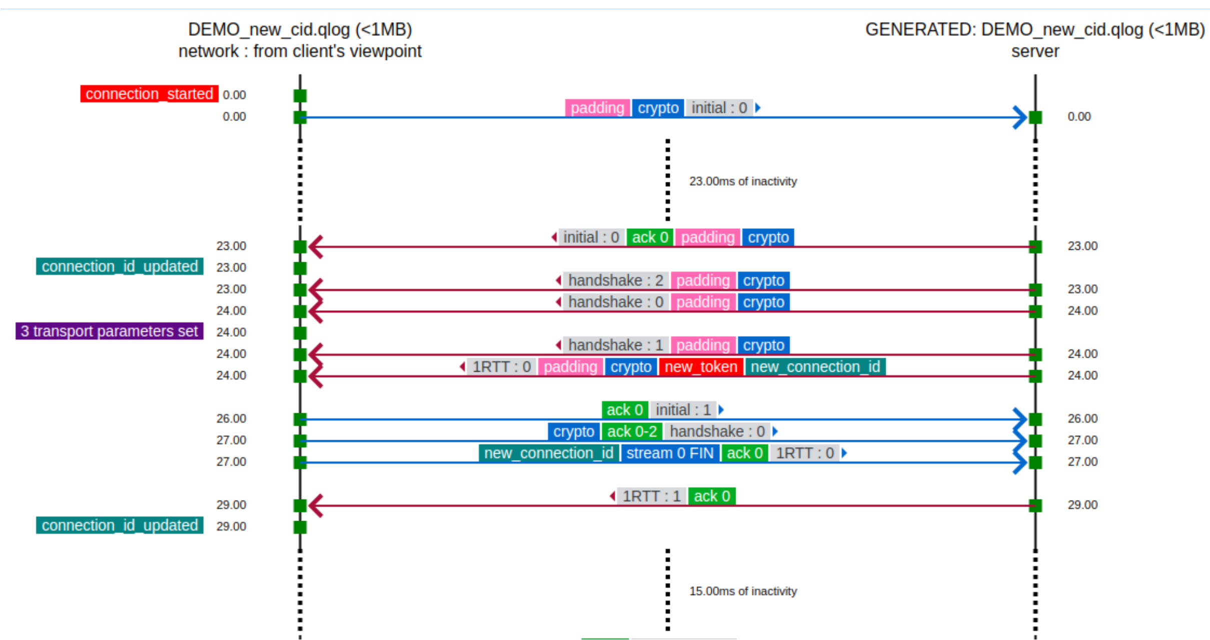


Figure 4.3: The figure shows a QVIS graphical output from a qlog trace file associated with an example QUIC client to server communication. We can see handshake information from both client and server, included within the traces would be server certificate information, key exchange values, connection establishment and QUIC parameter configuration

It is a very useful tool for beginners to identify individual connection streams, and client/server state changes when events occur. However, Wireshark's packet dissection provides more detail of the contents of each QUIC and UDP datagram, especially when identifying TLS handshake processes.

While many open source QUIC implementations support qlog output for debugging purposes, based on my research, QVIS is the only readily available visualization tool for the native format.

The developer of the QVIS tool created an accessory software, named pcap2qlog, which allows the conversion of pcap files to json, and then ultimately qlog formatted outputs. This would allow users to use conventional tools like tcpdump and wireshark to capture QUIC communication on a host network, and prepare it for visualization without using a dedicated QUIC client or server with qlog output support.

Personal testing however introduced a major issue. I found I was not able to successfully convert pcaps to qlog outputs using the pcap2qlog tool. An issue is open on the tool's github regarding the issue, where it is suggested that new versions of Wireshark have changed the way it parses pcap files when converting to json formats. Therefore causing errors when the tool finally converts from json to qlog. The sole developer Rmarx shows reluctance to update the tool, or provide direct pcap support for the QVIS tool

itself. The tool itself is only updated once every few years.

This issue with pcap converison causes an issue for testing environments. If an implementation of QUIC does not natively support qlog trace outputs, we will never be able to receive qlog traces from the server's perspective. Server flows are automatically generated if you have a client's perspective, but some internal state changes within the server will not be visible.

Chapter 5

Kubernetes - Technical Background

ubernetes is a open source container orchestration software, first created in 2014 by Google [5]. Containers are a high level software abstraction of a host machine, or server. These containers are virtual instances of some minimal operating system, combined with various tools and software built on top of it. This allows the creation of isolated lightweight environments that allow a specific service or range of services to run.

An example of this can be some SQL database software, separated outside of your host machine, cannot access the internet, and it is given a portion of your machine's disk space to store its data. The resources available to the container, and any software installed does not affect your host machine in any way.

A Kubernetes cluster is the combined effort of multiple servers or nodes, working together to allocate resources to these containers. There are two distinct types of nodes, Master nodes which delegate resources to the control plane, and worker nodes which delegate resources to containers.

The control plane controls the state of the cluster, and delegates the responsibility of newly created containers to worker nodes. Worker nodes allocate CPU resources, memory and disk space to allow containers to function in their isolated environments. Which node a newly created container is delegated to depends on the total state of the cluster, incorporating the requirements of the new container and which worker can best provide these resources.

5.1 Basic Cluster Components

Here we will discuss the basic building blocks for Kubernetes, and how a single service that may exist on your host machine, a load balancer, a database, or some cache service, can be instantiated within a cluster.

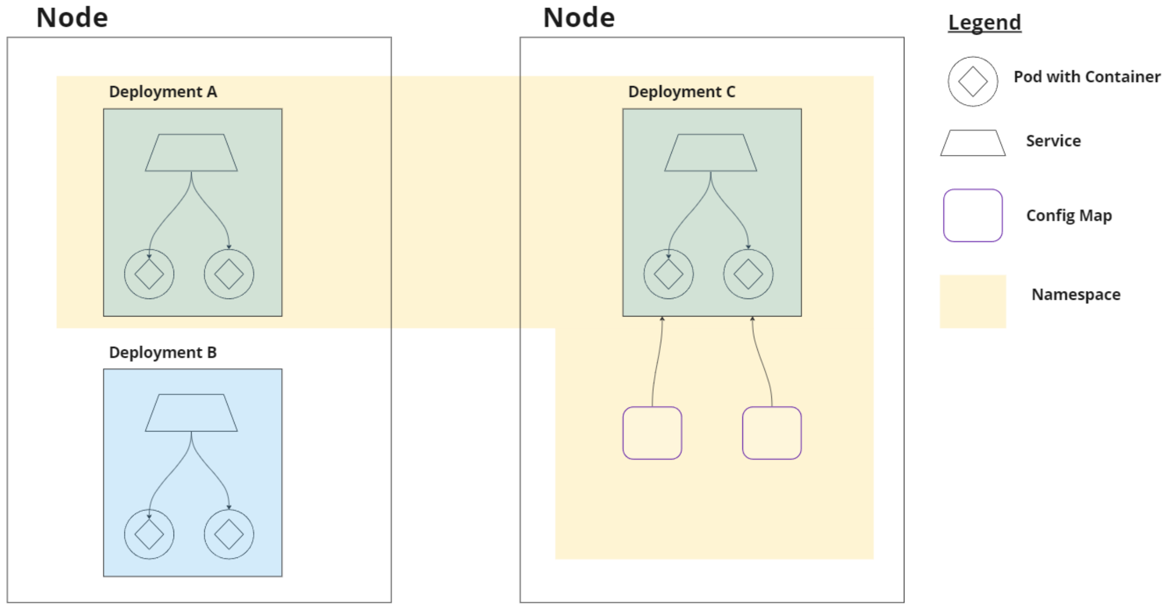


Figure 5.1: Figure showcases the basic Kubernetes components referenced below. From Deployments consisting of pods, their associated services, configMaps or secrets, and namespaces which logically encapsulate association

5.1.1 Nodes

Nodes are the components which make up the underlying cluster infrastructure. Whether they are master or control nodes, they are responsible for providing the CPU, Memory and Disk space needed for both containers and the core underlying services which control the cluster to function properly. Components are scheduled by the control plane to be places on Nodes.

5.1.2 Containers and Images

Kubernetes containers are built up using an underlying container runtime, such as docker [13], containerd [4] or others depending on the underlying operating system. These runtimes define the set of rules to separate a container process and isolate it from your host machine, alongside networking, storage and resource management.

```
FROM postgres
CP start.sh /opt/start.sh
CMD [./opt/start.sh]
```

Listing 5.1: The code shows a simple dockerfile image. The image is built from the basic postgres image, then copies a file onto the container, before finally running that file

On top of this are images. Images are the definition of what the core software of the container will be, and what start-up commands will be run when it begins. This can allow you to configure a working service, such as a database solution, without manually running installation commands within the container environment. Images can be stored on your local machine, or on some globally accessible image registry. A popular example is docker hub

5.1.3 Pods

Within Kubernetes, a pod could be considered to be the smallest unit. A single pod defines containers, with images, and what ports are accessible for those containers to interact with other pods in the cluster

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Listing 5.2: Shows a simple Kubernetes yaml file, defining "Kind" for the component described, "spec" contains container configuration features, such as "image" for the base container software, and "ports" to expose a container service outwards to the cluster

A Kubernetes yaml file can be used to define Kubernetes components, such as the pod. This information is read into the cluster and updates the cluster state, beginning the process of setting up the new component on a node.

When a pod with a specific image is defined, Kubernetes can find the image needed through online registries or locally built images.

5.1.4 Deployments

Deployments are a way of defining a collection of Pods in a single yaml file. This can allow you to create or delete pods by changing a single yaml, and allows the creation of multiple duplicate pods, each with their own unique ip address accessible within the cluster.

5.1.5 Services

When pods within a deployment go down and are restarted, they are given a new ip address. This can cause issues if two pods are interacting using their given ip addresses. A service is a way of attaching a static hostname onto a set of mutiple duplicated pods defined by a deployment. This allows pods to communicate with a set of pods using a service name instead of an IP. The service inherently load balances incoming connections across all the replicated pods, which is the act of intelligently routing network traffic to different sources, either through random chance, or based on the network congestion of each server or pod. There are two advanced archetypes of the default service, a nodeport and loadbalancer service. These are two methods of pushing Kubernetes services to be reachable from outside of the cluster.

A Nodeport service maps a port from a Kubernetes service to all of the worker nodes in the cluster. This is a way of allowing nodes which have been opened to the internet to allow services to reach outside the cluster. The main issue is that the port is open on all nodes in the cluster for routing.

Loadbalancer services are similar, but rely on third party methods of internet communication. This is commonly done with cloud providers, who have their own network configurations in datacentres, which allow them to map internal IP addresses to pods and services. Once this is done, cloud providers can map their own external load balancers, those existing outside of Kubernetes, to route traffic to the service within the cluster, through the nodes. This eliminates the issues with Nodeport services which take up ports on the host node.

5.1.6 Secrets and ConfigMaps

Secrets and ConfigMaps are very similar in design. They are stateful components which define some sort of variable to be provided to a pod or multiple pods. ConfigMaps tend to be used to store environment variable which are used at runtime by the pods, whilst secrets store encrypted authentication or passwords, in the case a pod must login with some credentials or API keys to access the internet or other services. These values are retained when a pod restarts

5.1.7 Namespaces

Namespaces are a virtual abstraction of resources within Kubernetes. They allow you to do separate groups of pods, services, or other components from each other. This can allow you to set specific user permissions on namespaced components, for example, to ensure within a company developers can only modify certain resources within the cluster. Hostname resolution for services are namespaced. This means within two separate namespaces, you can have the same set of pods and the same set of services exist simultaneously. If a pod tries to communicate with that service, it will by default prioritize communication with the service in its own namespace.

```
<service-name>.<service-namespace>.svc.cluster.local
```

You can instead define a long-hand hostname in this form, which specifies the name of the service, along with the namespace it resides in. This is useful to communicate with namespaced components explicitly regardless of the initial calling pod's location

5.2 Advanced Kubernetes Components

The worker and master nodes, which contribute to the state managing control plane, and resource allocating working area respectively, have underlying components used to fulfill their core functions

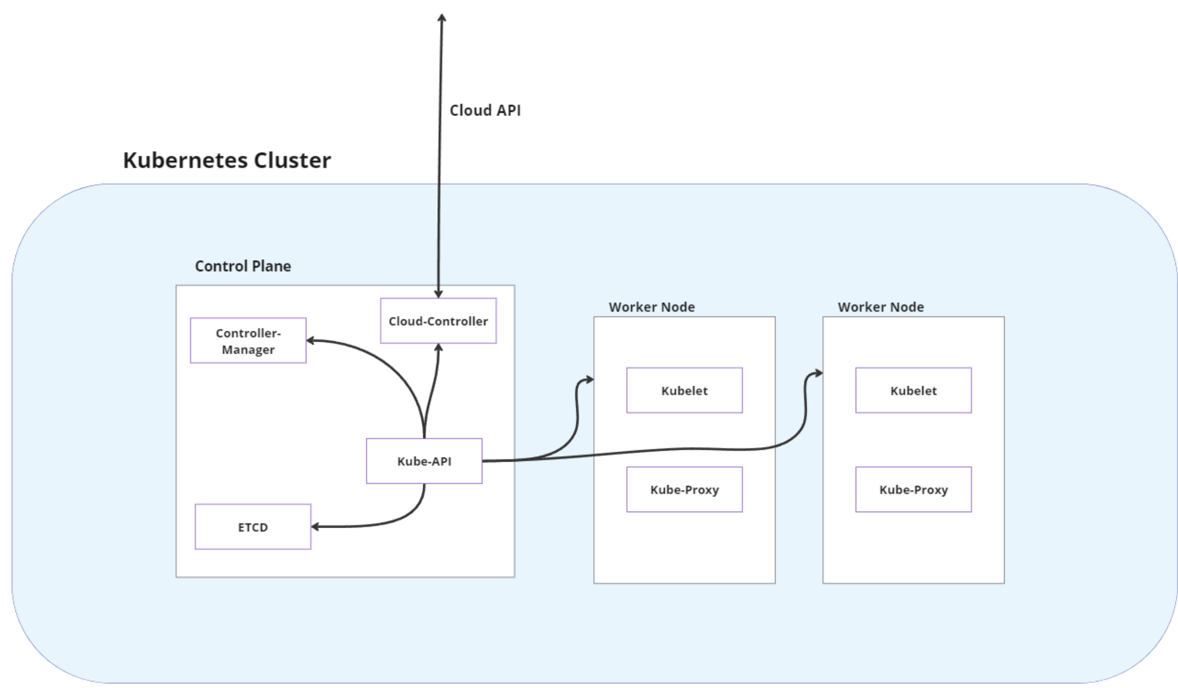


Figure 5.2: Figure shows the key components of the control plane and the basic components of schedulable worker nodes.

5.3 Control Plane

The Kubernetes control plane consists of a few key components that allow an initial cluster configuration to be created and functional. These are an etcd key-value store, the Kubernetes api-server, a controller manager, and a scheduler. The control plane itself can consist of one or more master nodes

5.3.1 Kube-apiserver

The kube-api server is the entrypoint into the cluster for a system administrator. It allows the user to do create, read, update, and delete (CRUD) operations within the cluster, and processes them to change the cluster state, or provide information to the user.

5.3.2 Etcd

etcd [10] is a software used for key-value storage applied to a distributed systems. It brings with it functionality for fault tolerance across distributed nodes, scalability to new nodes, consistency models across the nodes, and additional monitoring methods to ensure up to date values.

Within Kubernetes the etcd software is used to store cluster state in its key-value store, from which the control plane can make decisions on resource allocation, discover new services within the cluster. The Kubernetes API is a way of modifying or reading the etcd state data, if you have been given sufficient permission to do so.

Access to the etcd is equivalent to root access to the cluster, therefore certificates are setup across the different control plane components to ensure only authenticated users within the cluster can change cluster state.

5.3.3 Kube-controller-manager

A controller within Kubernetes is a type of service that defines some infinitely running process which controls the behaviour of some cluster function. Some examples are the Node controller, which helps responds to nodes failing within the cluster. The Job Controller, which runs Kubernetes Job objects, which support the creation and deletion of containers within the cluster.

This component in the control plane is responsible for running and managing all controllers, and packages all built-in Kubernetes controllers into a single binary upon initial installation.

Customized Controllers can be created yourself or deployed within your cluster to provide some non-native functionality. Ingress controllers, one of the main focuses of this dissertation, fall under this category. However they are not controlled by the control plane, and must be configured manually to interact with other control plane services [16]

5.3.4 Cloud-controller-manager

This controller-manager allows interoperability between Kubernetes controller functionality and distinct cloud providers, such as Google, Microsoft, Amazon, or other smaller bodies which offer Kubernetes services.

Some of the functionality offered is matching cloud load balancers to Kubernetes services, mapping network communication within Kubernetes to the overarching network within the provider's datacentre

5.4 Node Componenets

Node components, cover general purpose functionality that all nodes need to control cluster actions. Both the master nodes which comprise the control plane and the worker nodes which house pods, all use the following basic resources.

5.4.1 Kubelet

The kubelet is a process which ensures the relationship between pod descriptions and container health. Ensuring that containers are behaving exactly how described, restarting incorrect or failing ones.

5.4.2 Kube-Proxy

The kube-proxy is reponsible for the networking side of the cluster. Controlling transport layer forwarding between pods, port forwarding communication between the cluster and outside hosts, and defines functionality for how the service component operates. It controls how API requests from the client are handled coming into the cluster

5.4.3 Container Runtime

The container-runtime is the underlying software that determines and controls how containers are created, from initial startup and image fetching, to general uptime and monitoring. There are many usable container runtime softwares, with the most popular being Docker and Containerd.

5.5 Cluster Setup

You can setup and install all of these Kubernetes components manually using the Kubernetes open source implementation [17], using the scripts it provides. However this brings about a lot of unforeseen complexity. Setting up a master node for the control plane, initial networking for Kubernetes node discovery, installing all dependencies on each node, joining multiple nodes together to form a cluster and finally setting up client authentication to use the kube-api.

All of this is very complex and generally not too useful if we want to do testing only on a single machine. In fact if you want to set up a multi-node Kubernetes cluster manually, you would need to buy dedicated hardware and spend months configuring conditions of the machines before even beginning.

This is where dedicated setup software helps us. There are tools that exist for the sole purpose of providing an abstraction to the initial setup process. Networking can be simplified by just connecting all nodes to a single network, like in datacentre conditions, and allowing the software to discover them automatically. All installable files can be fetched from the internet and setup on the host machines. Authentication setup is done

automatically, such as TLS certificates for node authentication, and user authentication for the Kubernetes API.

Kubeadm [18] is an example of one of these tools, and reaches an initial cluster state which can achieve a Kubernetes conformity standard

To simplify things further for testing purposes, there are implementations for single-machine Kubernetes clusters specifically, where your node acts as a master node and worker node. These are the ones we will explore for our simple cluster use case.

5.6 Cluster Management Tools

A few tools exist for Kubernetes cluster management. Kubectl is a mandatory CLI tool built directly to communicate with the kube-api server residing in the cluster. It uses a configuration file stored on your host machine to authenticate you as a valid user in the Kubernetes cluster. Once validated, you are able to create HTTP based API requests to the cluster, changing its state or querying state of specific components.

The kubectl tool however, is quite minimal in its graphical representation. This can make it very difficult for someone with no prior Kubernetes knowledge to have a high level understanding of what cluster componenets they are interacting with. A few open source tools have been created since Kubernetes' inception to help provide users with either a more streamlined experience using kubectl, or a complete graphical overhaul.

5.6.1 Kubectx and Kubens

Kubens and Kubectx are tools that simplify part of the kubectl command suite. Within kubectl, chaning between namespaces or even between which cluster you want to make API requests to, requires multiple lengthy commands. Namespaces in particular are frequently changed if you are an infrastructure level developer who requires access to the entire cluster.

Kubectx allows you to list all available clusters referenced in your Kubernetes configuration file, and swap between them, incorporating auto-complete features. Kubens offers the same functionality for namespaces. These two are bundled together in a single binary

5.6.2 K9s

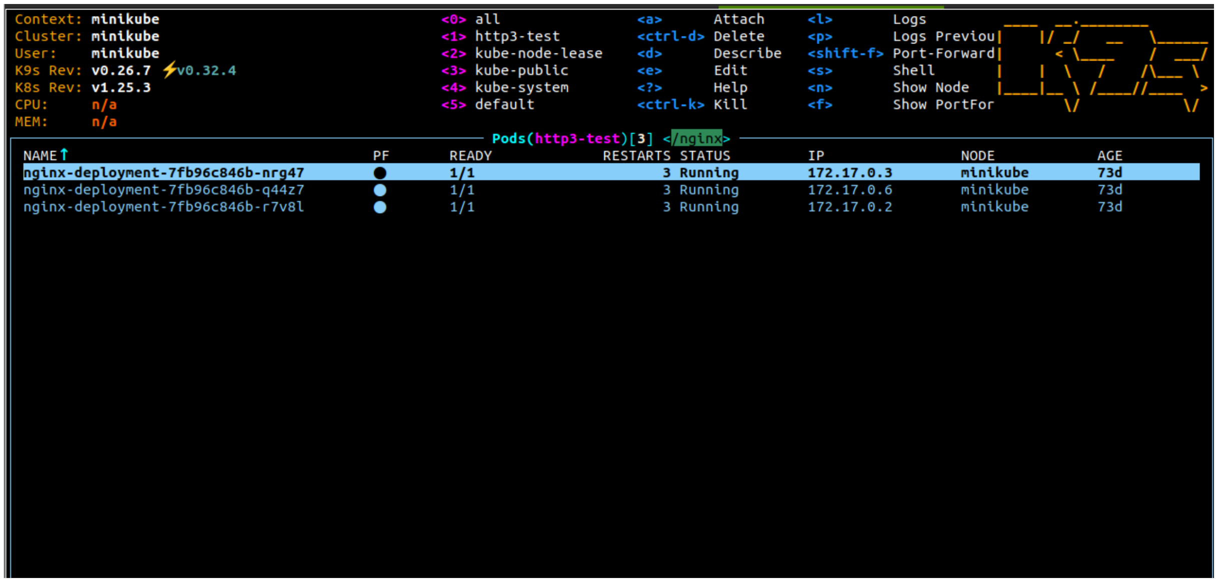


Figure 5.3: Image shows the k9s GUI terminal, focusing on pods in the http3-test namespace, amongst other k9s macro, and cluster information

K9s is an oldschool graphical UI design, simulating terminal interfaces from the early 2000s, focusing on keyboard commands to navigate the interface. It provides a clean area to display the Kubernetes components of interest, along with pre-defined macro commands to easily switch, or search between components. It is considered a very lightweight tool, which does not diverge greatly from the core kubectl command structure. It allows you to execute commands on pods, bring up and delete components, view their defined yaml files, setup port-forwards, among other simple features

5.6.3 Lens

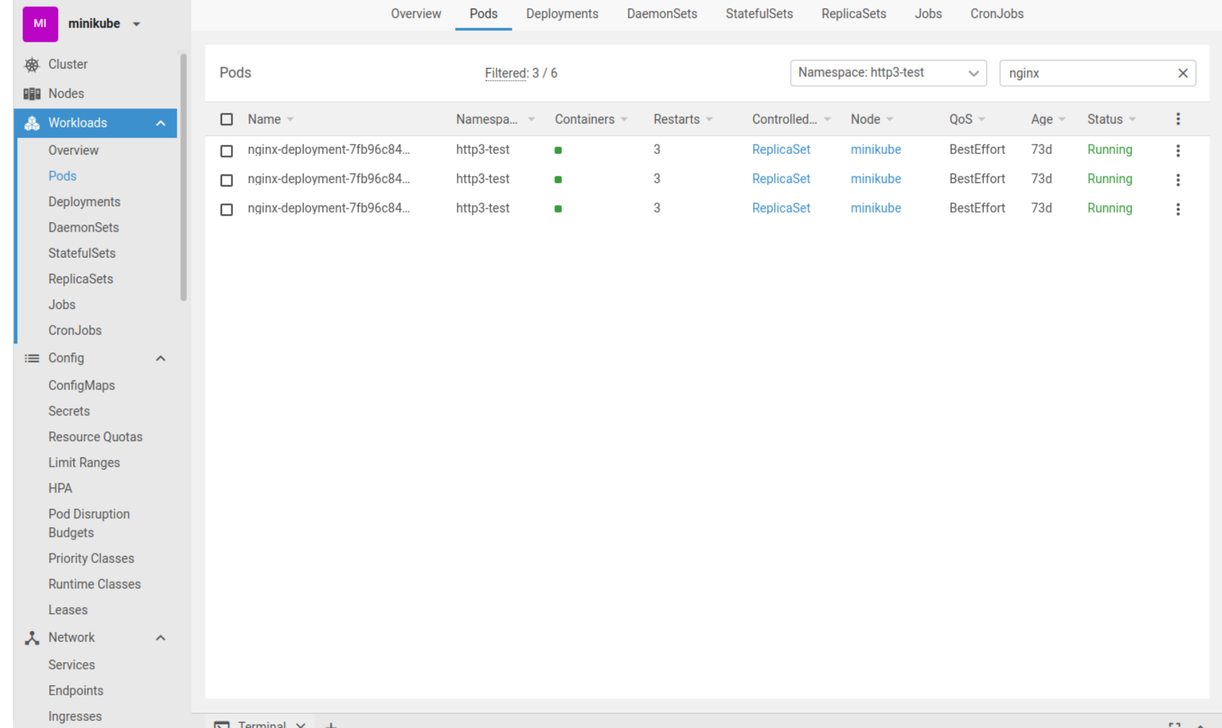


Figure 5.4: Figure shows the Lens advanced GUI, accounting for all Kubernetes resources within the side bar, and filtering down to pods in the http3-test namespace. This mirrors the same filter criteria as the K9s terminal

Lens is a highly advanced Kubernetes graphical interface. Unlike k9s it is very modern and focuses on a mouse based graphical interface. It operates on a freemium model, meaning for personal use the software is free, with extra features provided in a paid upgrade. For enterprise situations, licenses must be purchased for both free and paid versions. On top of the basic functionality provided by both kubectl and k9s, Lens strives to extend the functionality further. It offers the basic features at a comfortable few clicks notice to the user, while providing monitoring features, resource analytics, accessing namespaces within using commands, and making general information extremely transparent. The pro license features offer setting up custom user permissions through the software itself, extended plugins, single-sign-on, and even support for security audits.

5.6.4 Helm

Helm is a templating tool used to remove boilerplate Kubernetes yaml definitions. It allows you to create file-base objects called charts, along with defining a default values

file to populate the templated fields.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ template "fullname" . }}
  labels:
    app: {{ template "name" . }}
    chart: {{ template "chart" . }}
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
spec:
  replicas: {{ .Values.replicaCount }}
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: {{ template "name" . }}
      release: {{ .Release.Name }}
```

Figure 5.5: Figure shows a helm templating format, with the chain bracket contents " X " corresponding to variable in the helm values file.

Within the chart you can create basic Kubernetes yaml files, while including variable definitions within each line of the specificiation. A values file, contains a key-value mapping of these variables. Once you run, or install a helm chart, it will fill the templated variables with the values provided in the default values file, creating valid Kubernetes yaml.

Helm automates the process of installing these yaml files onto your cluster, including intelligent versioning, rollback and history. The broader scale application of this is, similar to images, you can download and install helm charts stored on the internet. This allows open source software to be installed and setup for your specific use-case, simply by providing a single custom values file during installation

5.7 Local Clusters

Local clusters are defined as Kubernetes clusters which can exist on a single host machine. it trades off higher available cluster resources from multiple nodes, in favour of lower setup and maintenance complexity. This lends very favourably for research and small scale environment testing.

5.7.1 K3d

k3s is a simplified, light weight distribution of the open source Kubernetes implementation, developed by a company called rancher. k3d is a community driven iteration, and deploys a docker-based wrapper around the open source k3s. It allows you to deploy a k3 Kubernetes deployment within docker itself. K3s comes with traefikproxy installed as an ingress controller, and a software called klipper to handle load balancing. However, having pre-installed software can lead to some issues when reconfiguring it. I found some cases where overriding the default ingress controller configurations caused issues. In the case of accessing you services from the host machine, mappings of ports from the cluster to the host machine must be defined on cluster creation

5.7.2 Minikube

Minikube is arguably the most popular method of setting up a local Kubernetes cluster. It was created by a Kubernetes special interest group (SIG) which saw a need for local kuberntes deployments for testing. It provides the ability to use a container environment or a virtual machine to drive the cluster creation. Minikube does not come with preinstalled third party software, and can so manual configurations would have a lower chance of poorly interacting with the cluster state. Accessing the services from inside of your minikube k8s cluster can be done from a tunnel command after cluster creation

5.7.3 Kind

Kind stand for "Kubernetes in docker", and is developed by a Kubernetes SIG group also. Its purpose was to provide a dockerized Kubernetes runtime, and follow core principle surround stability of the cluster, longevity of development, and automation. Kind, similarly must define port mappings from the host to the cluster on cluster creation, and could be an arduous process when reconfiguring in testing.

5.7.4 Comparison

Due to my setup requiring simplicity and also configurability, I felt that kind and k3d only offered one of these two in my initial research. Kind and k3d both both require the mappings of ports from the cluster to the host for testing to be defined on cluster creation, and therefore each test would require the reconstruction of the entire cluster. k3d comes with pre-packaged ingress and loadbalancer software, which can be hard to disable or reconfigure. Minikube is the best established local Kubernetes cluster for testing, and

one I am already familiar with from prior projects. As such I have decided to chose it as my local cluster starting point.

5.8 Remote Clusters

Remote Clusters are Kubernetes clusters that are comprised of multiple nodes which are not locally containerised or built up using virtual machines on a single host. There are then two options for implementing a remote cluster. Firstly, buying dedicated hardware and setting up initial operating system installations, necessary software dependencies for Kubernetes, and ensuring nodes are reachable over the local network. Finally, setting up the cluster either manually with the open source Kubernetes implementation, or through setup software like kubeadm, k3s or others. Secondly, opting for a cloud provider, and paying monthly costs. As a student, the expense of providing dedicated hardware, combined with the time commitment of setting up the multi-node Kubernetes cluster, a cloud provider is the only consideration.

The three main considerations are AWS's Elastic Kubernetes service (EKS), Microsofts Azure Kubernetes Service (AKS) , and Google's Google Kubernetes Engine (GKE). The three cloud providers operate on a similar precedence for cluster setup. Account creation with through their web portals, and using a cloud terminal or authenticating yourself to use their command-line-interface from your local machine. Afterwards is cluster setup, which uses virtual networks in their datacentres to configure and connect multiple physical or virtual machines as nodes to your Kubernetes cluster. Finally once the cluster is setup, you update your local Kubernetes configuration to allow you to connect to the cluster. The mapping between a cloud providers own resource to Kubernetes is where they differ the most, since each uses their own native configuration formats or terminal commands. At the simplest level they all provide almost the same functionality

The three main Kubernetes cloud providers differ mainly in their advanced features. Automatic node and cluster updates, provisioning new worker nodes automatically, cluster resource monitoring, and others. In our case we need a minimum viable product, with easy testing setup to showcase the difficulties in applying the QUIC standard. Therefore for cluster research we will focus on ease of setup, cost structure, and research commitment to create a working cluster.

5.8.1 EKS

Amazon's EKS comes with the full backing of AWS's monolithic globally accessible datacentre backbone. Amazon was one of the earliest adopters of the software as a service

business model, which is the movement of computation to datacentres, and offering piecemeal slices of computation, and storage to customers. This allowed them to build a large customer base and as a result pushed engineers, cloud technicians and general software engineers to become highly specialized in using tools within the AWS environment. This relationship between employees and clients dependance on AWS services, caused a level of tech debt in the industry, allowing AWS to monopolize the cloud computing landscape.

Amazon's EKS would lend well to those who have retained their AWS level knowledge regarding AWS native components like Elastic Computer Cloud (EC2), which offers abstracted computing to users, Elastic Block Store (EBS), which is virtual data storage similar to hard drives. These concepts translate directly to Kubernetes, where EC2 instances are provisioned as worker nodes, and EBS is provisioned as persistant storage for services like databases.

Cluster setup

Following the information in the cluster setup guide [3], EKS offers multiple methods to setup a cluster, but their eksctl tool allows you to create one in one step, however, once you have done some prerequisite steps firstly. You must create and configure a virtual private cloud (VPC), which is essentially a virtual network mapping of IP addresses to be assigned to virtual machines (VMs), servers, containers or cluster components. AWS insists you read their heavy documentation on choosing suitable addresses. The documentation further suggests you are made aware of AWS's Identity and Access Management (IAM) service. To simplify essentially ensuring your AWS account has permissions to create a cluster. Finally you can create a cluster with the following command:

```
eksctl create cluster --name my-cluster
--region region-code
--version 1.28
--vpc-private-subnets subnet-ExampleID1,subnet-ExampleID2
```

Listing 5.3: Shows the command to setup an EKS cluster, setting the name of the cluster, the region in the world to be accessed from, the Kubernetes version, whether the subnet for the cluster nodes should be accessible from the internet, –vpc-private-subnets defines private only. You can define a mix of both however

Once the cluster is created, you can receive the authentication information from the cluster to query it with your kubectl CLI locally. The difficulty of the EKS management

comes with the cluster Kubernetes components themselves. For HTTP setup you must assign a multitude of AWS components to map to your Kubernetes services. For configuring an ingress controller to be accessible outside of an EKS cluster, there are many complex steps to take. First, an AWS network load balancer has to be assigned to the cluster, a Nodeport Kubernetes service must be configured, which maps incoming traffic from the nodes to your own services instead. The AWS network load balancer must be configured manually to map for each port we want to expose from our Kubernetes service, to our node, then to the external load balancer. Finally a security group, which is AWS's network traffic control component, must be configured to allow traffic along these ports to the cluster.

Each and every step is manual, which you can argue strengthens a users knowledge about these core networking topics, and especially applied to AWS. However, it incurs a longer setup time, especially considering if we are only interested in simple HTTP3 testing.

Cost Model

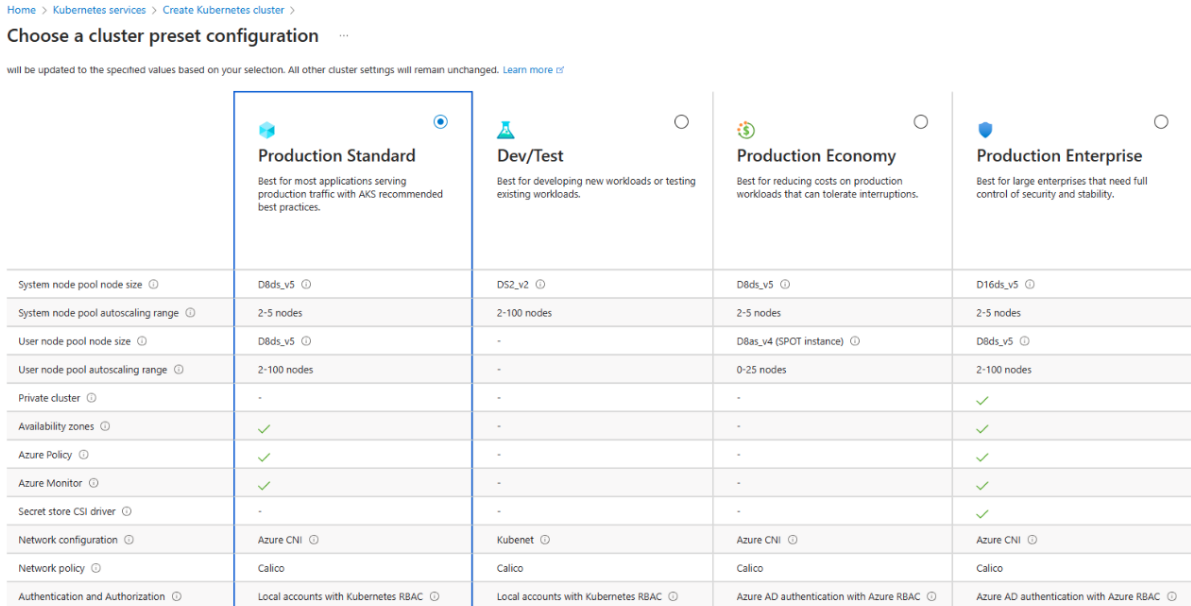
AWS does offer free 12 month use for EC2 instances, amongst other features, which allow you to assign worker nodes to your Kubernetes cluster. However, they charge you a fixed 10 cents per hour for cluster maintenance and uptime, which is unavoidable.

5.8.2 AKS

AKS is offered by Microsoft's Azure Cloud Service. I believe microsoft have managed to adopt google's Kubernetes software well into their own native cloud infrastructure.

Cluster Setup

AKS has an incredibly simple setup process for its cluster environment, and offloads any required knowledge as optional reading. This could be considered unfavourable, especially if you encounter more complex issues after cluster creation, but for our use case, a simple setup is optimal. Microsoft allows you to use either their native Azure terminal interface or a web portal to fully configure the cluster. The web portal configuration documentation is succinct and clear.



	Production Standard Best for most applications serving production traffic with AKS recommended best practices.	Dev/Test Best for developing new workloads or testing existing workloads.	Production Economy Best for reducing costs on production workloads that can tolerate interruptions.	Production Enterprise Best for large enterprises that need full control of security and stability.
System node pool node size	D8ds_v5	DS2_v2	D8ds_v5	D16ds_v5
System node pool autoscaling range	2-5 nodes	2-100 nodes	2-5 nodes	2-5 nodes
User node pool node size	D8ds_v5	-	D8as_v4 (SPOT instance)	D8ds_v5
User node pool autoscaling range	2-100 nodes	-	0-25 nodes	2-100 nodes
Private cluster	-	-	-	✓
Availability zones	✓	-	-	✓
Azure Policy	✓	-	-	✓
Azure Monitor	✓	-	-	✓
Secret store CSI driver	-	-	-	✓
Network configuration	Azure CNI	Kubenet	Azure CNI	Azure CNI
Network policy	Calico	Calico	Calico	Calico
Authentication and Authorization	Local accounts with Kubernetes RBAC	Local accounts with Kubernetes RBAC	Azure AD authentication with Azure RBAC	Azure AD authentication with Azure RBAC

Figure 5.6: Figure shows the multiple cluster configuration options for an AKS cluster, ranging from Standard, Dev, Economy and Enterprise level resource allocation

They are the only provider that have distinct cluster presets for testing purposes and whose step-by-step cluster creation process does not require any additional documentation research to complete. Similar to all other options, once the cluster is created, we receive the cluster API credentials into our Kubernetes configuration and can begin creating components

5.8.3 Cost Model

Azure offers two-hundred dollars of free credits for cluster computation. Cluster management itself is free, so once your credits are expended, you can retain your cluster if you do not use cluster resources

5.8.4 GKE

Kubernetes was originally created by Google in 2013 to compete directly against AWS's monopoly on the global cloud infrastructure. It's efficacy in server resource management has caused it to have a global cult following. Similar to how AWS has spawned workers and clients to have a dependence on their software, Kubernetes has become so deeply ingrained in infrastructure management, that both Microsoft and Amazon offer their own Kubernetes managed clusters. Google comes with the advantage of being the pioneers of

both the Kubernetes environment, and the development of HTTP3 and QUIC. This potentially adds an extra level of stability to the maintenance and setup of the environment.

Cluster Setup

The google cluster setup follows the standard principles of creating and registering an account with the Google Compute Enginer. The installation of the CLI locally [12] and authentication by logging in with you google account. You must then configure your gcloud configuration to use a region for deployment. The GKE documentation is very streamlined and does not force you to use cloud native components or terminology like AWS. It recommended a private cluster, which can be configured to be accessible by authorized IP addresses. VPCs and subnets are also created automatically

```
gcloud container clusters create dissert-cluster
--create-subnetwork name=my-subnet1
--enable-master-authorized-networks
--enable-ip-alias
--enable-private-nodes
--master-ipv4-cidr 192.168.0.0/28
--num-nodes=1 --project http3-test-in-gke
```

Listing 5.4: The following is the command for creating a google cloud cluster, creates subnets automaticaaly. ‘enable-master-authorized-networks‘ is used to later configure our host machine to access the kube-api publically. ‘enable-ip-aliases‘ allow secondary IPs to be assigned to pods, not just nodes. This allows traffic to be routed to pods without the nodes processing them. ‘enable-private-nodes‘ creates nodes which are not publically accessible

Once your private cluster is configured, within the google cloud portal, you can add your public ip address to the cluster’s authorized IP addresses. Google automatically configures external load balancers and assigns a public IP address if it sees an existing loadbalancer service within the cluster, circumventing the heavy setup that EKS requires to communicate with a service inside the cluster.

Cost Model

GKE offers three-hundred dollars of free credits to first time users of the Google Cloud platform. While the cluster is prices at 10 cents per hour like EKS, the free credits cover

this.

5.8.5 Comparison

For GKE, the resource mappings of the cloud resources to the Kubernetes cluster componenets is more seemless than AWS due to Kubernetes being a Google created software.

Cluster	Research Commitment	Documentation Complexity	Setup Complexity	Service Complexity
EKS	High	Med	Med	High
AKS	Low	Low	Low	Low
GKE	Med	Med	Low	Low

Table 5.1: Comparison of Kubernetes Environments between major cloud providers. Indentifyin Research Commitment needed to setup the cluster, documentation complexity, cluster setup complexity, and setting up a remotely accessible service

Cluster	Pricing	Free Credits
EKS	0.10/hr Cluster, Free Compute	None
AKS	Free Cluster, $0.10 to $1 per hour	$200
GKE	$0.10/hr Cluster, Variable Compute	$300

Table 5.2: Comparison of Kubernetes Environments between major cloud providers. Focus on cluster management and compute pricing, along with provided free credits

AKS offers the least complexity, setup time, and generous pricing model. EKS is arguably too complex, as the environment is reliant on competency with core AWS cloud concepts. GKE sits in the middle, but benefits from both stability of Kubernetes environment, as Kubernetes was built for google's cloud environment, and may offer some more streamlined support for HTTP3. Overall if simplicity of use and cluster setup is your primary goal, and you do not have prior experience in the other cloud providers' environments, choosing Azure's Kubernetes Service will ease the research commitment needed to begin your core HTTP3 testing.

I have decided to use the GKE environment as AKS does not offer much more ahead of GKE, as the pricing model covers the time scope of the dissertation period, and the slight

complexity in documentation is negligible when we consider only simple cluster setups are required in any case. The potential benefits of using a cloud environment designed for Kubernetes may prevent issues in implementation.

Chapter 6

State of the Art

This section showcases to the reader the adoption level of QUIC and HTTP3 globally, and how performance metrics are inconsistent depending on implementation and configuration of the protocol, despite following the RFC standard. We identify how Google's creation of QUIC has given them an eight-year advantage in implementing and improving upon the use of the protocol in their environments, especially in comparison to both their main market competitors. Furthermore, we conclude that Google's monopoly over browser use and search-related internet traffic, skews the perceived success of the HTTP3 protocol across the internet

6.1 QUIC and HTTP3 Usage Statistics

With how revolutionary QUIC claims to be, improving upon and upending TCP's reigning monopoly on the transport protocol networking scene, we have to question how widely deployed is this next-protocol giant. Cloudflare is a global content delivery network provider, providing a range of services to clients, such as DNS resolution for web domains, regional and local network load balancing, and malicious attack prevention services. Cloudflare being such a large central point for internet traffic across the world, allows them to collect and reveal regular internet metrics. For one such case, Cloudflare offers usage statistics targeting internet browser traffic [8], specifically HTTP3 traffic.

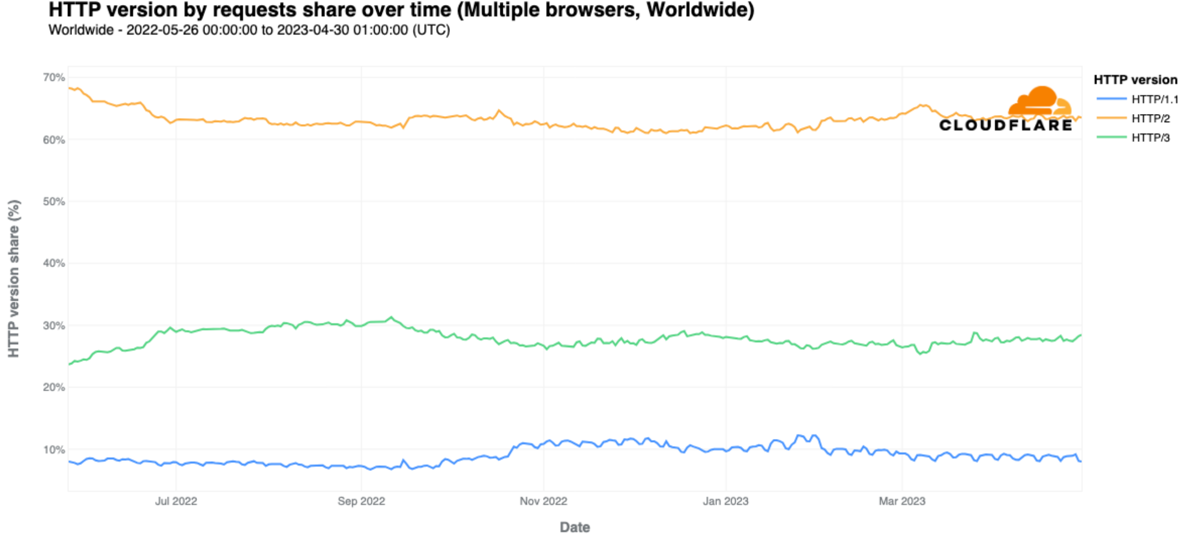


Figure 6.1: Figure shows cloudflares HTTP version share over global internet traffic towards their endpoints. Comparisons between HTTP/1.1 HTTP/2 and HTTP/3 are made

Despite the HTTP3 protocol becoming publically standardised in May of 2021, HTTP3 traffic has grown staggeringly, accounting for 25-30% of global internet traffic. Surely this must be a good indication of how a superior protocol can quickly replace even the most widely used TCP.

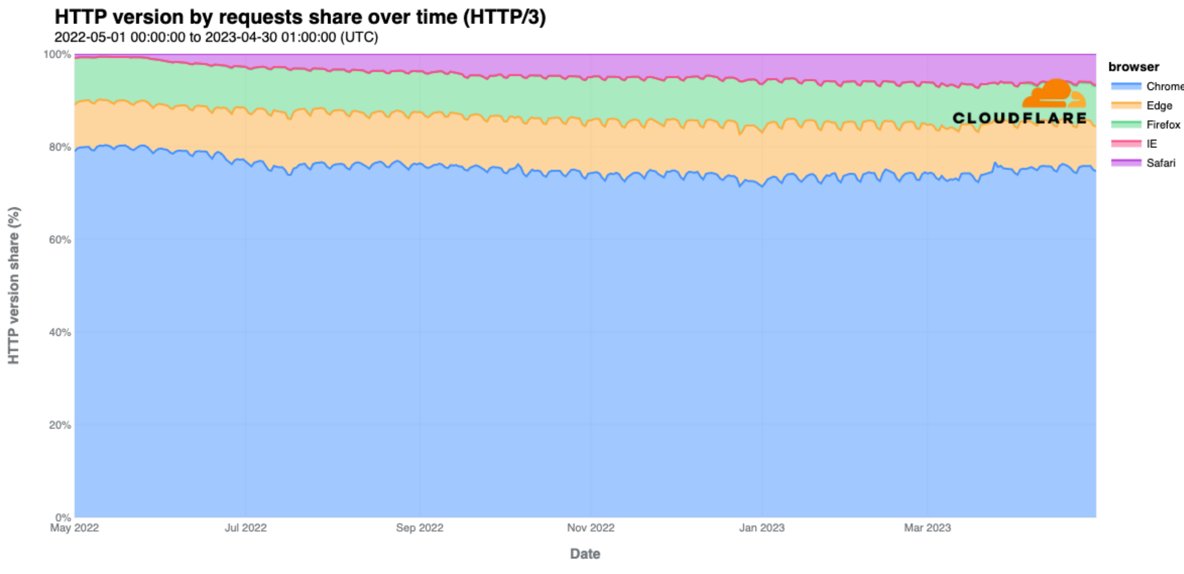


Figure 6.2: Figure shows HTTP/3 usage over cloudflare public endpoints, on a per browser basis. Comparisons between Chrome, Edge, Firefox, Internet Explorer, and Safari are made

In all of the above browsers, HTTP3 has become a natively supported software. However, for stability and interoperability, browsers will still communicate with remote web servers using older HTTP2 or HTTP1.1 version in order to establish a basic level of communication, and following it, upgrade to HTTP3 using the ALPN protocol if the web server supports HTTP3. Focusing solely on the HTTP3 traffic, Google spearheading the HTTP3 movement is unsurprising. Google began use of HTTP3 through QUIC during its first inception in 2013, essentially providing them an eight year head-start in protocol development, testing and configuration.

6.2 QUIC Implementations

Implementations of the QUIC protocol must adhere to the RFC standard regardless of whichever language they are to be implemented in. When choosing a language for an implementation, it is imperative that quirks or considerations of the language do not affect or alter any of the core underlying functionality required by the protocol. We will explore some of the most prevalent implementations in the landscape, and additional functionality they offer, which may assist us during our testing phases

6.2.1 AIOQUIC

AIOQUIC [1] is a python based QUIC implementation, which uses the asyncio python library to allow asynchronous communication handling. It attempts to rigorously follow RFC standards forthe QUIC protocol, and contains a custom TLS 1.3 implementation to account for QUIC being deeply routed directly with TLS, to achieve its 1 and 0-RTT communication. It provides an RFC compliant HTTP3 stack to interface with the defined QUIC components, and finally offers client and server python files which offer examples of use. It was created by Jeremy Lainé, a github user with a large background in telecommunications and networking.The core principle of his implementation is to provide an input/output (IO) free QUIC service. This means it is up to the user to develop any application level communication using his implementation, and only functions and objects which encapsulate QUIC's functionality are offered to the user. AIOQUIC acknowledges the difficulties QUIC produces for testing environments, both with its mandatory encryption and its lack of packet dissection in the open source space. The implementation offers the output of TLS session keys for decryption, ignoring invalid TLS certificates, qlog trace outputs, and lengthy examples to headstart local testing

6.2.2 Quic-go

Quic-go [29] is a QUIC implementation written in the Go programming language. It was originally created by github user Marten Seemann. It offers an RFC complient implementation, includes TLS session key outputs, and ignoring invalid TLS certificates, however does not offer qlog trace outputs. Compared to AIOQUIC its documentation is not as concise, and requires a stronger background in go before using it correctly. It is difficult to get any notable specifics about the quic-go implementation besides that it follows the RFC standard.

6.2.3 Comparisons

While QUIC implementations adhere to the most recent draft from the RFC standardisation of the protocol, it begs into question where the differences lie. Language specific implementations allows easier interfacing with language specific services, exisiting in already established architectures for large businesses. Some sections in the RFC standard are guidelines, and not explicitly mandatory for a compliant implementation. Specifically, different congestion control and flow control algortihms are suggested, where the exact specifics can be chosen by individual implementers, while still adhering to the QUIC standard. A paper regarding QUIC performance comparisons [34], identifies the key differences between QUIC/HTTP3 performance when compared against Google, Cloudflare, and Facebook hosted endpoints.

It concludes that the congestion control algorithm chosen heavily determines the performance of HTTP3 against prior TCP-based HTTP communication. Google's public endpoints performed more consistently, with improvements for small and large data transfers, multi or single-stream QUIC transfers, and packet loss scenarios. The most notable observations were the congestion control algorithm used by the QUIC server, The TLS configuration on the client being cohesive with the server, and ensuring QUIC communicates with 1-RTT, offered the most improvements over TCP. Considering Google Chrome accounts for 65% of browser usage, Google search accounting for a large portion of public internet traffic, and finally, Google having a large lead in QUIC implementation and testing, It is no surprise that Google performed the best when testing a variety of QUIC clients against their public endpoints. While it seems QUIC does offer notable improvements over its forty year old predecessor TCP, the overhead in protocol complexity and migration of existing services, causes reluctance in other mainstream competitors. This observation could be reflected in Cloudflare's browser share staistics of HTTP/3 internet usage 6.2. Perhaps in the next few years we will see more consistent performances of the

QUIC protocol across other large scale vendors

6.2.4 Implementation Considerations

Considering how the performance of individual QUIC implementations is heavily dependant between client and server configurations, and especially server congestion control choices, using an implementation with extended testing functionality would be ideal. AIOQUIC fits this critera, is implemented in python, which is very readble, offers verbose examples for how to setup local testing, and includes useful testing features, including qlog trace outputs.

6.3 Middlebox Ossification

Google have stated QUIC fundamentally tries to tackles the issue of internet ossification. Internet ossification is the act of middlebox machines, or routers between two end users having free reign to modify and route packets by their packet headers [20]. It highlights the conservative nature of internet routing, and packet header manipulation is highly determined on already applied and existing standards at the time of configuration. The introduction of new transport layer protocols, or the modification of existing ones, can cause non-deterministic errors in network routing across the internet. Poorly configured middleboxes may not recognize updgraded protocols, and may not support new ones.

QUIC addresses this issue partially by using UDP as a substrate, whilst QUIC is a transport layer protocol by design, it exists within user space and not directly within the operating system's kernels. This allows its native TLS integration to provide end-to-end encryption between users, further limiting what parts of QUIC's headers are modifiable by middleboxes. ALPN protocol negotiation is used as fallback mechanisms. However, even considering these measures taken, during protocol development and testing within Google in 2016, a single bit change in a public header caused catastrophic networking issues for specific users behind a firewall with explicit QUIC blocking. The single bit change caused the middlebox to not detect these packets as QUIC packets and allowed them to bypass the firewall [19]

6.4 QUIC packet flow

Analysing the packet flow from the client-server example from the AIOQUIC examples provides a good insight into the key mechanisms presented within the QUIC protocol and the general guidelines we expect when testing the protocol within Kubernetes.

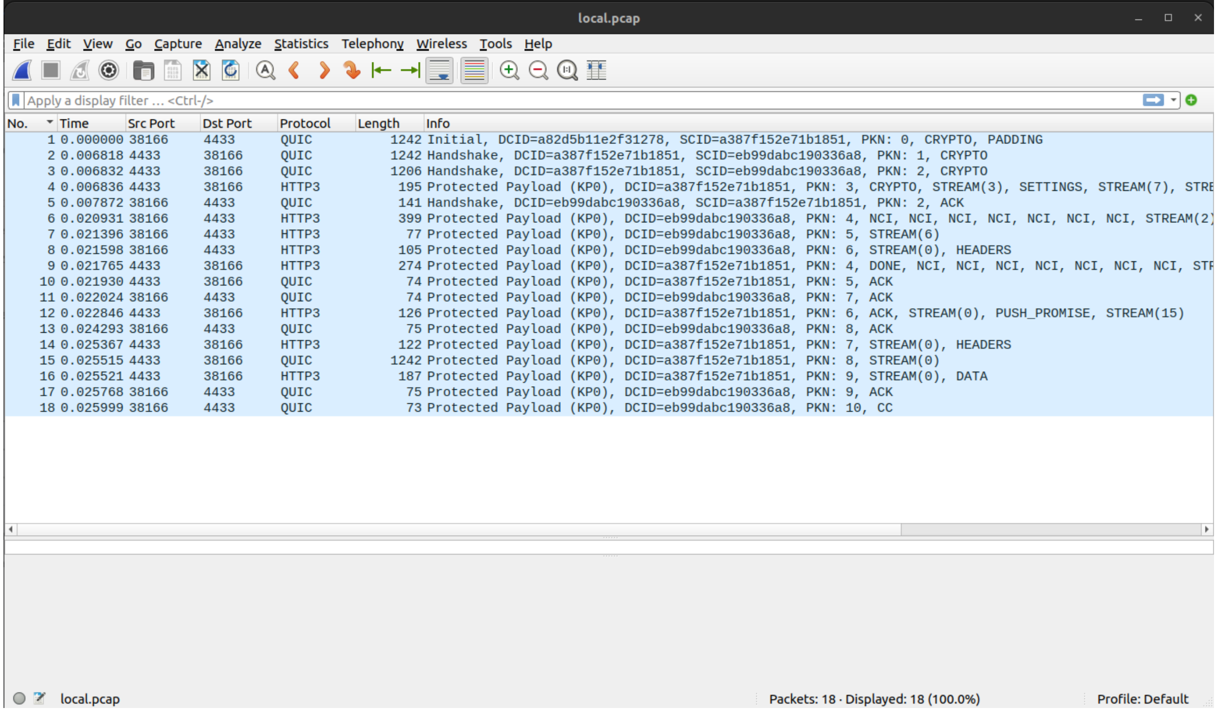


Figure 6.3: Figure shows Wireshark packet capture, showing the results of a QUIC client-server session. Each individual line number corresponds to a UDP datagram sent or received. The QUIC client communicates on port 38166 and the server listens on port 4433

We will describe deeply each individual packet in the example, referring to the datagram in the figure 6.3 above. At a higher level, the main communication flow is a handshake for QUIC and TLS. A HTTP3 get request for the default test hyper-text-markup-language (HTML) for AIOQUIC. Finally, a HTTP3 server push, which is when the server sends extra information to the client without the client asking. It does this in cases when the initial get request has additional objects or data associated, in our case it is a cascading-style-sheet (CSS) file to accompany the HTML. For each packet, we will label it with the corresponding UDP datagram number on the left-handside of 6.3. Furthermore, for transparency, the next sections will be labelled to describe the direction of the packet flow, client to server (C-S), or server to client (S-C).

Packet 1 C-S: Client Hello packet with TLS information

The Client sends a single UDP datagram to the server, inside is a single QUIC packet containing a QUIC CRYPTO frame. The CRYPTO frame contains TLS Client Hello Handshake information. It also contains an extension that follows the Application Layer Negotiation Protocol (ALPN), telling it the versions of HTTP3 it supports. The TLS Client Hello contains information regarding what cryptographic algorithms are supported

by the client, what methods of key exchange are to be used, and the initial key exchange information to generate a shared secret.

Packet 2 S-C: Server Hello, ACK, and ALPN Negotiation

The Server sends two QUIC packets within a single UDP dataframe to the client

QUIC packet 1

The first packet contains a Server Hello Message within a QUIC CRYPTO frame. This contains the chosen cryptographic algorithms from the ones provided by the client, and key exchange data for the client to use to generate a shared secret. The packet also returns an acknowledgement (ACK) that the server received the Client Hello

QUIC packet 2

The second packet contains information for the Application Layer Negotiation Protocol within a QUIC ACK frame. The server has negotiated with the client that it will use HTTP3 for further communication.

Packet 3 S-C: Further Handshake Information

The server sends TLS certificate information within a QUIC CRYPTO frame. It contains the digital certificate and a cryptographic verification that the certificate is tied to this TLS session. It also indicates that the handshake is finalised from the server's side. The client now can verify the digital certificate against a root CA. Both the client and the server have enough TLS session information to construct a shared session key.

Packet 4 S-C: Server Sends Stream Creation Packets

The server sends a QUIC packet and three HTTP3 packets in a single UDP datagram. The QUIC packet contains a CRYPTO frame with a TLS session ticket, so the client can re-initiate communication with the server faster in the future. The QUIC packet contains three STREAM frames. In our case, they create 3 Uni-directional QUIC streams to the client. The three HTTP3 packets define how each QUIC stream is configured. By default HTTP3 requires each peer to set up one control stream. This stream has the role of controlling flow, error handling and parameter updates for individual future streams. The two other streams are the encoder and decoder streams, which handle the instructions for encoding and decoding the data travelling across the communication line.

Packet 5 C-S: Client Acknowledges The Handshake

This packet contains an ACK frame to acknowledge the handshake packet received in the third UDP datagram above.

Packet 6 C-S: Client HTTP3 Streams and New Connection IDs

Similar to packet four from the server, the client creates three QUIC streams and three HTTP3 streams to link them. It creates the same control, encoding and decoding stream. Similarly, the client creates 7 NEW_CONNECTION_ID frames within the QUIC packet. These are made available for the future if connection migration will occur. The client also sends the max push ID along the HTTP3 control stream. This gives permission for the server to make push promises.

Packet 7 C-S: Client Encoding Stream

The client has used stream 6, which maps to the HTTP3 encoding stream defined in UDP datagram 4 above. This means the client is sending encoding instructions to the server. The Wireshark dissector does not provide the means to interpret the hexadecimal values associated with this unfortunately.

Packet 8 C-S: Client HTTP3 Get('/')

The Client created a short header QUIC packet. This indicates non-handshake or connection setup related information is being sent. Stream 0, a bidirectional stream is created with the FIN flag set, indicating the client will not use this stream further after this first use. A HTTP3 packet is within the datagram also. It defines a HTTP3 get request on the '/' url path.

Packet 9 S-C: Server New Connection IDs and QPACK encoding

Within a QUIC packet it creates 7 NEW_CONNECTION_ID frames in case of connection migration, this is a mirror of what the client did and is standard practice. A HTTP3 frame uses the unidirectional stream 7, which maps to the encoding stream for the server. This also mirrors the client

Packet 10 S-C: Server ACK for Client Encoding Stream

The server acknowledges the client encoding stream data, and implicitly all previous non-acknowledged packets before that one. Which is Datagram 7 and prior on our pcap.

Packet 11 C-S: Client ACK for Server Encoding Stream

The client acknowledged the server encoding stream and all previous non-acknowledged packets. This is datagram 9 and prior from the server on our pcap.

Packet 12 S-C: Server Push-Promise and Stream

The server sends a push-promise to the client, indicating the data it asked for in the get request, will be accompanied with other data it did not ask for. In this case the HTML it requested will be accompanies with a CSS stylesheet. The server sends this information on the bidirectional stream 0. The server then created a stream with id 15, to communicate the current push_id to the client.

Packet 13 C-S: Client ACK Server Push Promise

The client acknowledges the Server Push-Promise.

Packet 14 S-C: Server HTTP 200 OK

The server sends a HTTP code 200 to indicate its initial HTTP get request was received correctly.

Packet 15 S-C: Server Sends First Data Fragment

Server sends a UDP datagram with a fragment of the total HTML data to be sent to the client. It sends 1167 out of 1276 bytes. This contains the HTTP3 payload containing an example HTML landing page from AIOQUIC.

Packet 16 S-C: Server Sends Second Data Fragment

Server sends a UDP datagram with a fragment of the total HTML data to be sent to the client. it sends the remaining 112 bytes. The HTTP3 return request is reassembled by the client, with the two fragments, to create the final HTML. At this point, the HTML would be loaded if using a web browser as the HTTP/3 client.

Packet 17 C-S: Client Acknowledges

Client acknowledged the data and all prior unacknowledged packets.

Packet 18 C-S: Client Initiates Close Connection

Client sends a CONNECTION_CLOSE packet to cease communication with the server. It seems the server push was never fulfilled, perhaps because I ran the command in a command-line instead of a web browser.

6.5 Ingress Controllers

Ingress controllers fall along the category of controllers within Kubernetes. They are not controlled by the native kube-controller-manager service but are third party controllers. Ingress controllers are responsible for dynamically controlling how Kubernetes ingress is handled. This means they determine how connections from outside of the Kubernetes cluster are handled and distributed once entering the cluster.

Ingress controllers offer some basic functionality such as load balancing connections, forwarding network packets, and TLS authentication on behalf of backend webservers. Unlike normal ingress definitions within Kubernetes, controllers add an additonal level of complexity and allow more dynamic route setting. They can support auto-scaling and automatic TLS certificate renewal to allow a Kubernetes cluster to have high-availability (HA) networking, mimiking a production level service architecture.

Ingress controllers allow some level of TLS handling customization. There are three main features, TLS termination, which decrypts the incoming encrypted packets, before forwarding the plaintext data to the back-end server. TLS passthrough, forwards the encrypted packets directly to the back-end server, based on un-encrypted header information. Finally, TLS bridging, which terminates the initial TLS communication, but the the proxy establishes a new TLS encrypted connection with its backend servers.

TLS termination is standard practice, as it prevents back-end servers from handling computationally intensive decryption algorithms, and allows the proxy or load balancer to do advanced routing based not only on header, but only payload information. Google themselves use the QUIC protocol with TLS termination at their front end servers [19]

6.5.1 Nginx

Nginx is a open source software created in the early 2000s as high-performance, scaling web server. It was specifically created to solve early internet congestion issues once webservers reached over ten-thousand concurrent client connections.

Nginx later evolved to offer proxying, TLS authentication, and load balancing properties. Essentially allowing an Nginx webserver to be loadbalanced by another Nginx load

balancer. Its versatile design has made it one of the most popular webserver and load balancing services in the world.

Nginx later became a proprietary software, with its original open source version offering less functionality than its paid version. Some open soruce developers took it upon themselves to improve the core nginx software with a scripting language known as Lua, to allow more dynamic configuration, further customized extensions, and additional monitoring and metrics analytics. This Lua modified Nginx distribution is called Openresty, and is the Nginx distribution used in the natively supported ingress-nginx ingress controller for Kubernetes

HTTP3 Support

The core nginx software released experimental HTTP3 support during May 2023. It allows defining configuration files for load balancing and webservers to use HTTP3. It recommends the use of the same port for HTTPS and HTTP3 for ALPN protocol upgrades over the same connection.

Unfortunately the Openresty distribution of nginx, which is used by the Kubernetes ingress controller, only recently updated its core nginx version in late August 2023 [23]. Furthermore, despite it being successfully integrated, the Openresty team have decided to not support HTTP3 due to compatibility reasons with their existing Lua scripting, claiming multiple components of their software fail when using HTTP3 over QUIC. For this reason, ingress-nginx for Kubernetes does not yet offer HTTP3 support.

This highlights the interdependence of open source software. Despite the Nginx core open source distribution allowing HTTP3 support, Openresty, which is made for the sole purpose of extending the core version with scripting and additional functionality, does not successfully implement some of the originals core functionality. The scope of change brought about by the QUIC protocol is to great, even despite QUIC using UDP datagrams for the purpose of interoperability with existing networks.

6.5.2 TraefikProxy

TraefikProxy is a reverse proxy server, which promises real time routing updates at a simplistic level, with its unique selling point being automatic discovery for services. It has four main components, entrypoints, routers, middlewares, and services. Entrypoints are defined as port mappings for initial connections, they define how communication enters the TraefikProxy space. To simplify, you can separate HTTPS, HTTP, UDP, TCP and others communication protocols with entrypoints, to allow traefik to handle these types of requests in unique manners. Entrypoints can be defined for individual domains and

web paths also.

Routes control how to further route information once it has gone through an entry-point, and map it to an existing back-end service. This allows you to separate further individual packets, web paths or do authentication if credentials are required for a specific web path.

Services act as the final front before your individual servers. They can act as load balancers, forward packet headers for dissection, or configure web cookies for a client to always communicate with the same server. Services in TraefikProxy act very similar to services in Kubernetes

HTTP3 Support

TraefikProxy implements HTTP3 over quic using the quic-go library. It claims to offer client to proxy HTTP3 communication by default, with the TraefikProxy service terminating the TLS communication, finally forwarding unencrypted data to its load balanced servers. The documentation for Traefik only loosely discusses configuration for Kubernetes, and does not contain an explicit sentence regarding HTTP3 support within the environment.

6.5.3 Comparison

TraefikProxy and Nginx fundamentally offer the same service when considered at a high level, simple configuration methods for ingress fields and the architectural objects that provide these services are different. The main comparison of note is the interdependence between Nginx, Openresty and ingress-nginx for Kubernetes. TraefikProxy has claimed HTTP/3 support with its quic-go implementation, but does not explicitly define how to configure it within the Kubernetes environment.

6.6 Summary

Within this chapter we have both identified state of the art QUIC implementations, and Ingresses controllers, along with broadening our viewpoint of QUIC and its adoption. The comparisons between QUIC performance in public enterprise domains and browser usage through Cloudflare statistics, offers a baseline for mirrored comparisons when viewed at the open source level. Furthermore, we identified a foundation for a QUIC packet flow, which we can reference as our standard, when applying QUIC in different environmental areas.

Chapter 7

Problem Formulation

7.1 Problem Formulation

With reference to our state of the art discussion, HTTP3 over QUIC communication is viable only within proprietary environments, and even within them, Google offers the most stable level of QUIC client-to-server communication. Safari, Firefox, and Microsoft Edge lag behind in implementation, and identified performance irregularities may play a large part in the slow adoption.

Within this dissertation we aim to identify potential causes for this reluctance to switch. We focus on the open source implementations of QUIC communication, and through the use of ingress controllers within the kubernetes environments, we can simulate production level service configurations. Despite the RFC compliance of these standalone QUIC implementations, and the HTTP3 support claimed by these ingress controllers, we can highlight how the implementation discrepancies in the large scale comparison between Google, Facebook, and Cloudflare, are both mirrored and exacerbated when testing the standard in local Kubernetes environments.

The dissertation plans to address and test these issues through a piecemeal approach. This entails starting at local testing between two local RFC compliant QUIC implementations, and observing packet communication to establish a baseline standard for communication.

Furthermore, we will push this communication to intra-service communication within kubernetes. Finally, we will extend this to production-stable environments, using a well defined ingress-controller to handle communication from outside of the cluster inwards.

At each stage, we will identify not only any discrepancies between packet communication, but also difficulties in testing, issues observed in the ingress-controller through enabling HTTP3, and also kubernetes environmental issues through using HTTP3.

The dissertation hopes to paint a picture of why the migration from the outdated but wide-spread TCP protocol to QUIC is difficult, and how a globally perceived success globally, is actually achieved solely by Google's proprietary years of iteration and testing.

Chapter 8

Design

The following sections covers the core components needed to apply the HTTP3 and QUIC communication standard within a Kubernetes environment. This identifies the ideal starting point to make further

8.1 System Architecture

In order to design a cohesive system to test both the proper behaviour of the implemented protocol, while also testing the ease of implementation and protocol use, we must ensure a simple setup which contains the minimum viable needed services and configurations.

We need a basic kubernetes cluster, comprised of a single master/worker node, either as a single node combined, or two total, containing all initial kubernetes resources to ensure good cluster health. Within this cluster we would require simple backend servers provided through a group of deployments and a service to access them. An implementation of our chosen ingress controller, TraefikProxy, defined by configuration files that allow the service heading the backend servers connections.

The Ingress controller must then be reachable from a AIOQUIC client service, initiating HTTP3 QUIC connections. This can be done either through a portforward of the TraefikProxy service to the host machine if the cluster is local. Additionally, if the cluster is remote, an external IP address can be assigned to the hosting pod or service to provide the same entrypoint.

With this, HTTP3 requests should be able to be served.

8.2 System Transparency

Once the initial network components are deployed, we must provide additional functionality for packet capturing and network transparency. Ensuring we can collect enough information to make educated decisions on both cluster performance and HTTP3/QUIC behaviour.

8.2.1 Packet Capturing

Due to the always-encrypted nature of QUIC connections, packet capturing requires us to be able to decrypt these communications. Therefore we must use a QUIC implementation that is configurable to allow TLS session keys to be extracted, along with the pcap packet format. Depending on whether TLS passthrough is used or not, a packet capture must occur at one, or two stages of the connection pipeline. If passthrough is used, we must capture packets at the client, and the backend server, or if possible, solely at the proxy server, which is the middleman in our pipeline. The AIOQUIC client offers functionality to dump TLS session keys, along with ignoring TLS cert authentication for the remote server. Similarly, the AIOQUIC server provides this functionality. Therefore we do not need to capture packets on the ingress controller.

8.2.2 Packet Visualisation

We must be able to identify and separate individual packets and their payloads. Tcpdump will allow us to generate pcap files for wireshark visualization, similar to figure 4.1. Combined with the TLS session keys from our AIOQUIC client, we can identify minute details in each packet on the network. Similarly, we can use the QVIS tool in combination with wireshark, to identify individual QUIC frames and how they induce state changes within the client and server 4.3. This offers a clearer picture of client and server separation

8.3 QUIC Client Connections

Due to the current use for QUIC connections as primarily as an experimental web feature, and rarely used in an infrastructure service based communication network, rarely is QUIC used as a primary communication protocol by default. Within the RFC9000 specification itself, and Google's pre-RFC specification [19], a function for backwards compatibility was defined. By default servers will primarily work on standard HTTP2 or TCP based connection mediums, allowing for protocol renegotiation to a QUIC connection. This ensures optimisation for the common case. ALPN protocol negotiation is an inherent

part of QUIC's implementation. Depending on the tool or browser implementation, we can attempt to force HTTP3/QUIC only connections, ensuring the client attempts to connect immediately with our preferred protocol. The caveat in a real world scenario would be, you are not guaranteed to connect to a website if it does not support QUIC, or does not support direct HTTP3 in its underlying webserver implementation.

8.3.1 Web Client

AIOQUIC provides some suggested flags to be configured within chrome based web browsers to allow these required features. Appendix A.1 showcases AIOQUIC's recommended flags to be set on the browser level, to allow chromium based browsers to communicate with the AIOQUIC server without upgrading their connection from an older HTTP version. Appendix A.2 shows how to generate an encoded fingerprint based on the servers digital certificate associated public key.

8.3.2 CLI Client

The CLI client must be either native to a full RFC standard QUIC implementation, or interface directly with one. Similar to the web client, we want the ability to interface directly with the webserver without upgrading from an existing HTTP2 connection. Ignoring digital certificate authentication helps us save the time needed to setup any sort of TLS server certificates. Since there is no graphical interface, the client must have a method to output the TLS session keys that to allow us to decrypt the QUIC traffic and view it in a packet viewing software.

8.4 Kubernetes Setup

Kubernetes is the environment we will apply our standard in, as in the industry it is the gold standard for container orchestration. Applying the HTTP3/QUIC standard here will be a strong indicator for the level of difficulty extending the application of QUIC away from Google's proprietary control and onto the open market. Kubernetes setup involves the application of one of existing software distributions. Either setting up Kubernetes manually using the open source software, or using a prepackaged configuration for testing, such as minikube or k3s.

Once the Kubernetes cluster is set up, and a master node is running, preparation to add further worker nodes or master nodes can commence. In our case, the implementation is limited to a single local machine. This basic setup is ideal, where the only requirement

for our application of the standard is solely the proper running conditions of the initial environment.

8.5 Ingress Server

As discussed in the background research, kubernetes' native ingress functionality does not support QUIC and HTTP3 connections. As such, for a kubernetes cluster to follow our design requirements, requires the use of third party ingress controllers, such as TraefikProxy. By design the proxy service must support TLS certificates for QUIC controlled socket layer encryption and authentication, even if the certificates will be ignored within our use case. The proxy service must be able to handle these HTTP3 connections in a sufficient manner that is deemed to follow QUIC's RFC 9000 standards. Whether the connection is handled properly is a subject of testing during the implementation and results sections.

8.6 Backend Server

The back-end server is the simplest component of our connection pipeline. Depending on whether we use TLS passthrough or TLS termination will determine the type of backend server needed.If TLS termination is used, and QUIC packets are decrypted at the proxy level, the final backend server does not need to implement any QUIC server handling features, and can communicate between the Ingress proxy using TCP with older HTTP versions. Therefore some simple javascipt, nodejs or any other software which can serve simple html pages would work. If TLS passthrough is enabled, the backend server must contain an implementation of the QUIC protocol, following the RFC standard. Using a tested, widespread implementation would limit any potential protocol errors within the scope of the Ingress Controller specifically, easing the ability to test for misbehaviour within our system. The AIOQUIC python implementation provides a simple server, which serves basic test html files.

8.7 Summary

We have established the basic building blocks needed for minimal setup HTTP3 testing for a Kubernetes Ingress setup, and can now progress with implementation

Chapter 9

Implementation

These sections are a step-by-step progression of applying the QUIC standard within the Kubernetes environment, including unforseen issues which change the scope of the environment. It begins from initial cluster setup, small-scale testing, ingress controller configuration for HTTP3, and finally highlights our decision to move away from a local cluster to a cloud provided cluster

9.1 Kubernetes Cluster Setup

We begin by setting up our minikube based, Kubernetes local cluster. Once installing our linux supported binary and installing it with a package manager, we can setup our single-node Kubernetes cluster with a simple "minikube start" at the terminal

```
pascal@pascal-Precision-5540:~/.config/k9s$ minikube start
😄  minikube v1.28.0 on Ubuntu 22.04
✨  Using the docker driver based on existing profile
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🏃  Updating the running docker "minikube" container ...
🎉  minikube 1.33.0 is available! Download it: https://github.com/kubernetes/minikube/releases/tag/v1.33.0
💡  To disable this notice, run: 'minikube config set WantUpdateNotification false'
```

Figure 9.1: The figure shows the output of starting a minikube cluster. It uses the docker driver to create cluster nodes, pulls Kubernetes images required for control plane and node components to run within the cluster, and provides further versioning or update information

Minikube will create the cluster, and update your local Kubernetes configuration file to allow you to make API requests and change cluster state. Once the cluster is setup, we are free to deploy custom Kubernetes components defined in Kubernetes yaml.

9.2 Host Protcol Testing

We begin by setting up a working AIOQUIC client and server to test conditions on our host machine, and view the behaviour of the default example requests provided by the AIOQUIC implementation. We do this using Wireshark packet captures.
Appendix A.4 shows the basic commands to start the AIOQUIC server on our localhost. Appendix A.3 shows the basic commands to initiate a QUIC connection through a HTTP3 request.

```
pascal@pascal-Precision-5540:~/Desktop/Dissertation_Repos/Dissertation-Files$ python3 aioquic/examples/http3_client.py --ca-certs aioquic/test
s/pycacert.pem https://localhost:4433/ --quic-log ./qlogs/client --secrets-log tls-handshake-secrets/local-ssl-keylog-secrets.log -k
2024-04-27 00:08:59,612 INFO quic [7876faa1ce4012b2] ALPN negotiated protocol h3
2024-04-27 00:08:59,613 INFO client New session ticket received
2024-04-27 00:08:59,618 INFO client Response received for GET / : 1276 bytes in 0.0 s (2.275 Mbps)
2024-04-27 00:08:59,618 INFO client Push received for GET /style.css : 0 bytes
2024-04-27 00:08:59,618 INFO quic [7876faa1ce4012b2] Connection close sent (code 0x100, reason )
```

Figure 9.2: The figure shows the output of locally testing two AIOQUIC instances, a client and server respectively. The output is the reply to its HTTP get request, recorded in the terminal

Now that we have established the protocol works correctly, we can begin applying it within Kubernetes.

9.3 Image Building

Since Kubernetes is a container orchestration tool, we must follow the standard container building setup. Therefore our first step is to create images which contain the AIOQUIC dependencies and installation requirements. Appendix 5.1 shows the basic image defined in a dockerfile format, comprised of aioquic dependencies, simple scripts and additional network related tools. A few tools were installed additionally to ensure we could test additional cases, or print successful output logs, netstat to show what ports our container is listening on, screen in case we need to create threads. netcat to test UDP communication is working. dnsutils to use the nslookup command, allowing us to test if Kubernetes DNS names are working properly, for service name resolution.

The image was then built and pushed locally to minikube using:

```
minikube image built -t [name] .
```

9.4 Kubernetes Yaml Construction

Now that the image is built and loaded into minikubes local context, we can begin creating a yaml file to create a server and client pod within our cluster.

Appendix A.6 shows the completed yaml for a http-3 client pod, the server is identical, excluding naming convention.

Appendix A.7 shows the setup.sh script, which is run on pod creation, ensuring the pod runs the approriate client or server functionality. A big issue encountered during the setup process, is Kubernetes would automatically restart the pods immediately after creation. Kubernetes classifies runtime by ensuring a process is currently running without failure. While the server is an AIOQUIC server which listens forever, the client pod runs a single command and stops. This would prevent me from accessing the pods and running further tests, such as packet captures. To allow meaningful manual testing, I decided to include an infite loop in the setup.sh script which prints the status of the pod.

```
while true ; do
  sleep 600
  echo -e "Currently running server at $(date) is: \n\n$(ps -ef | grep "[
done
```

Listing 9.1: Shows the infinite loop required to ensure Kubernetes does not assume the pod is failing

Extending onto this, in Kubernetes, livenessProbes can be defined to customize and categorize whether your pod is working correctly. Here I listed the active processes on the container, and ensure a python one is running. If a dedicated thread was not created, which is what my setup script ensures, the container is restarted.

```
livenessProbe:
    exec:
         command: ["/bin/bash","-c","ps -ef | grep [p]ython"]
    initialDelaySeconds: 120
    periodSeconds: 30
```

Listing 9.2: Shows the custom liveness check which ensures a python process is running on the pod

9.5 Packet Capturing

Wireshark is a graphical interface, and cannot be easily run in containerized minimal environments. I decided to opt for command-line capturing tools, such as Tcpdump. On the AIOQUIC server pod, I ran a Tcpdump command, targeting the UDP protocol and

the port 4433 which the server was actively listening for QUIC connections. Since QUIC frames are encapsulated in UDP datagrams, I could receive all QUIC traffic by filtering for the UDP protocol.

```
sudo tcpdump udp port 4433 -X -w /opt/test.pcap -vv
```

Listing 9.3: Command to listen for udp packets, "-X" shows ascii output of hex values, "-w" writes all listened packets to a pcap format file.

A HTTP3 get request was then initiated from the AIOQUIC client pod. Once the handshake was established, and data finalized, I interrupted the Tcpdump to receive an output pcap file, containing all exchanged packets. Concurrently, the AIOQUIC server had produced TLS session keys, which are output into a file. I combined the pcap alongside the session keys within Wireshark to decrypt the QUIC communication. Alongside this, a qlog capture was output from both the client and server pods, for later QVIS inspection. From the Wireshark and AIOQUIC client outputs we can conclude that QUIC communication was successfully established between pods inside the cluster.

9.6 TraefikProxy Testing

TraefikProxy is our ingress controller of choice, it comes with a preconfigured implementation in the local cluster service k3d, and in minikube, our choice, we install it manually to have an easily configurable setup. TraefikProxy was chosen because it offered some unique testing considerations. TraefikProxy's use of the quic-go QUIC implementation, allows us to compare discrepancies between AIOQUIC to AIOQUIC local testing, versus AIOQUIC client to quic-go communication. Notable observations could potentially mirror the findings from [34], showcasing the findings when exploring open source technologies.

Traefikproxy was installed using a helm installing, by first downloading the chart from their git repository, and then configuring a values file for our specific implementation. My first step was to use the default values and attempt a basic configuration, to ensure everything went smoothly.

To achieve this I prepared a basic deployment and service for nginx pods, referenced in Appendix A.8, which serve a basic test html file for testing. If the HTML is visible in a browser, then our ingress service is working. The deployment was setup and the Traefikproxy helm chart was installed using default values with the following command:

```
helm install traefik traefik/traefik
```

Listing 9.4: Shows the helm installation command, which creates a new helm release within the cluster called traefik, and installs the locally referenced "traefik/traefik" helm chart

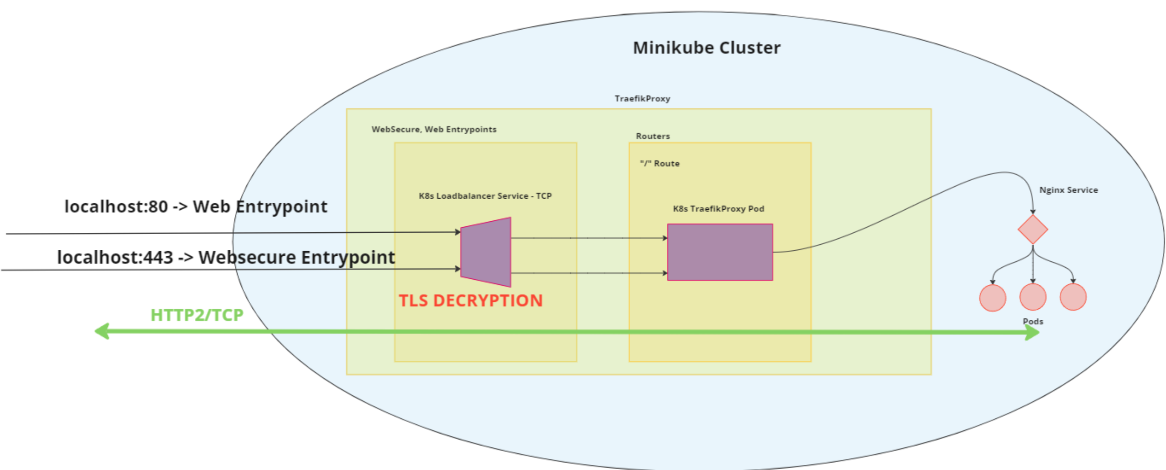


Figure 9.3: The figure shows the architecture diagram for the port-forwarding to the host machine. It defines the traefikproxy endpoints and routers as logical areas over kubernetes services and pods. Entry-points are mapped from the loadbalancer to the Traefik pod, where its decrypted if needed, then forwarded to the nginx back-end service

This created the default entrypoints for traefik, the traefik dashboard, and exposed ports to access them both on the service heading the pod deployment. Routes from the entrypoints to the backend nginx deployment must now be created. As described in the background research, traefikproxy can pickup and create routers either from their own custom defined Kubernetes resources, IngressRoutes, or can pickup native Kubernetes Ingress. Appendix A.9 covers the ingress setup for the nginx pod, to be automatically picked up as a router by traefikproxy. Once the routers, entrypoints and deployment were set up, I portforwaded the loadbalancing service sitting in front of the traefikproxy pod to my host machine. My chrome browser was able to successfully load the nginx html when accessing from the default web entrypoint.However, when accessing from the web-secure entrypoint on port 443, which handles HTTPS communication, it would always fail.

This was immediately a huge issue, as QUIC communication requires encryption, and if I could not get basic TLS working for HTTP2 and older, I could not progress. I made an assumption that perhaps there was an issue with the TLS authentication, and

searched through the TraefikProxy helm chart to find out how the certificates were handled. Through all the values file documentation, I could not find meaningful steps to apply custom TLS certifcates. I created a self-signed certificate for the localhost domain while inside the traefikproxy pod, and added the certificate to be trusted by my host machine. The traefikproxy official documentation had limited resources for TLS configuration for the helm chart. The documentation does not paint a full picture of how to configure TLS properly, with the examples provided not specifying if it defines Kubernetes, helm, or just the normal traefik installation on a host machine. I tried setting flags within the Kubernetes pods, adding key-value pairs to the yaml file, creating Kubernetes secret files, and nothing would work to allow me to provide my custom TLS certificates

Finally I found a ticket opened on the traefik helm chart github, which proposed a solution to use one of Traefik's TLSStore custom Kubernetes resources, and naming their custom certificate "default". It seems a lot of developers were having issues with figuring out how to properly set up certificates for Traefik. However, TLSStores would not work with Traefik automatically creating routers from Kubernetes' own ingress resource. I was forced to create an IngressRoute custom traefik resource. Appendix A.10 shows a Kubernetes secret, containing a tls public and private key b64 encoded. Appendix A.11 shows the TLS store yaml definition, which references the Kubernetes secret file and the keys stored. Appendix A.12 shows the Ingress Route yaml definition, with the route provided for the nginx-service

However, while my browser successfully showed the TLS certificate provided was in fact my own, the webpage still failed to load when accessing from the websecure entrypoint. Finally, I realised I was accessing the webpage through "localhost:443", which is the port for the websecure entrypoint, however, I never explicity defined "https://localhost:443" in my web url. Unfortunately, this was the issue all along, which created wasted time setting up self-signed TLS certificates.

9.7 HTTP3 with TraefikProxy

The documentation within both the helm chart and the official webpage for traefikproxy mentioned the simplicity of extending traefikproxy to use http3. These were the suggested changes to the values file.

```
ports:
  websecure:
    http3:
```

```
    enabled: true
```

Listing 9.5: Shows a helm values file configuration for enabling the http3 feature in traefikproxy

The actual changes to yaml files for Kubernetes due to these helm values changes, are the addition of an extra port in the loadbalancer service in front of the traefikproxy service, and a mapping to the newly exposed port on the traefik pod

```
ports:
- port: 80
  name: "web"
  targetPort: web
  protocol: TCP
- port: 443
  name: "websecure"
  targetPort: websecure
  protocol: TCP
- port: 443
  name: "websecure-http3"
  targetPort: websecure-http3
  protocol: UDP
```

Listing 9.6: Shows the port mapping on the Kubernetes loadbalancer service to the traefikproxy pod. This is the result of applying the http3 configuration for traefikproxy

It adds a second condition for the handling of UDP services on the same port as the websecure entrypoint. Furthermore, it adds two flags to the container's environment variables "experimental.http3=true" and "entrypoints.websecure.http3".

Unfortunately, I still could not get the nginx web page to be returned. So I decided to do further research on the matter. Eventually I found two major references regarding minikube and Kubernetes. Firstly, Kubernetes port-forwarding to access the internal pods only supports TCP based communications. This is still an open topic within the Kubernetes open source implementation [**?**]. Secondly, Minikube's other configuration options, using NodePort services or ingress, both require the use of minikube tunnel to forward connections to the host machine. Minikube tunnel does not currently support UDP forwarding either.

I was presented with two options. The first is, k3d and kind also have local testing functionality, with references to UDP forwarding. I can spend time testing k3d and kind, hoping that other limitations would not occur during the testing process. The second would be to commit to porting my applications to a cloud provider, and provisioning an external load balancer from the cloud provider using a static public IP address.

9.8 Cloud Environment

Based on prior research, I have decided to use the GKE environment as AKS does not offer much more ahead of GKE. The pricing model covers the time scope of the dissertation period, and the slight complexity in documentation is negligible when we consider only simple cluster setups are required in any case. The potential benefits of using a cloud environment designed for Kubernetes may prevent issues in implementation. We will now begin with initial conditions for setting up our google services and cluster creation, followed by continuing our http3 testing

9.8.1 Account Creation and CLI setup

The first step of the process is to create or use a google account. I decided to use my college provided gmail account to progress in this step. I applied for the free trial, which provides approximately three-hundred United States Dollars of credits for use, lasting 2 months. I completed this process through my web browser.

The google cloud console allows you run google cloud commands and communicate with you resources residing in one of their datacentres. One issue with this, is using it through the web interface makes it difficult to apply and use local storage devices on my personal machine. I opted to begin the processes of setting up the google cloud, gcloud interface on my local machine.

Following the installation and setups steps from the documentation provided was quick. Connecting to my google account, and authenticating through the browser, creating a project in the EU-west region for low latency, and then finally beginning the process of setting up a GKE cluster.

As discussed previously, google offers two types of cluster archetype, with some configurations for each. Private Clusters with no outside access to the control plane, and

private clusters with authorized outside access to the control plane.

9.8.2 Cloud TraefikProxy Testing

Once again we attempt an intial setup with HTTP3 not enabled, allowing us to debug any other potential issues before we implement our HTTP3 solution. I installed the traefikproxy helm chart with default values, then applied the referenced files in Appendix A.6, and Appendix A.9

Google cloud automatically detected the use of a LoadBalancer service type, which is defined in the helm chart image, and automatically provisioned an external IP address, with the relevant port mappings for the web and websecure entrypoints.

Upon querying the IP address within my chrome browser, I was successfully able to complete a HTTP and HTTPs request.

9.8.3 Cloud HTTP3 Testing

```
pascal@pascal-Precision-5540:~/Desktop/Dissertation_Repos/Dissertation-Files$ kubectl get svc
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                       AGE
nginx-service   ClusterIP      10.0.126.6     <none>        80/TCP                        53d
traefik         LoadBalancer   10.0.123.84    <pending>     80:31990/TCP,443:32656/TCP    3s
```

Figure 9.4: Figure shows the result of a 'kubectl get svc' command, which shows the state of service objects, including their cluster-IP and external-IP addresses.

I then proceeded to enable HTTP3 features as in my previous minikube testing. However I encountered an issue, the google cloud service would not provide an external IP address for my TraefikProxy loadbalancer service.

```
service/traefik            Ensuring load balancer
service/traefik            LoadBalancers with multiple protocols are not supported.
deployment/traefik         Scaled up replica set traefik-fd74d578d to 1
```

Figure 9.5: Figure shows the error output when creating a Loadbalancer of multiple port protocol types. The error prevents an external IP address from being assigned to the load balance service

The logs produced by my traefikproxy pod were indicating that something google cloud did not support shared ports for multiple protocols. I began numerous methods of manually modifying the Kubernetes yaml files to change this

Separate Protocol Ports

My initial test was to change the ports in which the http3 services are forwarded to the traefikpod. This could be done through the helm chart values file by setting the "http3-advertised-listeners" value to a different port.

```
- port: 443
  name: "websecure"
  targetPort: websecure
  protocol: TCP
- port: 4443
  name: "websecure-http3"
  targetPort: websecure-http3
  protocol: UDP
```

Listing 9.7: Shows choosing a separate port for each protocol, an attempt to circumvent mixed protocol support issues

This produced yaml output shown above in the Kubernetes loadbalancer service, changes the port that is advertised on the external IP address, but on the traefik pod itself, both entrypoints share the same port. This is because traefik must route HTTP3 requests through its TLS secure entrypoint. This solution present a potential fix on the external load balancer provided by the cloud provider.

Unfortunately I encountered the same error, that external load balancers do not support multiple protocols regardless of port allocation.

UDP For All Entrypoints

```
        - port: 443
          name: "websecure"
          targetPort: websecure
          protocol: UDP
        - port: 443
          name: "websecure-http3"
          targetPort: websecure-http3
          protocol: UDP
```

Listing 9.8: Shows the use of a single protocol for the loadbalancers traefikproxy entrypoint mappings, an attempt to circumvent the unsupported mixed protocol issue

I then attempted to configure the websecure entrypoint to only allow UDP traffic. I assumed that due to my AIOQUIC forcing http3 communication without upgrading, I did not need TCP based communication on the websecure entrypoint for traefik. While the google cloud service successfully provided me with an external IPfor testing, unfortunately I could not get a single HTTP3 reply returned with this configuration. Something within the traefikproxy configuration did not support this, and the documentation for the software offered no guidelines for this case

Load Balancer per Protocol

I found a reference to a Kubernetes Github ticket opened regarding allowing mixed protocol loadbalancer services. It stated that as of Kubernetes minor version 1.24, mixed protocol services are enabled by default. Despite these changelogs and the assurance of other commenters within the final few comments on the ticket, it would not work. The error itself regarding mixed protocol load balancers not being supported, seemed to be an issue in google's external load balancers.

```
## -- Enable HTTP/3 on the entrypoint
## Enabling it will also enable http3 experimental feature
## https://doc.traefik.io/traefik/routing/entrypoints/#http3
## There are known limitations when trying to listen on same ports for
## TCP & UDP (Http3). There is a workaround in this chart using dual Service.
## https://github.com/kubernetes/kubernetes/issues/47249#issuecomment-587960741
http3:
  enabled: false
```

Figure 9.6: Figure shows a comment present in the traefikProxy values helm file, offering a potential solution for the multi-protocol load balancer issue.

Within traefikproxy's documentation regarding http3, it also references the previous issue of Kubernetes not supporting mixed protocol load balancing. However, It provides a suggestion to use two separate services for a single traefik pod, to handle the two transport protocols. Appendix A.13 shows an attempt to create two separate load balancers, one to handle tcp entrypoints, and one for the http3 websecure entrypoint

While our attempt was successful at creating two separate load balancers, and each of which was successful at being assigned an external IP address, HTTP3 communication from our AIOQUIC client would fail when accessing the IP address and port for the HTTP3 websecure entrypoint. My only conclusion is there is some relation between the default websecure entrypoint and the the websecure entrypoint. It would account

for the previous two solutions of changing the default entrypoint to UDP not allowing communications directly with the HTTP3 entrypoint. My first thought is, it is in reference to ALPN negotiation. For most Implementations of HTTP3, a client initiates a regular HTTP2 or HTTP1.1 handshake with the server, and only once the server informs the client it can support HTTP3 requests, through the alternative service header in the HTTP reply, can HTTP3 communication occur.

```
configuration = QuicConfiguration(
        is_client=True,
        alpn_protocols=H0_ALPN if args.legacy_http else H3_ALPN,
        congestion_control_algorithm=args.congestion_control_algorithm,
        max_datagram_size=args.max_datagram_size,
    )
```

Listing 9.9: Shows a snippet from the AIOQUIC github source repository, showcasing the condition of choosing supported application layer protocols for ALPN negotiation

AIOQUIC itself allows either HTTP version 0.9 or HTTP3 requests to be defined in its supported ALPN definitions, but not both at the same time. Since upgrades are not supported, this may indicate to me that I cannot use AIOQUIC to force HTTP3 only communication with traefikproxy. However, I continued to research further

Static IP - Multi Service

I found a refernce on the google cloud documentation which references splitting up load balancers to handle separate transport protocols. While the reference was for general private cloud VMs and container instances using the google compute engine, I tried to apply the same notions to the GKE cluster. The documentation references creating a single static IP address, and assigning it to two individual load balancers. These load balancers can then be configured through the Google CLI to handle protocols separately. In GKE, since external load balancers are automatically created to follow Kubernetes loadbalancer service yaml, I experimented with this firstly.

```
gcloud compute addresses create --region="eu-west"
```

Listing 9.10: Command to reserve a single static public IP address within the eu-west region

My first step was to reserve an external IP address within my google compute region, rather than let GKE automatically assign one for each load balancer. Appendix A.14 shows my attempt to configure two distinct loadbalancer with the same external IP address. GKE allows you to assign a "LoadBalancerIP" key to the spec section within the Kubernetes yaml file. The GKE engine will assign your reserved static IP address to it once the service is started up within the cluster. Now that we have two load balancer services running on the same external IP address, I tested the AIOQUIC client at the available IP.

```
pascal@pascal-Precision-5540:~/Desktop/Dissertation_Repos/Dissertation-Files$ python3 aioquic/examples/http3_client.py --ca-certs aioquic/test
s/pycacert.pem https://34.89.71.52/ --quic-log ./qlogs/client --secrets-log tls-handshake-secrets/local-ssl-keylog-secrets.log -k
2024-04-21 01:36:04,445 INFO quic [b0dbee700bcf79a1] ALPN negotiated protocol h3
2024-04-21 01:36:04,466 INFO client New session ticket received
2024-04-21 01:36:04,469 INFO client Response received for GET / : 612 bytes in 0.0 s (0.224 Mbps)
2024-04-21 01:36:04,470 INFO quic [b0dbee700bcf79a1] Connection close sent (code 0x100, reason )
```

Figure 9.7: Figure shows the output of an AIOQUIC client, receiving a HTTP3 return request from the remote cloud server.

Finally our first successful HTTP3 reply message has been received. The process to achieve this, in what would be considered a simple ten minute setup for older HTTP versions based on TCP, took extensive research into the QUIC protocol, QUIC implementations, TLS security, Kubernetes networking and finally Cloud offerings.

9.9 Summary

This section has taken us through the implementation considerations for QUIC within Kubernetes, and identified how despite the majority of software used is globally deployed and in some cases considered the main tool or software in the area, there were extreme issues in QUIC testing and deployment. Finally a working HTTP3 testing setup was reached after arduous effort.

Chapter 10

Evaluation

10.1 Results

The results section is an extra assertion of the correctness of our solution. Throughout the implementation section we followed the chronological progression of the project towards a working solution. AIOQUIC provides a minimal interface to print successful packet sending and receiving on a command-line interface. Therefore, this section focuses on the environmental differences using the HTTP3 and QUIC protocols, by identifying the individual packets, asserting whether differences offer any notable observations. The comparisons using QVIS and Wireshark allow us to view overall communication with QVIS, and more detailed packet formation through Wireshark's QUIC dissection. A notable observation is QVIS displays individual QUIC packets, while Wireshark shows UDP datagrams, which may contain multiple different packets

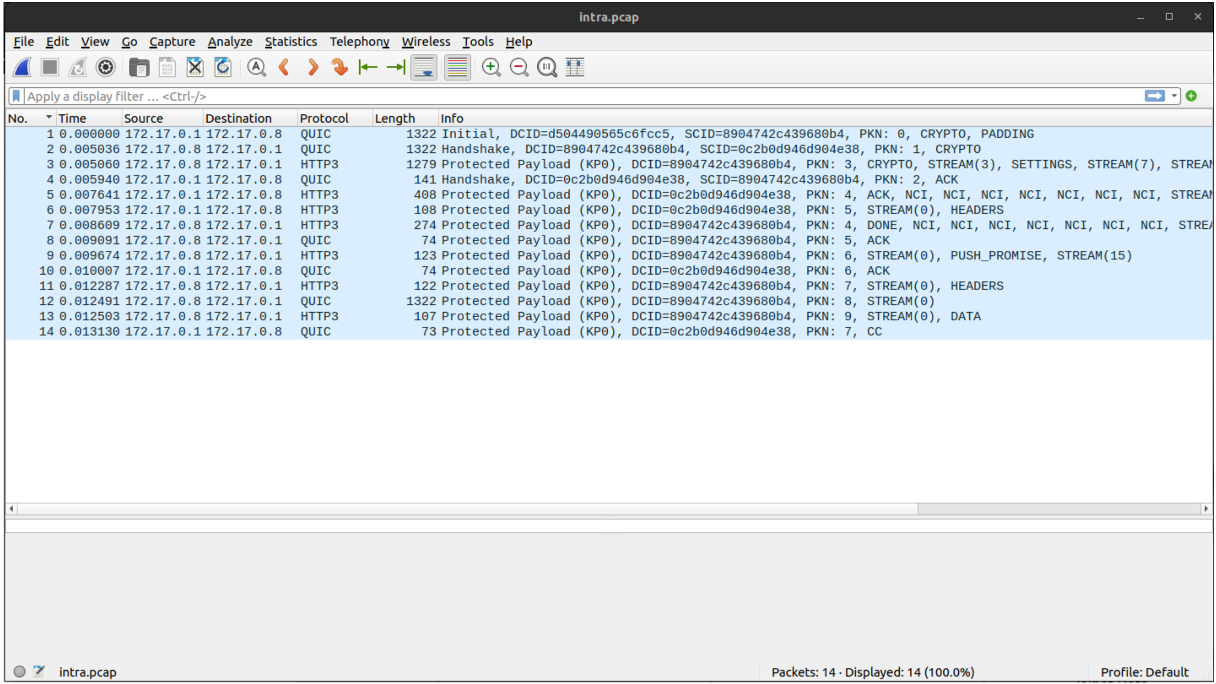
10.1.1 Local AIOQUIC

Our local AIOQUIC testing results were already showcased in figure 6.3. They were achieved by running an AIOQUIC client and server on the host machine, with a Tcpdump packet capture software listening on the server's listen port. The step by step packet analysis is identical to the previous example. We have established AIOQUIC in a neutral control environment like the host machine, is the baseline for our test cases. The behaviour of the packets in the examples is how client and server packets should loosely behave

10.1.2 Intra-Cluster Testing

The intra-kubernetes test results were done within the cluster, on the AIOQUIC http3-client and http3-server pods. The image defined in Appendix 5.1 was packaged with

the Tcpdump software. Similar to the local testing, the pcap was generated from the Tcpdump, ported into Wireshark for decryption with the TLS session keys output from AIOQUIC, and the following results were obtained.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.17.0.1	172.17.0.8	QUIC	1322	Initial, DCID=d504490565c6fcc5, SCID=8904742c439680b4, PKN: 0, CRYPTO, PADDING
2	0.005036	172.17.0.8	172.17.0.1	QUIC	1322	Handshake, DCID=8904742c439680b4, SCID=0c2b0d946d904e38, PKN: 1, CRYPTO
3	0.005060	172.17.0.8	172.17.0.1	HTTP3	1279	Protected Payload (KP0), DCID=8904742c439680b4, PKN: 3, CRYPTO, STREAM(3), SETTINGS, STREAM(7), STREA
4	0.005940	172.17.0.1	172.17.0.8	QUIC	141	Handshake, DCID=0c2b0d946d904e38, SCID=8904742c439680b4, PKN: 2, ACK
5	0.007641	172.17.0.1	172.17.0.8	HTTP3	408	Protected Payload (KP0), DCID=0c2b0d946d904e38, PKN: 4, ACK, NCI, NCI, NCI, NCI, NCI, NCI, NCI, STREA
6	0.007953	172.17.0.1	172.17.0.8	HTTP3	108	Protected Payload (KP0), DCID=0c2b0d946d904e38, PKN: 5, STREAM(0), HEADERS
7	0.008609	172.17.0.8	172.17.0.1	HTTP3	274	Protected Payload (KP0), DCID=8904742c439680b4, PKN: 4, DONE, NCI, NCI, NCI, NCI, NCI, NCI, NCI, STRE
8	0.009091	172.17.0.8	172.17.0.1	QUIC	74	Protected Payload (KP0), DCID=8904742c439680b4, PKN: 5, ACK
9	0.009674	172.17.0.8	172.17.0.1	HTTP3	123	Protected Payload (KP0), DCID=8904742c439680b4, PKN: 6, STREAM(0), PUSH_PROMISE, STREAM(15)
10	0.010007	172.17.0.1	172.17.0.8	QUIC	74	Protected Payload (KP0), DCID=0c2b0d946d904e38, PKN: 6, ACK
11	0.012287	172.17.0.8	172.17.0.1	HTTP3	122	Protected Payload (KP0), DCID=8904742c439680b4, PKN: 7, STREAM(0), HEADERS
12	0.012491	172.17.0.8	172.17.0.1	QUIC	1322	Protected Payload (KP0), DCID=8904742c439680b4, PKN: 8, STREAM(0)
13	0.012503	172.17.0.8	172.17.0.1	HTTP3	107	Protected Payload (KP0), DCID=8904742c439680b4, PKN: 9, STREAM(0), DATA
14	0.013130	172.17.0.1	172.17.0.8	QUIC	73	Protected Payload (KP0), DCID=0c2b0d946d904e38, PKN: 7, CC

Figure 10.1: The figure shows the pcap post-tls-decryption. Unlike the local test it only contains 14 packets, due to clever packet stuffing for TLS handshake information, within a single UDP datagram

While the total packet number has changed from 18 to 14, it must be reminded that within a single UDP datagram multiple QUIC frames. This can cause cases where a single datagram at a QUIC handshake step may have different contents, but ultimately the same processes happen by the end. One noticeable difference in UDP datagram packing between the local and this intra capture is the local session used 3 datagrams to do the server hello, server certificate and set up initial HTTP3 and QUIC streams.

In the intra network example, it did so in two packets. It packed a fragment of the certificate into the initial UDP that contained the server hello, then instead of creating a separate UDP datagram dedicated to certificate information like in the local capture, it instead put the remaining certificate information in UDP datagram number 3, where the certificate fragments were reassembled, and the initial HTTP3 and QUIC streams are created in the same UDP datagram. Overall the process was the same, but while the local communication took 25ms, the intra-kubernetes session took 13ms. The opportunistic packing of QUIC frames into a single datagram allowed a faster overall handshake. In

terms of behaviour, it was perfectly in line with the baseline local example, as we would expect. Next we will test host to cluster communication.

10.1.3 Cloud HTTP3

The cloud environment test results presented some interesting findings. Firstly, the use of TraefikProxy's implementation of quic-go, in contrast to the client implementation using AIOQUIC, caused issues in Wireshark when attempting to decrypt the packets with the appropriate TLS session keys.

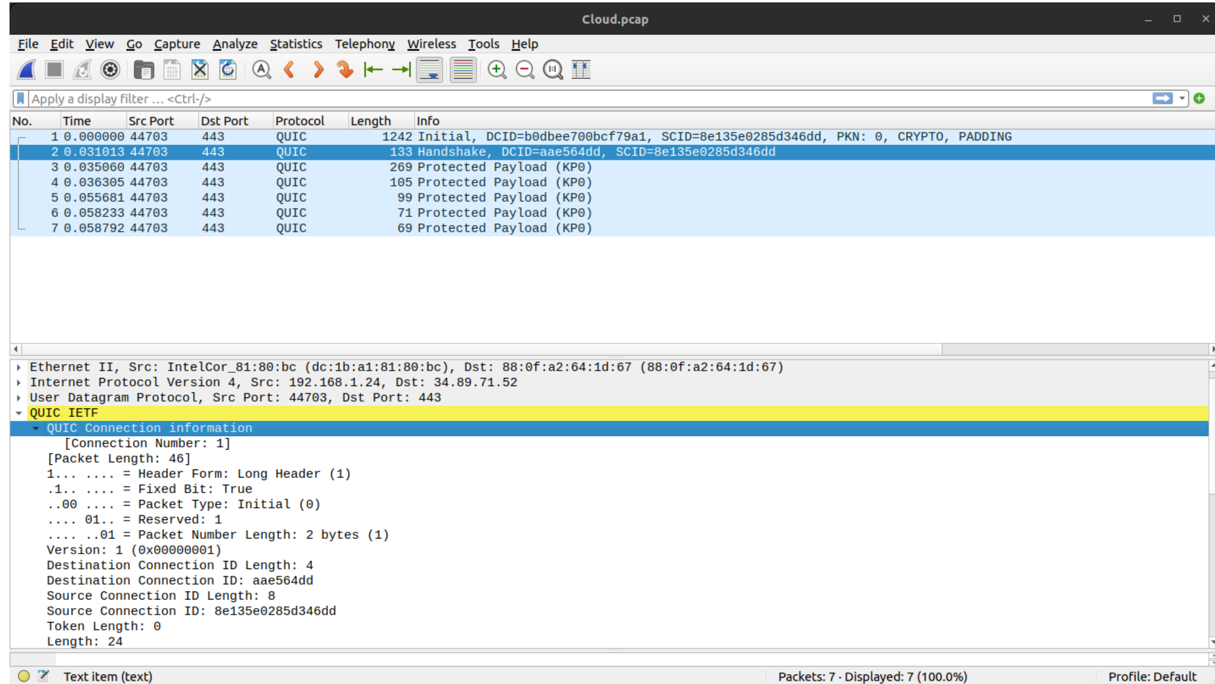


Figure 10.2: The figure shows the 7 packet QUIC session to the cloud server, running quic-go under traefikproxy. The differences in expected packet format caused Wireshark to be unable to decrypt the pcap file

Previously we noticed a similar decryption error failure when researching QVIS packet visualization, and converting pcap files to qlog trace files using the pcap2qlog tool. There, disagreements in QUIC packet layout for json formats between Wireshark and the pcap2qlog tool caused decryption errors in the final qlog output. It can be assumed that implementation differences between AIOQUIC and Quic-go, have produced a non-standard pcap file output, which Wireshark cannot successfully decrypt. Ironically however, our qlog output from AIOQUIC, whose packet traces are decrypted internally within AIOQUIC using the same TLS session keys, succeeded in producing a decrypted output, and could be visualized within QVIS

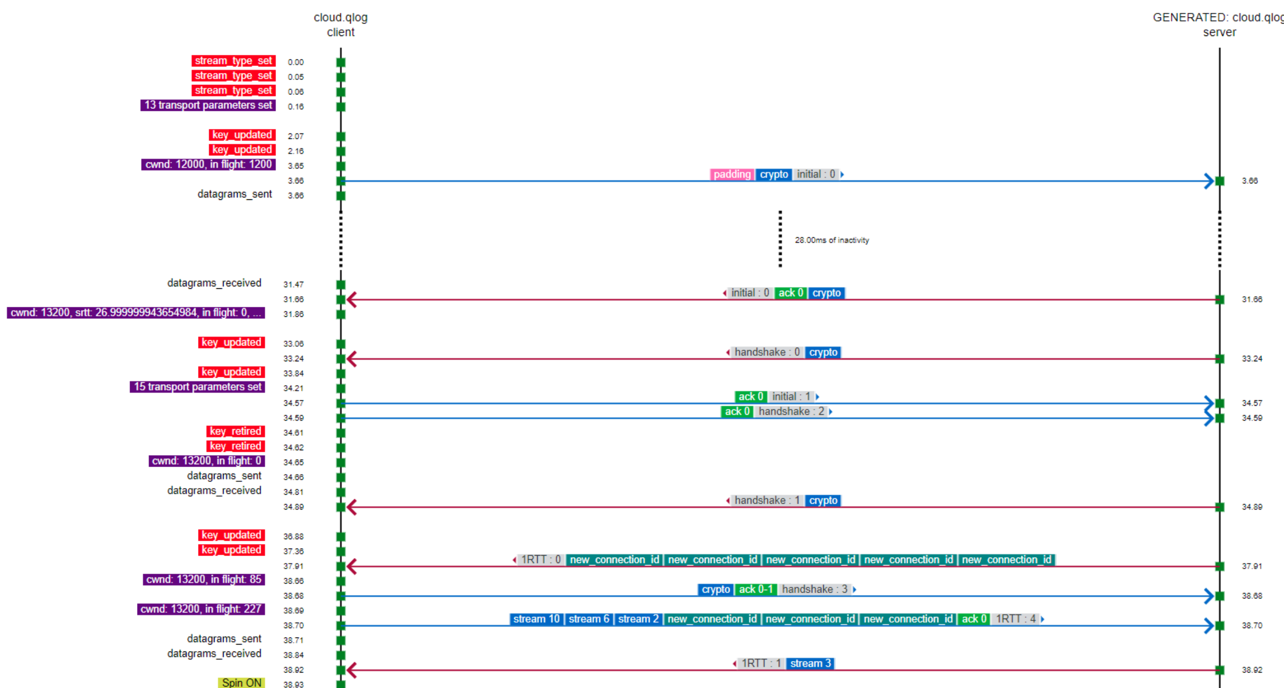


Figure 10.3: The figure shows the first half of the Qvis trace for client to cloud server QUIC communication, it covers steps between the client handshake, new connection setups, client control, encoding and data streams, the client HTTP3 get request, and finally an encoding stream message on stream 3 from the server

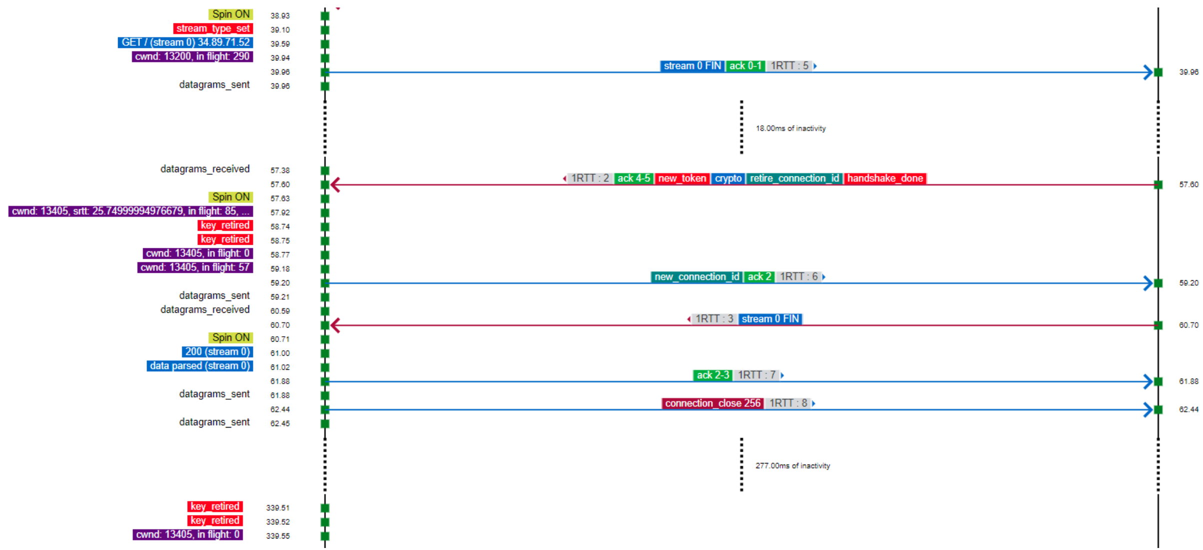


Figure 10.4: The figure shows the second half of the Qvis trace. It shows the new connection token for connection resumption being passed to the client, the HTTP3 response message, and the connection close

Compared to previous examples with both an AIOQUIC client and server, it was

impossible to return the qlog trace output from the server specifically, since configuring quic-go from traefik's internal helm chart was impossible. Therefore our QVIS output contains information as viewed from the client only. Inspecting QVIS output, we notice much longer round-trip time delays between packets, which is to be expected when communicating with an internet endpoint. The total communication time was 62ms, three times as long as the local QUIC tests and approximately 5 times longer than the intra-cluster tests.

The AIOQUIC client creates three unidirectional QUIC streams for HTTP3 control, encoding and decoding, or stream id 2, 6, and 10 respectively. Surprisingly, we only see a control stream defined by the quic-go server, stream id 3. This indicates the server has no intention to do header compression using the QPACK format. Another difference is the server sends a "new_token" frame to the client, this is used for 0-RTT session resumption in future connections. The prior examples for QUIC -¿ QUIC communication did not facilitate that.

10.2 Summary

We have showcased how the working configurations reached in our implementation still had noticeable differences between each other. Slight timing differences between host and intra-cluster tests allowed the intra-cluster test to opportunistically encapsulate multiple QUIC CRYPTO frames in a single UDP datagram, shortening the handshake time.

During cloud testing, differences were immediately more notable, with the total round trip time quadrupling and packet dissection issues occurring once heterogeneous QUIC implementations were used to generate the pcap file. Even in a best case testing scenario, with a minimal cluster configuration, discrepancies arose in the testing of the new software.

Chapter 11

Conclusions & Future Work

Whilst the QUIC protocol attempts to tactically enter the inter-networking landscape, it cannot avoid inherent issues with change in such solidified environments. Discrepancies in performance across non-Google production endpoints[34], limited solving of ossification across middleboxes[20] resorting to ALPN negotiation for support failover, all showcase the difficulty in adoption at large-scale levels. When evaluating the usage statistics of the protocol, Google is spearheading its use, and benefits the most out of properly configured, cohesive QUIC communication.

Google's large market share of both global inter-networking and browser usage, combined with years of private protocol testing and iteration, have highlighted just how much effort is needed to change the standard for transport protocols.

Our testing in the open source domain has identified extreme difficulties in the application of the QUIC protocol within both heavily standardised environments like kubernetes, popular loadbalancing and proxy services like TraefikProxy, and within dedicated packet dissection and visualization tools like Wireshark and QVIS. Further interdependence between open source projects, such as Nginx core to OpenResty and Kubernetes' ingress-nginx, showcases how complex software depth only exacerbates the problem.

11.1 Future Work

The observations made in the environmental testing and configuration in this dissertation will hopefully inspire and provide foundational information for any QUIC protocol testers that follow in my suite. Time committed to technical research on both protocols, implementations, cloud providers, and the Kubernetes environment, combined with multiple bumps with failed QUIC application, impeded the amount of testing possible by the end of the dissertation.

Exploring the relationship between different TLS termination modes, and identifying any possible discrepancies between QUIC communication at the multi-level would have opened new avenues for analysis. TLS passthrough would allow an interesting observation of how ingress-controllers can handle QUIC communication when only header information is available, and may bring to light similarities between middlebox routers between end-user internet communication.

Bibliography

[1] aiortc Authors. aiortc/aioquic GitHub Repository. https://github.com/aiortc/aioquic.

[2] K. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681, September 2009.

[3] Amazon Web Services. Amazon EKS Documentation - Create a Cluster. https://docs.aws.amazon.com/eks/latest/userguide/create-cluster.html, 2024.

[4] C. Authors. Containerd. https://containerd.io, 2024. Accessed on 29.04.2024.

[5] T. K. Authors. Kubernetes. https://kubernetes.io, 2024. Accessed on 29.04.2024.

[6] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, May 2015.

[7] V. G. Cerf and R. E. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22, 1974.

[8] Cloudflare. Http/3 usage: One year on. *https://blog.cloudflare.com/http3-usage-one-year-on*, 2021.

[9] P. J. Denning. The science of computing: The ARPANET after twenty years. *American Scientist*, 75, 1987.

[10] etcd Authors. etcd - Distributed reliable key-value store. https://etcd.io/.

[11] W. Foundation. Wireshark. https://www.wireshark.org.

[12] Google Cloud. Google Cloud SDK Documentation - Install. https://cloud.google.com/sdk/docs/install.

[13] D. Inc. Docker. https://www.docker.com/, 2024. Accessed on 29.04.2024.

[14] J. Iyengar, M. Nottingham, and W. Ruellan. QPACK: Header Compression for HTTP/3. RFC 9001, June 2021.

[15] J. Iyengar, M. Nottingham, and W. Tarreau. Hypertext Transfer Protocol Version 3 (HTTP/3). RFC 9000, June 2021.

[16] Kubernetes. Kubernetes Documentation - Running Controllers. `https://kubernetes.io/docs/concepts/architecture/controller/#running-controllers`.

[17] Kubernetes. Kubernetes GitHub Repository. `https://github.com/kubernetes/kubernetes`.

[18] Kubernetes. Kubernetes Documentation - Creating a Cluster with kubeadm. `https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/`, 2024.

[19] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC transport protocol: Design and internet-scale deployment. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 183–196, New York, NY, USA, 2020. Google.

[20] Y. Liu, F. Zhu, H. Wang, Y. Zhang, and B. Zhang. Learning-based adaptive online testing for web services under service-oriented architecture. *IEEE Transactions on Services Computing*, 9(5):709–723, 2016.

[21] R. Marx. Head-of-line blocking in quic and http/3: The details. `https://github.com/rmarx/holblocking-blogpost`, 2024.

[22] H. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, J. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[23] OpenResty Authors. OpenResty Pull Request 920. `https://github.com/openresty/openresty/pull/920`.

[24] OSI Model. Application layer. `https://osi-model.com/application-layer/`.

[25] OSI Model. Transport layer (Layer 4) of OSI Model. *https://osi-model.com/transport-layer/*.

[26] J. Postel. User Datagram Protocol. RFC 768, August 1980.

[27] J. Postel. NCP/TCP Transition Plan. RFC 801, November 1981.

[28] J. Postel. Transmission Control Protocol. RFC 793, September 1981.

[29] quic-go Authors. quic-go GitHub Repository. `https://github.com/quic-go/quic-go`.

[30] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[31] rmarx. GitHub Profile - rmarx. `https://github.com/rmarx`.

[32] L. Schumann, T. V. Doan, T. Shreedhar, R. Mok, and V. Bajpai. Impact of evolving protocols and covid-19 on internet traffic shares. *IEEE Internet Computing*, 25, 2021.

[33] The Tcpdump Group. Tcpdump - A powerful command-line packet analyzer. `https://www.tcpdump.org/`.

[34] A. Yu and T. A. Benson. Dissecting performance of production quic. *ACM Transactions on Internet Technology*, 2024.

Appendix A

Appendix

A.1 Browser Testing

```
chromium-browser \
--enable-experimental-web-platform-features \
--ignore-certificate-errors-spki-list=BSQJ0jkQ7wwhR7KvPZ+DSNk2XTZ/MS6xCb
--origin-to-force-quic-on=34.89.71.52:443 \
https://34.89.71.52:443/
```

Listing A.1: Code snippet shows browser flag configuration for HTTP3 requests. "enable-experimental" turns on QUIC support, "ignore-certificate.." defines the fingerprint of the certificate we are to ignore, and "force-quic" ensures we only use a HTTP3 connection

```
    openssl x509 -in tests/ssl_cert.pem -pubkey -noout | \
    openssl pkey -pubin -outform der | \
    openssl dgst -sha256 -binary | \
    openssl enc -base64
```

Listing A.2: Code snippet shows how to generate a fingerprint for the TLS certificate, to allow browsers to ignore authentication for it. It generates a base64 encoded output of the public key associated with the server's digital certificate. This fingerprint can be used to explicitly ignore authentication of the chrome browser

A.2 Local Testing

```
python3 aioquic/examples/http3_client.py --ca-certs aioquic/tests/pycacer
```

Listing A.3: The following code shows the required flags to run a HTTP3 QUIC client with AIOQUIC. –ca-cert flag allows the python client to verify the server's certificate. It sends a get request to localhost:4433. –quic-log specifies the client outputted qlog trace of all the packets between itself and the server. -k allows the client to ignore certificate authentication, to simplify TLS certificate handling

```
python3 aioquic/examples/http3_server.py --certificate aioquic/tests/ssl_
```

Listing A.4: The following code shows the required flags to run a HTTP3 QUIC server with AIOQUIC. –secrets-log outputs the session keys between the client and server for packet decryption. –quic-log outputs the server qlog trace for packet between itself and the client

A.3 Image Building

```
FROM postgres
CP start.sh /opt/start.sh
CMD [./opt/start.sh]
```

Listing A.5: Code block above shows the setup of a basic image, using ubuntu as its minimal OS starting image. It then copies the aioquic github repository into the container, then uses the apt package manager to install network testing tools, python, and the pip3 commands for the AIOQUIC implementation to work Page 26 of 31r

A.4 AIOQUIC-Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
        name: http3-client
        namespace: http3-test
        labels:
        app.kubernetes.io/name: http3-client
```

```
spec:
        replicas: 1
        selector:
                matchLabels:
                        app.kubernetes.io/name: http3-client
        template:
                metadata:
                        labels:
                                app.kubernetes.io/name: http3-client
        spec:
                containers:
                - name: http3-client
                        image: raosp/http3-tools
                        imagePullPolicy: Always
                        command: ["/opt/scripts/setup.sh"]
                        args: ["--server"]
                        ports:
                                - containerPort: 4433
                                name: http3-listen
                                protocol: UDP
                        livenessProbe:
                                exec:
                                        command: ["/bin/bash","-c","ps -e
                                initialDelaySeconds: 120
                                periodSeconds: 30
---
apiVersion: v1
kind: Service
metadata:
  name: http3-client
  namespace: http3-test
  labels:
    app.kubernetes.io/name: http3-client
spec:
  selector:
    app.kubernetes.io/name: http3-client
  ports:
```

```
  - protocol: UDP
   port: 4433
    targetPort: http3-listen
```

Listing A.6: The above code block defines both the deployment and service yaml for the http3-client pods. The http3-server pod is identical bar the name field changes. both components are deployed in the http3-test namespace. The pod has a container with the prior defined docker image, and runs the setup.sh script to begin services. It has a livenessProbe which checks if a python process is running on the container to ensure pod health. The service maps the container exposed port to be accessed globally in the cluster

```
HOST_ID="default"

set -euo pipefail

while test $# -gt 0; do
  case "$1" in
  --server*)
      HOST_ID="server"
      shift
      ;;
  --client*)
      HOST_ID="client"
      shift
      ;;
  *)
      break
      ;;
  esac
done


if [[ "$HOST_ID" != "0" ]]; then
  if [[ "$HOST_ID" = "server" ]]; then
    echo "launching http3-tools in $HOST_ID mode"
```

```
    /usr/bin/screen -LdmS python3 /opt/aioquic/examples/http3_client.py \
      --ca-certs /opt/aioquic/tests/pycacert.pem https://http3-server:443
      --secrets-log /opt/ssl-keylog-secrets.log -k


    elif [[ "$HOST_ID" = "client" ]]; then
      echo "launching http3-tools in $HOST_ID mode"


      /usr/bin/screen -LdmS python3 /opt/aioquic/examples/http3_client.py
        --ca-certs /opt/aioquic/tests/pycacert.pem https://http3-server:4
        --secrets-log /opt/ssl-keylog-secrets.log -k

    fi
    while true ; do
      sleep 600
      echo -e \
          "Currently running server at $(date) is:
                                \n\n$(ps -ef | grep "[p]ython" | awk '{print $8, 
    done
else
     echo "No host type specified, use flags --server or --client"
fi
```

Listing A.7: The setup.sh script checks whether the pod is a client or server, defined within the environment variables at deployment. it uses the screen software to spin up a separate thread to run each command. Regardless of type, the pod sleeps and echoes its status every 10 minutes. This ensures kubernetes will not restart the pod

A.5 Traefik Testing

```
apiVersion: apps/v1
        kind: Deployment
        metadata:
          name: nginx-deployment
```

```
    namespace: http3-test
    labels:
        app: nginx
spec:
    replicas: 3
    selector:
        matchLabels:
            app: nginx
    template:
        metadata:
            labels:
                app: nginx
        spec:
            containers:
                - name: nginx
                  image: nginx:1.14.2
                  ports:
                      - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
    name: nginx-service
    namespace: http3-test
spec:
    selector:
        app: nginx
    ports:
        - protocol: TCP
          port: 80
          targetPort: 80
```

Listing A.8: Code describes kubernetes yaml for a simple nginx deployment. mounted onto a service at port 80 to provide the communication to the whole cluster

```
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: ingress-test
  namespace: http3-test
  annotations:
    traefik.ingress.kubernetes.io/router.entrypoints: websecure
    traefik.ingress.kubernetes.io/router.tls: "true"
spec:
  rules:
    - http:
        paths:
          - path: /
            pathType: Exact
            backend:
              service:
                name:  nginx-service
                port:
                  number: 80
```

Listing A.9: Code describes an ingress component, adding extra annotations to configure how traefik treats it. Defining that ingress should only use websecure endpoints. Paths that match the root filepath, will be reverse proxied to the nginx servicesr

```
apiVersion: v1
kind: Secret
metadata:
  name: tls-secret
  namespace: http3-test
type: kubernetes.io/tls
data:
  tls.crt: LS0tLS1C...
  tls.key: LS0tLS1C...
```

Listing A.10: Code above depicts a kubernetes secret, base 64 encoded, container public and private key information for TLS certificates. Secrets can be mounted onto a pod to be used in sensitive authentication steps

```
apiVersion: traefik.containo.us/v1alpha1
```

```
kind: TLSStore
metadata:
        name: default
spec:
        defaultCertificate:
        secretName: tls-secret
```

Listing A.11: Code above describes a TLSStore mounting. This is standard measure for making secrets available to TraefikProxy for TLS authentication, here we reference the previous secret file

```
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: example-ingress
spec:
  entryPoints:
    - websecure
  routes:
    - match: Host(`localhost`)
      kind: Rule
      services:
        - name: nginx-service
          port: 80
```

Listing A.12: The above is identical to the native ingress kubernetes component described prior, however it used Traefik's own CRD object to define it. This is mandatory to use TLSStores

```
apiVersion: v1
kind: Service
metadata:
  name: traefik
  namespace: http3-test
  labels:
    app.kubernetes.io/name: traefik
    app.kubernetes.io/instance: traefik-http3-test
    helm.sh/chart: traefik-26.0.0
    app.kubernetes.io/managed-by: Helm
```

```
  annotations:
spec:
  type: LoadBalancer
  selector:
    app.kubernetes.io/name: traefik
    app.kubernetes.io/instance: traefik-http3-test
  ports:
  - port: 80
    name: "web"
    targetPort: web
    protocol: TCP
  - port: 443
    name: "websecure"
    targetPort: websecure
    protocol: TCP

---

apiVersion: v1
kind: Service
metadata:
  name: traefik-http3
  namespace: http3-test
  labels:
    app.kubernetes.io/name: traefik
    app.kubernetes.io/instance: traefik-http3-test
    helm.sh/chart: traefik-26.0.0
    app.kubernetes.io/managed-by: Helm
  annotations:
spec:
  type: LoadBalancer
  selector:
    app.kubernetes.io/name: traefik
    app.kubernetes.io/instance: traefik-http3-test
  ports:
  - port: 4433
    name: "websecure-http3"
```

```
    targetPort: websecure-http3
    protocol: UDP
```

Listing A.13: The following kubernetes yaml showcases an attempt to configure two separate load balancers for a single pod, adhereing to the recommended solution referenced in the traefik helm chart

```
apiVersion: v1
kind: Service
metadata:
  name: traefik
  namespace: http3-test
  labels:
    app.kubernetes.io/name: traefik
    app.kubernetes.io/instance: traefik-http3-test
    helm.sh/chart: traefik-26.0.0
    app.kubernetes.io/managed-by: Helm
  annotations:
spec:
  type: LoadBalancer
  loadBalancerIP: 34.89.71.52
  selector:
    app.kubernetes.io/name: traefik
    app.kubernetes.io/instance: traefik-http3-test
  ports:
  - port: 80
    name: "web"
    targetPort: web
    protocol: TCP
  - port: 443
    name: "websecure"
    targetPort: websecure
    protocol: TCP

---

apiVersion: v1
kind: Service
```

```yaml
metadata:
  name: traefik-http3
  namespace: http3-test
  labels:
    app.kubernetes.io/name: traefik
    app.kubernetes.io/instance: traefik-http3-test
    helm.sh/chart: traefik-26.0.0
    app.kubernetes.io/managed-by: Helm
  annotations:
spec:
  type: LoadBalancer
  loadBalancerIP: 34.89.71.52
  selector:
    app.kubernetes.io/name: traefik
    app.kubernetes.io/instance: traefik-http3-test
  ports:
  - port: 443
    name: "websecure-http3"
    targetPort: websecure-http3
    protocol: UDP
```

Listing A.14: The following kubernetes yaml showcases two separate load balancers with an external static ip reserved and provisioned for both load balancers