QUIC over high Bandwidth-Delay Product links

Q Misell

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Bachelor of Science
of the
University of Aberdeen.

School of Natural and Computing Sciences

2024

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: 2024

Abstract

This project presents an improvement to data transfer times over links with high bandwidth and high delay (bandwidth-delay product). This is achieved by signalling capacity discovered on previous connections to new connections, allowing a jump in the data in flight using IETF Careful Resume. These modifications to QUIC provide up to a doubling of data transfer rates on geostationary satellite internet connections, improving the user experience for applications such as video streaming and file downloading.

Acknowledgements

I would like to thank Dr. Raffaello Secchi for his exceptional guidance as a project supervisor; and Prof. Gorry Fairhurst, and Ana Custura for providing fruitful feedback on all aspects of this project. I would additionally like to extend my gratitude to members of the IETF community who have engaged with and commented on my Internet-Drafts during this project.

Contents

Acronyms

AEAD Authenticated Encryption with Associated Data

API Application Programming Interface

BDP Bandwidth-Delay Product

BGP Border Gateway Protocol

CDDL Concise Data Definition Language

CDN Content Delivery Network

DDoS Distributed Denial of Service

DNS Domain Name System

GEO Geostationary Earth Orbit

gRPC gRPC Remote Procedure Calls

GSO Generic Segmentation Offload

HTTP Hyper-Text Transfer Protocol

HTTP/1.1 HTTP Version 1.1

HTTP/2 HTTP Version 2

HTTP/3 HTTP Version 3

I/O Input/Output

IP Internet Protocol

ISP Internet Service Provider

IW Initial Window

LEO Low Earth Orbit

Mbps Megabits per Second

NetEm Network Emulator

OS Operating System

PEP Performance Enhancing Proxy

RTT Round Trip Time

SSH Secure Shell

TCP Transmission Control Protocol

TLS Transport Layer Security

UDP User Datagram Protocol

VLI Variable-Length Integers

VM Virtual Machine

Chapter 1

Introduction

1.1 Motivation

Satellite Internet has oft been a forgotten part of the Internet infrastructure. Remote or extreme environments which require Internet access often use Geostationary Earth Orbit (GEO) Satellite Internet. Due to the nature of the distances involved, Round Trip Times (RTTs) of over a second are common. Such high delays on a network connection prove problematic for traditional Internet protocols which are unable to fully utilise the available network capacity.

This provides for a degraded user experience, especially in applications such as video streaming and file downloads where maximal utilisation of the available bandwidth is critical to a pleasant user experience.

QUIC is a new Internet transport being designed to replace the Transmission Control Protocol (TCP) as the go-to stream transport for the Internet. It has many advantages over TCP such as encryption by default and multi-stream flow control. Its primary use to this project is that it is much more extensible than TCP and is still being actively developed, so changes to support new use cases can be made with far less friction than with TCP.

QUIC includes mandatory TLS encryption, as this has become the norm for the Internet. TCP, being developed in an academic environment in the 1980s, is much more trust based than QUIC and thus can be meddled with by parties along the connection. In some cases, such as Satellite Internet connections this can be useful to enable Performance Enhancing Proxies (PEPs). These proxies fake parts of the TCP exchange to help clients deal with the problems of satellite Internet connections. Due to the encrypted nature of QUIC a different solution is needed.

1.2 Objectives

Discussion has been happening in the IETF on a protocol called Careful Resume to make better use of bandwidth in these situations [1]. At the start of this project no concrete implementation existed of Careful Resume, nor was its efficacy and affects on other Internet traffic fully studied.

This projects sets out to do the following:

- create an implementation of Careful Resume for QUIC
- design and implement a signalling protocol for communicating the data required to enable Careful Resume
- provide a protocol specification to the IETF for the devised signalling method
- test the performance of Careful Resume on several network conditions
- test the safety of Careful Resume when network conditions are poor

The result of this project would ideally be a real world improvement in the user experience of QUIC over high Bandwidth-Delay Product (BDP) links, such as satellite Internet. The code should be battle tested and ready for deployment at an Internet scale.

Chapter 2

Background & Related Work

This chapter serves to familiarise the reader with the concepts, technologies, terminology, and current challenges that will be required to understand further sections.

2.1 QUIC

Transmission Control Protocol The TCP is the dominant transport protocol on the Internet, carrying some 90% of all Internet traffic [2]. TCP provides a reliable, connection oriented, stream protocol between two endpoints (IP address and port number) [3], akin to plugging an Internet-sized serial cable between applications. TCP's near 50 year history has provided invaluable insight into designing Internet transports that can work at a global scale, informing how newer transports (such as QUIC) should be designed.

QUIC In recent years a move had been made towards newer transport protocols, some hoping to replace TCP as the general Internet transport [4]. QUIC is a progression of work started at Google to speed up their web applications and includes a few key innovations over TCP.

TCP is most commonly implemented as part of the Operating System (OS) kernel, tying updates and improvements to the TCP stack to operating system updates. This also prevents an application from using their own stack for more specialised applications where precise tuning of the transport layer would be beneficial. QUIC moves the majority of the transport control mechanisms over to userspace, whilst using User Datagram Protocol (UDP) packets on the wire, so as to remain compatible with existing routers.

QUIC could have defined an entire new Internet Protocol (IP) datagram type for its carriage, however this would cause issues with protocol ossification on the wider Internet. Whilst the standards state that a router should carry an IP packet without inspecting its contents, many routers do not transport packets other than TCP or UDP particularly well, if at all [5]. Thus, UDP was chosen as a means to carry QUIC over the Internet without running into the problems protocol ossification presents.

Connection IDs In TCP, a connection is tied to a specific tuple of (source address, source port, destination address, destination port). If a client changes

address (e.g. when moving from WiFi to mobile data) then the entire connection will have to be torn down and setup again.

QUIC abstracts over identifiers in lower layers by using a persistent Connection ID [6]. A Connection ID is assigned at the start of a connection, and allows a packet from a different source address to be attributed to the correct connection. Thus, QUIC relies very little on the semantics of the underlying datagram transport, as long as QUIC packets get from their source to their destination intact.

Transport Layer Security & Packet Protection Since the design of TCP in the 1980s, social views (and the state of the art) on encryption over the Internet have changed significantly. Gone are the days of the Internet being a trusted academic network where nobody would dare spy on otherwise unencrypted personal communications. It has become almost mandatory to use encrypted communications over the Internet these days [7], with an even increasing push towards securing all aspects of Internet communications.

QUIC has Transport Layer Security (TLS) baked in from the start, serving as a core element of connection negotiation and packet protection [8]. Every piece of data is encrypted unless it's absolutely required to be visible to an outside observer to enable the connection to happen. This serves the dual purpose of helping combat protocol ossification, as any points for extensibility are hidden from wire observers.

Version and feature negotiation Another core aspect of QUIC is that it is upgradable and extensible. TCP has, by specification, extension options. However, due to protocol ossification they are unlikely to survive a trip across the Internet. Version negotiation allows an entirely rewritten specification of QUIC to be deployed, without the need for an Internet flag day [6], [9].

The primary method for feature negotiation in QUIC is its Transport Parameters. Transport Parameters are a set of configuration options carried as a TLS extension [8]. Transport Parameters are identified by a numeric identifier and contain arbitrary binary data relevant to each parameter.

Additionally certain parameter IDs are reserved, specifically $31 \cdot N + 27$ for integer values of N. This ensures unknown parameters are ignored correctly by implementations [10], and serves as a further defence to the ossification of QUIC.

Variable-Length Integers Many parts of QUIC use Variable-Length Integers (VLI) to encode data in a space efficient manner. VLIs allow protocol fields to encode up to a 62 bit integer, whilst only taking the minimum number of bytes required to encode each integer.

The first two bits of a VLI encode $log_2(L)$ where L is the length of the integer in bytes, thus integers are encoded is 1, 2, 4, or 8 bytes, supporting 6-, 14-, 30-, or 62-bit values. The remaining bits contain the integer in big-endian encoding. This encoding is presented visually in table 2.1.

Byte 0	1	2	3	4	5	6	7
00 (L=1)	Integer value						
01 (L=2)	Integer value						
10 (L=4)	Integer value						
11 (L=8)	Integer value						

Table 2.1: Encoding of Variable-Length Integers

Streams QUIC provides multiplexing of multiple streams of data, sent independently over one connection without a blockage in one stream affecting data transfer in other streams. QUIC streams can either be reliable and ordered (akin to TCP), or unreliable datagrams (akin to UDP) [11]. Streams can be created by either party in a connection [10].

0RTT connections QUIC provides another improvement on TCP by introducing an option known as 0-RTT ([10] section 17.2.3), in which a client can start sending data to the server before completing the handshake by using TLS session resumption. The client uses a previously received TLS resumption ticket to signal which keys are being used to encrypt this early data.

Assuming that the server does not insist on a full handshake (which it may do for any number of reasons, including being unsure of the link used) the client can start downloading data within only one RTT. This can greatly speed up the loading of previously visited web-pages, especially if the page consists of many small assets served from different CDNs.

Address validation tokens QUIC does not have always require a full handshake to start a connection before sending possibly significant amounts of data. Especially in the case of a 0RTT connection the server will likely respond with a large chunk of data straight away. Being a respectful user of the Internet is important, and without proper validation an abused 0RTT connection can be an excellent vector for a Distributed Denial of Service (DDoS) attack. A DDoS is caused when a significant amount of data is sent to an endpoint that does not expect it, causing it to become overloaded and crash/fall off the Internet. It should not need to be explained why this is bad.

To avoid this situation, QUIC will selectively employ address validation if it is unsure of the source of packets initiating a connection. The primary method for doing this is Retry packets ([10] section 8.1.2). This is a challenge response protocol where the client repeats a magic token back to the server in a retry of the initial connection establishment. The act of this token traversing the Internet in both directions confirms both sides are talking to who they think they're talking to. This method is, however, not without its disadvantages. Its main issue is that it extends the time to open a QUIC connection by at least one RTT. Given the goals of this work this is evidently undesirable.

As an alternative to Retry packets is given in [10] section 8.1.3, called Address Validation Tokens. In this method a server sends a client tokens in a `NEW_TOKEN` frame to store during one connection for use during a subsequent connection. These tokens are opaque blobs that allow a server more certainty that it has talked to this client before, and can therefore bypass address validation with Retry packets.

2.2 Extending address validation

Address validation tokens are opaque blobs that the client makes no effect to inspect; this makes them an ideal place to place data specialised to the configuration of an individual server or group of servers. Work has been done on this in the past showing the ease with which these tokens can be extended and co-opted for other purposes.

Trusted out-of-band validators There are stages in loading a web-page that happen before initiating a QUIC connection. One of the most important is a Domain Name System (DNS) lookup to resolve a domain name to a server on the Internet. There is nothing in QUIC that states a token must be issued by a server, and thus work has been done on having these tokens issued in response to DNS queries. This replaces the RTT consumed by a Retry packet with the existing RTT that is required for DNS resolution. This can provide up to 400ms of savings on opening a QUIC connection on a typical residential Internet connection [12].

Shared tokens An alternative to extending the DNS to issue address validation tokens is to signal that these tokens can be shared between different servers. Consider that `youtube.com` and `google.com` are hosted on the same Google infrastructure, but without knowledge of this a client would have to store and manage different tokens for both of these sites, thinking them to be independent.

By adding a new transport parameter ([10] section 7.4) to signal groups of servers that mutually trust each other's address validation tokens the authors achieve a 142ms average decrease in connection handshake time [13].

2.3 HTTP Version 3

The initial application for QUIC was HTTP Version 3 (HTTP/3), the third major version of Hyper-Text Transfer Protocol (HTTP) deployed on the Internet. HTTP is the protocol that turns the Internet into the World Wide Web. Previous versions of HTTP (HTTP Version 1.1 (HTTP/1.1) and HTTP Version 2 (HTTP/2)) relied on TCP as their transport protocol, however HTTP/3 was designed in tandem with QUIC to take advantage of the protocol improvements provided by QUIC. HTTP/3 is a mapping of the fixed HTTP semantics onto QUIC [14], [15].

HTTP/1.1 does not include a way to serve multiple requests on one connection simultaneously, therefore multiple TCP connections are often used when

retrieving web pages over HTTP/1.1. This is not ideal for network efficiency and congestion control as no state is shared between these connections [14].

HTTP/2 introduces multiplexing to allow one TCP connection to be used for multiple HTTP requests, however this protocol still suffers from head of line blocking due to the inability for HTTP/2 to communicate state with TCP; a blockage on one stream will affect all other streams [14].

HTTP/3 draws heavily from HTTP/2 and uses many similar data encodings. Some features of HTTP/2 are now features of QUIC, whilst others are implemented in HTTP/3. Each request-response in HTTP/3 is one stream in QUIC, and streams are never re-used. This allows requests to be processed independently, taking advantage of the properties of QUIC streams.

2.3.1 Control stream & QPACK

Not all elements of HTTP/3 use independent streams. When a HTTP/3 connection is set up a control stream is used to exchange `SETTINGS` frames between endpoints. These define parameters applicable to the entire HTTP/3 connection.

HTTP/3 additionally defines a shared compressor for HTTP headers called QPACK. HTTP requests and responses often contain the same data between them (e.g. every response will likely have the same `Server:` header); QPACK allows for these common headers to be defined in a compression table scoped to the entire connection, not just an individual request [16].

An overview of the interactions between QUIC, HTTP/3 and QPACK is provided in figure 2.1.

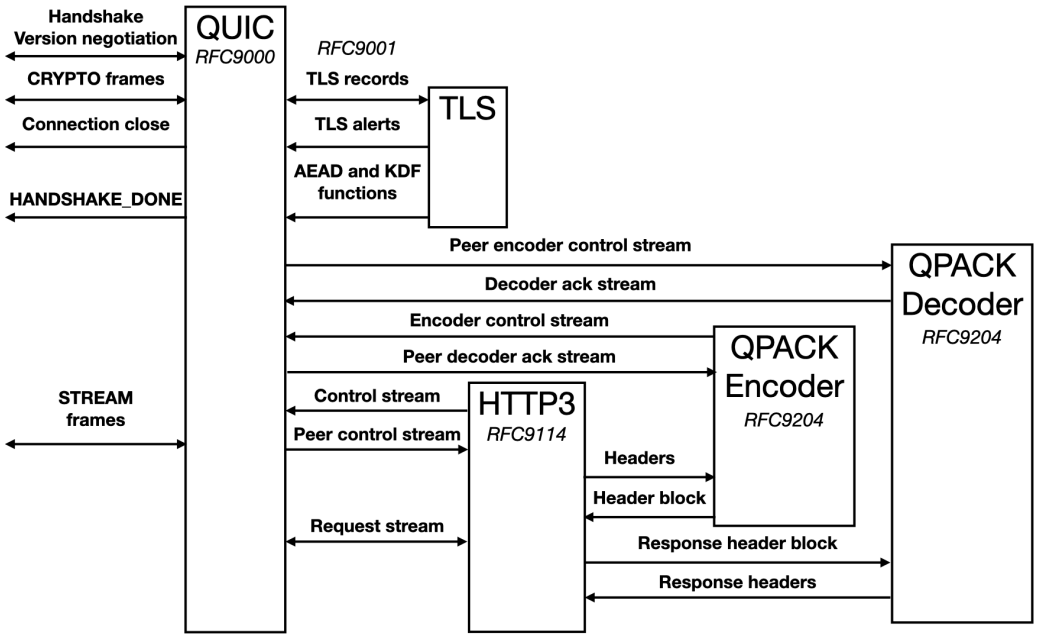

Figure 2.1: Interactions between QUIC, HTTP/3 and QPACK

2.4 Internet Congestion Control

The Internet is a shared access medium; that is, anyone can send data wherever they desire without first requesting a bandwidth allocation. Data is transferred on a best effort basis and may not be delivered if a path is overly congested. Because of this the “Requirements for Internet Hosts" [17] mandates that Internet nodes perform congestion control to ensure the stable operation of the Internet. Congestion Control is the adjustment of the rate at which data is sent based on feedback from the network [18], usually in the form of detecting lost packets, or in more recent years by using Explicit Congestion Notifications [19].

It is useful to define two types of congestion that can be experienced on the Internet:

- **Incipient congestion** is a consequence of multiplexing traffic flows at bottlenecks. The link may not be over utilised, but there exists a mere transient spike in traffic that causes some packets to have to be buffered or dropped. Such congestion is normal an entirely expected on a functioning Internet.

- **Persistent congestion** happens when endpoints overestimate the available path bandwith and send a sustained flow of packets larger than can fit through a bottleneck. This form of congestion is undesirable on a functioning Internet as it can lead to starvation of resources and congestion collapse.

Congestion collapse A poorly designed transport that results in persistent congestion can severely degrade service for all other Internet users, so called “congestion collapse" [20]. These effects were first observed during the development of the Internet in the 1980s [21], [22].

The problems of congestion collapse have generally been avoided by improvements over time in TCP congestion control, however new protocols not based on TCP (such as QUIC, and modifications to it) must be careful not to repeat the mistakes of early TCP [23].

Bandwidth-Delay Product The Bandwidth-Delay Product of a network link is a relatively simple concept. It is a measure of the amount of data in flight at any one given time, a product of the bandwidth (in bits or bytes per second) and the RTT of the path. A path with a large BDP is more commonly known as a **LFN!** [24].

TCP is extremely unfair towards flows with a high BDP. TCP’s ability to increase utilised bandwidth is limited to one packet per RTT [25]. The BDP in the thousands of packets causes TCP to take a very long time to ramp up to utilising the full available bandwidth on such links. Short flows will never be able to fully utilise these links due to the use of slow start. [26], [27]

Congestion Window A core part of a standard congestion controller is its congestion window. It is a measure of the amount of data that can be in flight (unacknowledged) at any one time. In an ideal state the congestion controller will eventually enter a steady state in which the congestion window is equal (or near) to the available BDP of the network path. It is the job of a congestion controller to make updates to the congestion window in response to feedback from the network, such as packet loss.

Initial Window At the start of a connection the congestion controller has no information about the available bandwidth of a network path. Thus, it must start conservatively to avoid creating unnecessary congestion. The Initial Window is the value to which the congestion window is set before feedback about the network path is available [28]. It represents a safe starting point for a path that is not suffering persistent congestion. Historically this was 4 packets, but in recent years a move towards 10 packets has taken place [29].

On high BDP paths the Initial Window (IW) can have a significant effect on the time to complete a data transfer. The following equation can be used to calculate data transfer time under various scenarios, assuming that TCP Slow Start [25] is in use:

$$T = (RTT \cdot (log_2(W_f/W_i) + 1)) + (L/B) + T_p \tag{2.1}$$

Where:

1. T is the connection completion time, in seconds
2. RTT is the RTT of the connection, in seconds
3. W_f is the final congestion window
4. W_i is the initial congestion window
5. L is the size of data being transferred, in bytes
6. B is the available path bandwidth, in bytes per second
7. T_p is the sum of processing time at both connection endpoints

Given this and the indicative path parameters for Low Earth Orbit (LEO) and GEO satellite connections in table 2.3, the indicative times in table 2.2 to fully saturate a network path can be calculated.

Some Content Delivery Networks (CDNs) already make adjustments to the IW based on proprietary information to speed up connections on certain classes of networks [30].

Slow Start and HyStart++ TCP was initially designed to use Slow Start to ramp up quickly from the relatively conservative IW to the real path capacity. Slow Start increases the congestion window by one packet for each acknowledgement received, effectively doubling the congestion window every RTT when there are no losses [31].

IW	LEO	GEO
10	320ms	4,320ms
100	150ms	2,661ms
200	102ms	2,161ms
400	50ms	1,690ms

Table 2.2: Comparison of transfer completion times with varying IWs

Once congestion is detected (likely because the congestion window went too high) the congestion controller enters the congestion avoidance phase, where it stays for the rest of the connection. On entering congestion avoidance the congestion window is halved to account for the likely overshoot during slow start, after which the congestion window is increased linearly instead exponentially as was the case in Slow Start. The goal of the congestion avoidance phase is to maintain equilibrium, that is maintain the congestion window as close as possible to the available capacity. This is shown graphically in figure 2.2.

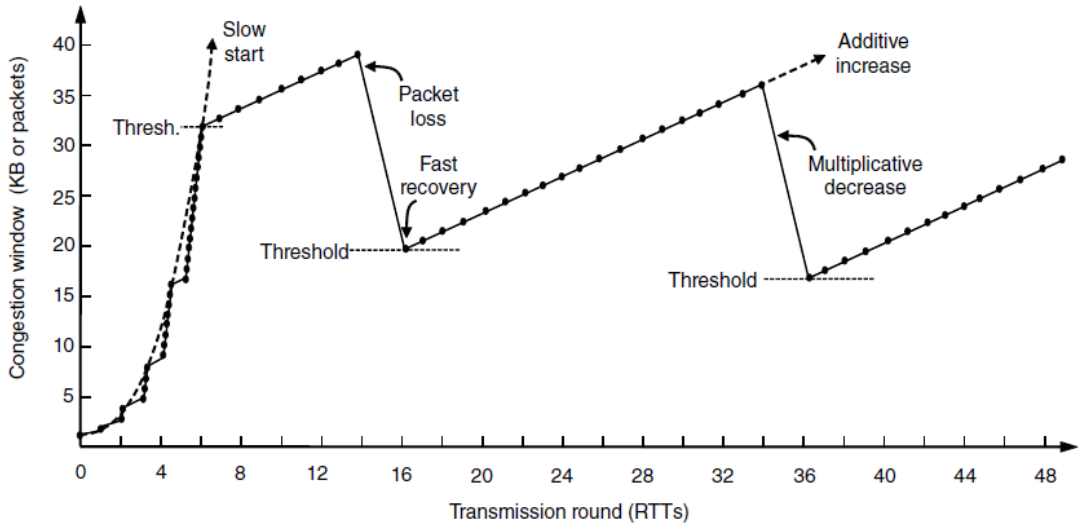

Figure 2.2: Changes to the congestion window during Slow Start and Congestion Avoidance

Slow Start will often result in an overshoot of available capacity, and manifests itself as excessive data being sent to the network during connection start-up [32]. Overshoot causes packet loss not only for the overshooting flow, but also other flows sharing the same bottleneck. This can significantly impact other flows sharing resources along a path.

To mitigate this congestion, controllers are moving towards using HyStart++ over Slow Start. It seeks to enter congestion avoidance before packets are lost by incorporating additional signals such as ACK timing heuristics or a change in RTT. HyStart++ keeps track of the minimum RTT along a path. On a

congestion free path, this variable adds up to the sum of all propagation and transmission times of the links that make up the path. Other components of RTT can be attributed to queuing and additional processing times along the route. Thus the difference between the current RTT and the minimum RTT is a heuristic for path congestion [33].

On exit from HyStart++, Conservative Slow Start is used to detect if the exit from HyStart++ was premature due to transient congestion and potentially return to HyStart++. During Conservative Slow Start, the congestion windows grows exponentially as in regular Slow Start, but with a smaller exponential base, resulting in less aggressive growth. If the RTT reduces then HyStart++ is resumed, as the trigger to exit was likely spurious. Otherwise, the congestion avoidance phase is entered.

2.5 Satellite Internet

Satellite Internet is a useful tool where traditional fixed line Internet would be impractical or prohibitively expensive. It can be used to provide Internet access to remote areas, or on a moving vehicle (such as a ship). Satellite Internet is becoming an increasingly important method for accessing the Internet. Satellite based mobile Internet is being developed into the latest 5G standards by the 3GPP, and the SpaceX Starlink service is constantly increasing in market share [34].

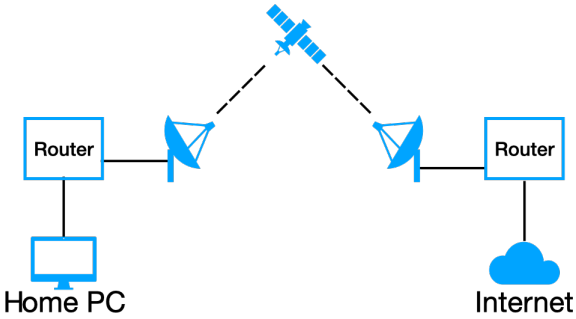

Figure 2.3: Illustration of data transfer in satellite broadband connections

Contemporary satellite Internet constellations can be categorised into two broad categories: GEO and LEO. GEO satellite are in an orbit of around 35,000km above the Earth - high enough to stay fixed above one point on Earth, whilst LEO satellites orbit between 600km and 1,200km and are constantly moving across the surface of the earth.

With GEO satellites, the round-trip-time for a connection is at least 440ms (purely from the speed of light), however with other delays (such as processing and buffering at network interfaces) accounted for RTTs of 600ms are typical [35]. An RTT of 50ms is typical of SpaceX's Starlink LEO constellation, closer to the typical RTT of a residential broadband connection, however with much higher variability and packet loss [36].

The bandwith delay product for various typical path bandwidths (from 10

Megabits per Second (Mbps) to 500 Mbps) on typical LEO and GEO satellite connections are shown in table 2.3.

	LEO (rtt=50ms)		GEO (rtt=600ms)	
Capacity	BDP (bytes)	BDP (packets)	BDP (bytes)	BDP (packets)
10 Mbps	65,536	49	786,432	583
20 Mbps	131,072	97	1,572,864	1,165
50 Mbps	327,680	243	3,932,160	2,913
100 Mbps	655,360	485	7,864,320	5,825
500 Mbps	3,276,800	2,427	39,321,600	29,127

Table 2.3: Bandwidth-delay products of various satellite links

2.6 Transmission Control Protocol Performance Enhancing Proxies

Given the challenges TCP faces with the BDP of typical GEO Satellite Internet (as discussed in section 2.4), solutions to this have been sought to make browsing the Internet a more pleasurable experience on GEO Satellite Internet.

To alleviate the issues associated large BDPs and random losses Satellite Internet, providers have traditionally utilised TCP Performance Enhancing Proxy to hide these delays from applications using the link [4].

Figure 2.4 shows the working principle of a TCP PEP. PEPs break the end-to-end principle of the Internet by breaking the TCP connection into three parts - TCP between the client and the satellite terminal, a proprietary TCP-like protocol over the satellite link, and TCP between the ground station and the Internet. The Internet Host on either side of the PEP communicate using entirely standard TCP and are unaware of the PEP in the middle of the connection.

By faking ACK packets based on the known bandwidth available over a satellite, a PEP can influence a congestion controller to set its congestion window close to the ideal congestion window in a much shorter space of time.

2.7 Stateful Transmission Control Protocol

A concept that has been around since the 1990s to improve TCP performance on successive connections is that of interdependence between the control state of TCP connections [37]. There is little to no point in maintaining and calculating congestion control information for multiple connections to the same destination as they will all come to the same conclusion about the network path. This idea has not seen widespread deployment in spite of later updates to the standard [38].

Variations on the basic ideas in the IETF standards have been presented by many. One example of this is Stateful-TCP in which BDP parameters from a previous connection are stored and reused on new connections. On new connections the peer IP address is looked up in a hash table and the congestion

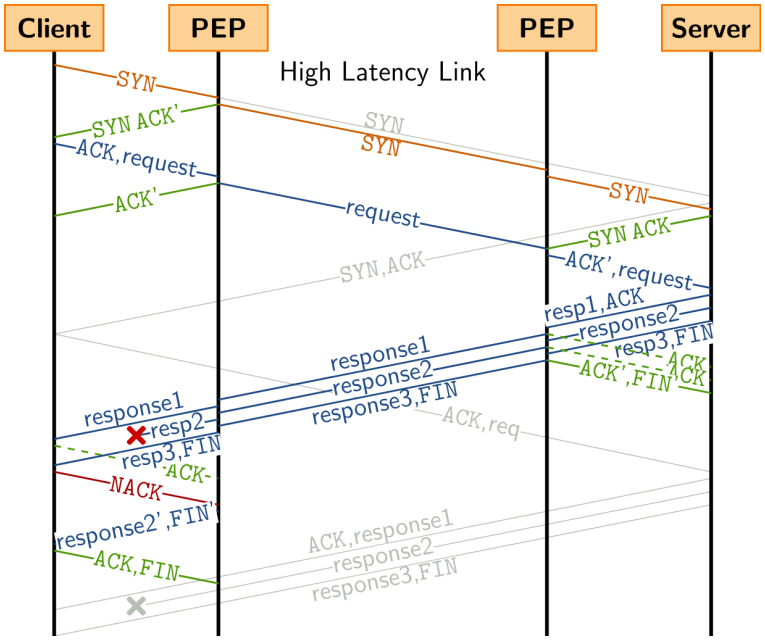

Figure 2.4: Working principle of a TCP PEP
© Sedrubal, CC BY-SA 4.0, via Wikimedia Commons

window is initialised to the BDP from a previous connection [39]. This can significantly speed up connections on links with high BDPs and does not require action by the client.

2.8 Careful Resume

Another idea on the same basic principles in the works at the IETF is the Careful Resume protocol. This once again re-uses congestion control parameters between connections, but provides a more well defined specification for exactly how they are re-used and provides additional safeguards to ensure the congestion controller is not causing harm to itself or other Internet users. It additionally has the advantage of being written without assuming the use of TCP, allowing it to be used with any Internet transport, such as QUIC [1].

Careful Resume primarily stores two pieces of information between connections: the previous congestion window and the previous RTT. The exact method for this storage is not defined and is the subject of this work. Additional data such as expiry is at the discretion of the implementer.

Careful Resume uses the saved RTT to determine if the path is likely the same as the one on which the data was stored. If the RTT is less than half the previous RTT or greater than 10 times the previous RTT it is deemed that this is not the same link and it is not safe to proceed with careful resume.

If the link is suspected to be similar to the previous link, the congestion window is jumped to half the previous congestion window - a significant increase, but only half to avoid too much effect on other connections sharing the same bottleneck. The Careful Resume algorithm then provides methods to check that this wasn't a dangerous jump and a method to retreat to safety if it was

before significant impact is had on other flows. This project's implementation of Careful Resume is discussed in greater detail in section 4.3.

2.9 Detecting link bandwidth

In spite of the general inability to reserve bandwidth on the Internet, or to know exactly how much bandwidth is available, there are some techniques available in specific situations that allow a network endpoint to gain information about available bandwidth on a network link.

Border Gateway Protocol Link Bandwidth Extended Community Cisco proposed in a now expired Internet-Draft an extension to the Border Gateway Protocol (BGP) for a router to signal to its peers what bandwidth is available on the link between them [40].

The BGP is the protocol used to communicate between different Internet Service Providers (ISPs) about the reachability of different parts of the Internet over each ISP. For example; ISP A might have a BGP connection ("peer") with ISP B. ISP A can then say to ISP B over that that connection 'Hello, I'm ISP A you can reach 2001:db8:a::/48 through me'. With the extension Cisco proposed the message would instead take the form 'Hello, I'm ISP A you can reach 2001:db8:a::/48 through me, the bandwidth for this link is 100 Mbps'.

Unfortunately, this extension was not standardised nor widely deployed. Whilst its signalling may be useful to address the problem at hand in this project, the lack of any known implementation in the wild makes this a purely academic extension, and therefore is not discussed further.

Starlink gRPC Remote Procedure Calls Interface The Starlink access terminal (Dishy) listens for Secure Shell (SSH), HTTP, and gRPC Remote Procedure Calls (gRPC) connections on `192.168.100.1` no matter what addressing scheme is in use on the network it is providing Internet access to. These Application Programming Interfaces (APIs) are not public documented by Starlink, but they run on standard protocols and thanks to the hard work of some reverse-engineers we have documentation of which calls are available and what data the terminal can provide [41].

These APIs have been investigated during this project for their usefulness in providing feedback on available link bandwidth or situations where the connection may not perform well. This is discussed further in section 4.5.

2.10 Tools and programmes used

Rust Rust is a young systems programming language which provides memory safety without the performance penalties normally associated with such languages. It was chosen for its efficiency at low-level tasks whilst providing excellent safety guarantees to the programmer. Traditionally, memory safety is achieved in a language by giving up some control over how the programmer interacts with memory; typically, a garbage collector is employed, restricting

control over memory layout and deallocation, and introducing performance penalties in the form of reference counting. Giving up such control is simply not an option in systems programming, thus Rust was designed to be a language offering absolute memory safety but retaining compatibility with use in systems programming contexts [42].

More than mere memory safety, the extremely expressive type system of Rust provides safety against more than what a traditionally safe language can offer. It can protect against errors such as iterator invalidation [43] (where an iterator is made invalid in some form by mutations occurring during iteration) and against data races (ensuring there is no unintended communication between threads with shared memory).

Rust gives the programmer minute control of data layout in memory, and places heavy emphasis on zero-cost abstractions [44], where nothing unnecessary is included in the final output, and the standard library provides helpers that won't cause additional performance issues.

> What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

All of this makes Rust an ideal choice for developing an implementation of a modern transport protocol. Its low-level memory management abilities allow efficiently working with network packets, and its expressive type systems allows compile time elimination of whole classes of bugs that are otherwise inevitable in today's inherently multi-threaded Internet applications.

Cloudflare Quiche Quiche (`https://github.com/cloudflare/quiche`) is an open-source implementation of QUIC in the Rust programming language maintained by Cloudflare, Inc. Cloudflare themselves use Quiche to power their CDN's QUIC implementation, making it an battle tested and reliable implementation of QUIC. Quiche is additionally used in software such as `cURL`, Mozilla `neqo`, and `netty` [45].

Quiche serves as the base QUIC implementation on which the experimentation work in this project was completed. Quiche provides only the QUIC state machine, handling packet processing and connection state. It is the responsibility of the application to provide an event loop and I/O support to exchange Quiche's state with the network. Details of the work undertaken to provide this event loop are detailed in section 4.1.

QLog Understanding why a connection behaved in a specific way can be an extremely challenging task of de-tangling all the interacting state machines that form a network endpoint.

To aid with debugging network protocols, the IETF QUIC working group have written (and are currently working on standardising) the QLog specification [46]. This is especially useful in the context of QUIC, as little (if any) data can be obtained from QUIC's encrypted wire image. Instead, logging has to occur

within each endpoint after decryption. This also allows various internal state parameters that would never be transmitted over the wire to be saved within the log for later inspection.

QLog itself is not specific to QUIC; it is a generic, extensible, schema that provides for a shareable, aggregatable and structured logging format. Different documents define specific QLog events for protocols such as QUIC and HTTP/3 [47], [48].

QLog files output from Quiche were used extensively in testing and debugging this project, providing invaluable insight into the internals of why things aren't working as expected.

Chapter 3

Design

Given problems laid out in the previous chapter, it was decided at the start of this project that its aim would be to produce an implementation of Careful Resume for QUIC deployable at an Internet scale.

3.1 Requirements

Careful Resume stores congestion control parameters between connections to allow re-using these parameters with a new connection on the same path. As currently specified, it has no discussion on how these parameters are persisted between connections, and some research is required to arrive at a method that works in most (or ideally all) situations.

Whatever extensions that are made to QUIC should be backwards compatible with existing Internet deployments of QUIC, that is deployable without a flag day [9]. The work should also not cause a slow down or other degradation of service to Internet endpoints unaware of this work.

3.2 Congestion Control Data Storage Location

BDP information can be stored primarily in two places, on the server, or on the client. We will ignore the case of storing data with a trusted third party as ultimately this will entail either the server or the client (or both) talking to said third party at some point during the connection. Both have their advantages and their disadvantages, and one may be more suitable than the other depending on the network infrastructure and corporate politics at play.

In the rest of this document it is assumed that the endpoint receiving the QUIC connection (the server) is the entity with a large amount of data to transfer to the endpoint initiating the QUIC connection (the client), whereas the client has very little data to transfer to the server. Examples of such scenarios include browsing the web with HTTP or downloading files with FTP. Careful Resume has also been suggested for video upload from clients, however this presents a slightly different set of challenges and is therefore out of scope for what is only a BSc project.

Storage on the server In this method, a client can be entirely unaware that a server is using Careful Resume; the server is storing data locally and acting

entirely on its own in adjusting its congestion controller. This method has the primary advantage that it requires no action on the part of the client. The client can initiate new connections oblivious to the fact that Careful Resume is being used by the server, and (hopefully) magically get better performance.

This method however has significant disadvantages for cloud providers who operate large clusters of load balanced servers, or in fact anyone operating a highly visited website. Consider the case case of a small operator (or even an individual) running their website on a single server or a small group of servers. Keeping track of stored information for the relatively small number of connections they'll receive is for all practical purposes a non-issue. However, in the case of such large cloud providers such as Cloudflare, Fastly, and Google keeping such state between all of their servers is a significant engineering challenge, and alternative options should be sought.

Storage on the client For the reasons above it is more appealing in the contexts of cloud providers to offload the storage of information to the client. Clients are, generally, single devices operating with one link to the Internet, such as mobile phones, or desktop computers. Clients of this kind also handle on the orders of magnitudes less connections than a server in a cloud provider. A server may handle hundreds of users at once, but a person isn't likely to have more than a few web browser tabs active at once.

It would be a relatively small burden on the client to receive a piece of information from the server for storage, and later return to the server; similar protocols already exist such as the TCP SYN cookie or cookies in HTTP. These all allow a client to show information from a previous session to the server, allowing personalisation of the connection. This is exactly what is needed to enable Careful Resume to happen. This solves the problem of distributed state across a load balanced set of servers, as the server that receives the package of information is the only one that needs to know about it.

This also creates an opportunity for client interaction, further discussed in section 3.5.

Which to choose? There is nothing that says we have to choose one or the other. These two locations of storage can co-exist on the Internet, with some servers choosing to manage things entirely and some choosing to outsource data storage to clients. If a server wishes to store congestion control information locally there is nothing preventing it from doing so. The server may do it in any way it pleases and no standardisation work is required to enable this.

There are definite use cases for storing this information on the client, and given that this requires co-operation between client and server standardisation work is required in this area. This is the primary focus of the project. A method for server side storage of information will be specific to each deployment of QUIC and work in this area would provide little if anything that can be shared

between different deployments. A standardised method for client-server communication provides more value to the Internet as widespread adoption of a new protocol requires agreement on what the protocol is.

3.3 Congestion Control Data Storage Method

There are several different methods that could be used to communicate information to be stored on the client, all with their advantages and disadvantages. Some can be discounted immediately, however some are worthy of further investigation.

TLS Session Tickets TLS contains within it a concept of session resumption (also discussed in section 2.1), in which a client can initiate a new TLS connection to a server it has previously communicated with without having to complete a full TLS handshake [49]. TLS defines a session ticket as an opaque label that the client returns, thus limiting the possibility for client interaction (see section 3.5).

There exists an extension mechanism that would allow the TLS server to signal that the ticket is not in fact opaque and that the client can decode it as some well-known structure to obtain information that would be useful for client interaction. There are, however, issues with this approach. The first is more of a philosophical issue, in that this ties information about the transport layer to information on the application layer. QUIC as it in currently deployed may require TLS, however this does not preclude unencrypted QUIC (although the value of this is debatable given what we now know about Internet security [50]). More likely something will come along in future to replace or update TLS rendering this extension incompatible and in need of reworking.

The other issue is more practical; in Quiche (as an example) the BoringSSL library is used to provide TLS routines. While BoringSSL supports session resumption tickets, it does not expose any API for inspecting their contents, and does not allow the application to add its own data to them [51]. The situation is similar with other TLS libraries.

One final issue with TLS session resumption tickets is that it only allows this information to be transmitted in 0-RTT sessions. There is value in supporting careful resume on 1-RTT sessions as well as 0-RTT, for example because 0-RTT and the 1-RTT contexts are separate security contexts or because an application does not implement 0-RTT TLS.

For the above reasons we can discount TLS session tickets as a viable avenue to pursue.

QUIC transport parameters QUIC contains a method to convey additional information about connection properties in the form of transport parameters [10]. A new transport parameter could be defined to convey congestion control information when initiating a connection.

This method seems on the surface rather neat and in keeping with the design ethos of QUIC, however has one major problem: transport parameters are only sent at the start of the connections. Some other method would have to be defined to convey new congestion control parameters to the client as the connection progresses. There is no useful congestion control information available at the start of a connection so the server would have nothing of use to send to the client in the transport parameters. Additionally transport parameters are re-used without re-negotiation on a 0-RTT connection, thus rendering this method useless for such connections as no new information could be conveyed.

For this reason we can also discount QUIC transport parameters as a method to carry congestion parameters. We should however not discount transport parameters as a method to negotiate and signal transmission of congestion parameters via a different means, as this is precisely what transport parameters have been designed into QUIC for [10].

Something at a higher layer This method is a little vague because the higher layer could be anything: FTP, HTTP, SSH, etc. It has the same (if not a stronger) philosophical issue as with TLS session tickets; it ties transport information to something at a layer that is not transport. A method for communicating congestion control parameters would have to be defined for *every* protocol that runs on top of QUIC. There would also be an application engineering problem of passing this data back and forth between the transport and application layers in the application.

This method is likely to cause significant issues and frustrations in future, and so can also be discounted.

Address validation token During discussions on the IETF QUIC working group mailing list, it was raised that QUIC contains a method called tokens that could be used to store this information[1]. These are further discussed in section 2.1. A transport parameter could be defined that allows a client to know that the token has some defined structure rather than being an opaque blob to allow for for client interaction (see section 3.5).

The use of the address validation token allows another neat behaviour that no other options provide: clients that are not aware of a this standard will simply echo back the token. This allows the desired offload of congestion control state from the server to the client without requiring any modification to the client. This solves the usual chicken-and-egg problem of new protocols on the Internet where neither party has the motivation to implement the standard as nobody else implements it yet.

This method appears to be a workable solution that is worthy of pursuing

A new QUIC frame The final solution to be discussed is defining an entirely new frame within QUIC. All communication within a QUIC connection happens

[1] `https://mailarchive.ietf.org/arch/msg/quic/gxgT6KT6Vl6LLExIkUmAOY5vgUM/`

in frames, and if more features are to be added to QUIC they're generally implemented in the form of frames [10]. This allows the most flexibility as the semantics of what the frame contains, when to send it, and the meaning of the frame, are all free to be defined in a standard. There are already well defined methods within QUIC for defining new frames, so this approach should have little issue with implementation.

The main disadvantage of this method is that it requires implementation by both clients and servers. This presents the chicken-and-egg problem discussed above. Servers will have little incentive to implement it if no clients supports it, and clients will have little incentive to implement it if no server supports it. This problem could be alleviated if one large cloud provider implements the new frame, starting the snowball of more clients and more servers implementing it. This method also appears workable and is worthy of more discussion.

Storage Method Conclusion It was therefore decided early into the project that work would continue on researching and evaluating, implementing, and discovering flaws in both the address validation token and the frame method. Both appear to have their advantages and disadvantages and at this early research stage is was not clear that one was superior to the other.

3.4 Congestion Control Data Storage Contents

With both a new frame and address validation tokens some well defined data (for client interaction, see section 3.5), and some opaque data (to allow for extension by the server) will be exchanged between the server and the client. The next question to be answered is what exactly should that data be?

There is also a desire in the QUIC working group not to make the whole semantics of this data well-known, as a server may wish to experiment with what data it stores on the client without requiring coordination with the client[2].

The first concern to address is that of validating information from the client. If the server blindly trusts congestion control parameters from the client then a malicious - or merely broken - client could cause the server to perform a DoS attack against itself or its network by sending for more data than can fit onto the path. The data stored on the client therefore has to be authenticated in some way. One method to do this would be a HMAC [52] over the congestion control data, however this should be left as an exercise to the implementer. The client has no need to verify the data, and there may be other considerations on the server side that influence how this data is authenticated. The specific implementation used in this project is discussed in section 4.4.

The server will also want some confidence that the path being used is the same (or substantially similar) as that of the previous connection where the congestion control information was generated. If a calculated bandwidth is used on a different path of a lower capacity, this could overload the new path.

[2] `https://mailarchive.ietf.org/arch/msg/quic/_FgnGQxWEFVoo67lgXirDdkgNBE/`

The exact method to determine the same link is discussed in greater detail in Careful Resume [1]. It is left to server preference what information to store to validate the path. The token used to verify the path need not have any meaning to the client as the client would have nothing it could do with the information.

An additional desire of this information is an expiry, as after a certain period of time its likely that even on the same path available bandwidth will have changed, or the underlying path is different after a substantial period of time. The exact length of this expiry requires further experimentation, and is also likely subject to different factors in different deployments.

As an aside, the client could be using privacy addresses [53] and regularly rotate its IP to improve its privacy. The server therefore likely shouldn't use merely the IP address as the sole method for identifying a network path. Alternatives such as matching by subnet or other proprietary information should be used.

The best method therefore seems to be a structure containing the following (or similar) information:

1. An endpoint/link verification token
2. An expiry field
3. Parameters to allow for client interaction (as per Careful Resume)
   (a) Computed round trip time
   (b) Computed path bandwidth
4. An optional request from the client to use a lower path bandwidth (see section 3.5).
5. Some opaque data for the use of the server only
6. Some cryptographic verification over important data, specific to each deployment

3.5 Client Interaction

A client may become aware of a change in its link to the rest of the Internet or the capacity available on that link (as discussed in section 2.9). It may in this case wish to signal this knowledge to the server to allow more appropriate congestion control parameters to be used, avoiding an overshoot of link capacity. This overshoot can have dramatic consequences both for the connection in question, and other connections sharing the same link [54].

The client could take a few actions based on information it knows about the available link capacity.

- Not send the BDP data at all, causing a fall-back to more traditional congestion control and avoiding Careful Resume altogether.

- Ask for a different link bandwidth estimate to be used if it knows that in the BDP data is now inaccurate.

- Ask for half the link bandwidth if it is starting two QUIC connections over the same link (or a third for three etc.). However, this requires more thought as two common endpoints does not mean one common path (such as when ECMP is in use). Further discussion of this specific avenue is out of scope for this project but it is presented as a prompt for possible future work.

Additionally, the QUIC client could pass this link knowledge, and stored BDP data, to a higher layer. This could help influence decisions in applications such as MPEG-DASH [55] about which stream to use based on what the available link capacity is and what the server's congestion controller is likely to do with the link.

3.6 Bandwidth-Delay Product Frame

The initial avenue perused for conveying the information needed for Careful Resume to function was a new frame within QUIC. This seemed ideal as by defining an entirely new frame we have full control over its semantics within QUIC.

A new QUIC transport parameter (`enable_careful_resume_indication`) was defined to allow a client to signal that it accepts and understands this new frame, as is required by the QUIC standard to allow a new frame to be used. When the server feels it has sufficient information (as defined by local policy) on the BDP of the path, it sends this new frame to the client for it to store. The format of the frame was envisaged as follows, using the encoding specification defined in QUIC [10]:

```
BDP_FRAME {
  Type (i) = 0xTBD,
  Lifetime (i),
  Saved Capacity (i),
  Saved RTT (i),
  Saved Endpoint Token (...)
  Hash (...)
}
```

As this was never deployed at an Internet scale, the frame type integer has not been defined. The other fields convey the following information, fulfilling the requirements laid out in section 3.4:

- **Lifetime** A Unix timestamp [56] of the time at which this data will be considered invalid.

- **Saved Capacity** The last computed congestion window capacity of the link, in bytes.
- **Saved RTT** The last computed RTT of the link.
- **Saved Endpoint Token** Server specific information to identify the network path at a later time
- **Hash** A server specific cryptographic hash over important information to prevent client meddling.

When starting a new connection, a client can request that Careful Resume is used and return the BDP information in another frame, or send a different frame to request that Careful Resume is avoided (client interaction). These frames were eventually never defined as we had abandoned BDP frames in favour of address validation tokens.

The main disadvantage of the new frame method is that it creates a chicken and egg problem. That is, clients have to implement it before its of any value to servers, and its unclear what the incentives are for implementation. This would likely slow adoption, as this is already a niche problem that most QUIC implementers would have little interest in spending their time on anyway.

3.7 QUIC Address Validation Tokens

During discussion on the QUIC working group mailing list about the specifics of implementing the frame based solution, it was raised that QUIC already contains an address validation token that could be extended to support this use case [3].

Address validations tokens are opaque data that are echoed back to the QUIC server on a new connection by the client already. This solves the chicken and egg problem; servers can start sending BDP information to clients without clients knowing what they contain, and it will work with every QUIC stack already deployed. Later, clients who are aware of the nature that these tokens have been extended will find value in implementing code to examine their contents and perform client interaction.

Using the already discussed transport parameters, a server can signal to clients that it has extended the address validation token and that there are meaningful data in the address validation token, to allow for client interaction.

An initial attempt at this defined the following format for address validation tokens:

```
BDP Token {
    Address Validation Length (i),
    Address Validation (..),
    BDP Data Length (i),
```

[3] https://mailarchive.ietf.org/arch/msg/quic/e7x6rEvJY0FAP_pgclEYoXFoeJ8/

```
    BDP Data Value(..),
    Saved Capacity (i),
    Saved RTT (i)
}
```

The address validation fields allow for the original purpose of the token to be preserved, and the other fields provide for the same BDP data as was contained within the frame to be conveyed. The endpoint validation and hash data have been moved to a more generic BDP Data field, to allow the server to put whatever information it desires in there without needing an update to the standard. This was raised as a desirable characteristic by multiple people in the QUIC working group, notably Cloudflare who stated this would be useful to experiment with different congestion controllers without the client having to be aware of these experiments.

3.7.1 Extensible Tokens

It was realised later in the project that extending tokens this way is somewhat limiting on future work. If such a fixed structure was defined for address validation tokens it would be impossible to define new data that the client could read inside these tokens without starting from scratch. An extensible method of putting data in the address validation token was therefore required.

A later revision of the draft standard therefore defined Extensible Address Validation tokens. These take the following form:

```
BDP Token {
    Address Validation Length (i),
    Address Validation (..),
    Extension ID (i),
    Extension Data Length (i),
    Extension Data (..),
    ...
}
```

The address validation field is still present to fulfil the original purpose of the tokens in QUIC, but additional data is added via numbered extensions. A registry for these extension IDs can be maintained in the same way other ID registries are maintained for QUIC ([10] section 22).

The extension block can be repeated many times to include as many or as few extensions as the server desires. The client need not understand every extension, it will only inspect those for which it understands what the ID means, and it cares to do anything with.

These extensible tokens can initially be used only for conveying BDP information, but provide an extension point for future work by others where they wish to store data on a client without a client having to negotiate a new protocol to do so.

3.7.2 Bandwidth-Delay Product Tokens

Given the above move to Extensible Tokens, the BDP information extension was defined as follows:

```
BDP Token {
    Private Data Length (i),
    Private Data (..),
    Capacity (i),
    RTT (i),
    Requested Capacity (i),
}
```

BDP data has been renamed to private data to make it clearer that this is data specific to the server and is of no relevance to the client, and the other fields retain the same meaning. This private data should contain the server's integrity information. An example of how this might be achieved is presented in section 4.4.

An addition made at this stage is the requested capacity field, to allow for greater expressibility in client interaction. In previous iterations the client only had two options, send the data and request Careful Resume, or send nothing and avoid Careful Resume. With the new requested capacity field the client can state its knowledge of link capacity to the server. The server must consider this request if and only if the requested capacity is equal to or less than the previously discovered and saved capacity. This is to avoid a network overload by a misbehaving client. If the network capacity is indeed more than the previously saved capacity then it is left to the congestion controller to discover this, not client request.

A copy of the current state of the draft standard is included in appendix A for reference.

3.8 Careful Resume QLog

An important part of designing and implementing a new standard is to be able to test and debug its behaviour. As part of this project it was therefore required to define extensions to QLog to convey the internal state of the Careful Resume implementation to aid in its debugging. An explanation of QLog is provided in section 2.10.

QLog events are defined using the Concise Data Definition Language [57]. The Concise Data Definition Language (CDDL) allows for QLog structures to be defined that can be mapped to a multitude of encodings depending on what is best suited for each situation such as CBOR [58] or JSON [59].

In keeping with other events in QLog it was decided the best time to output events relating to Careful Resume was when a state change of the Careful Resume FSM occurs. That is when Careful Resume moves from Reconnaissance

to Unvalidated or from Validating to Safe Retreat. Additional data is attached to this event relating some some internal state variables of Careful Resume, and a reason for this state change is provided. These reasons are a great insight into seeing if the Careful Resume implementation is detecting network conditions correctly, or instead reacting spuriously.

The reasons available for a state change are:

- `packet_loss` Packets have been lost on the network, indicating likely congestion.
- `ECN_CE` Careful Resume has received an Explicit Congestion Notification [19] from the network and is reacting to it [4].
- `congestion_window_limited` The application has more data available to send than the Congestion Window, necessitating that Careful Resume be used to jump the Congestion Window to a higher value.
- `cr_mark_acknowledged` Packets sent during the Unvalidated and Validating phase have now been acknowledged, allowing the congestion controller to have greater confidence in the true available bandwidth.
- `rtt_not_validated` The RTT of the network is substantially different than that of the previous connection, so Careful Resume has been disabled to avoid causing network congestion.
- `exit_recovery` Careful Resume has decided the congestion event is over at the congestion controller can continue as normal

[4] ECN is not currently implemented in any version of Careful Resume

Chapter 4

Implementation

4.1 Quiche Tokio Wrapper Library

Cloudflare Quiche in and of itself does not perform any interaction with the network; it is a pure state machine. It is up to the application developer to handle interfacing with the OS and in turn the network. As a first step in implementing this project it was therefore necessary to build a wrapper around the Quiche state machine to handle network traffic.

The go-to library for asynchronous event based programming in Rust is Tokio. Tokio provides a runtime allowing Input/Output (I/O) tasks to be completed concurrently, and working with a network is an inherently concurrent operation [60].

When a connection is opened a UDP socket is setup (listing 1), the Quiche state machine is initialised (listing 2), and a new task is started to handle passing messages between Quiche and the network in the background (listing 3 and 4). Message queues are setup to handle passing messages from the host application to the Quiche state machine for processing (listing 5).

These message queues are Multi Producer Single Consumer, meaning that multiple threads in the host application can interact with the QUIC connection. This is useful, for example, to divide streams between different threads and enable greater concurrent use of the QUIC connection. Helper functions, such as that in listing 7, provide a cleaner interface to users of the wrapper library than directly interacting with the message queues.

This state machine can be pushed forward by three things:

- **Packet received** When the operating system notifies Tokio that there are new packets available to process the task is woken up and passes these packets through the Quiche state machine to process.

- **Control message received** When the host application wishes to perform an action with the QUIC connection it is passed to the task in the form of a control message. These are then processed to cause the correct actions to be taken on the Quiche state machine.

- **Timeout expired** Sometimes Quiche needs to perform an action, but cannot perform it immediately (for example when congestion control forbids it). In this case it will provide a timeout the task that once expired with wake up the task and cause the Quiche state machine to make progress again.

After all of these events the task checks if the Quiche state machine ended up in a connection closed or error state and propagates this up to the host application if it is the case (listing 6).

```
let socket = socket::UdpSocket::new(bind_addr).await?;

// Pacing is discussed later
if !socket.has_pacing() {
 config.enable_pacing(false);
}

let local_addr = socket.local_addr()?;
debug!("Connecting to {} from {}", peer_addr, local_addr);
```

Listing 1: A new UDP socket is initialised for network communication

```
let conn = quiche::connect(
  server_name, &scid, local_addr,
  peer_addr, &mut config
)?;
```

Listing 2: Setting up the Quiche state machine

Streams Streams are handled in a similar fashion to the outer connection wrapper. Each stream object contains a reference to the control message queue and the shared connection state object. Each stream in turn implements `tokio::io::AsyncRead` and `tokio::io::AsyncWrite`, allowing it to be used a generic I/O object with other parts of the Tokio library.

There are a few additional functions on the stream objects that are QUIC specific such as:

- `stream_id` Returns the unique ID of the the stream within the QUIC connection
- `is_server` Is this stream a server initiated stream
- `is_bidi` Is this stream a bi-directional or uni-directional stream

These functions are not common to other generic I/O objects, so required implementation outside of a Tokio trait.

A diagrammatic overview of the interactions between different objects is presented in figure 4.1.

```
let recv_socket = socket.clone();
tokio::task::spawn(async move {
  let mut buf = [0; 65535];
  loop {
    let (len, recv_info) = match recv_socket.recv_dgram(
      &mut buf
    ).await {
      Ok(v) => v,
      Err(e) => {
       error!("Failed to read UDP packet: {}", e);
       break;
      }
    };

    if let Err(_) = packet_tx.send(
      (buf[..len].to_vec(), recv_info)
    ).await {
      break
    }
  }
});
```

Listing 3: Forwarding received packets to the Quiche state machine

Pacing Pacing plays an important role in a better behaved QUIC implementation. By default congestion controllers can be quite bursty; that is, send out a large tranche of packets and then wait a while for the next tranche. This can have a negative impact on the network as it can cause temporary congestion. Pacing, as the name implies, causes these tranches to be sent out with a gap between each packet.

To save the host application from having to constantly context switch to the operating system to send paced packets (an expensive operation) the Linux Generic Segmentation Offload (GSO) mechanism was used to let the operating system handle the pacing of these packets. Each packet sent to the OS has a timestamp for when it should be sent on the network attached to it [61].

Pacing is not implemented in the standard Tokio wrapper around Linux UDP sockets, so a custom implementation had to be written. The key piece of code is to enable the `TXTIME` extension on the socket to let the OS know that pacing will be used (listing 8). This code is in an `unsafe` block because it interacts directly with OS memory and may violate Rust memory safety rules. Much OS level programming in Rust requires these `unsafe` blocks because of this. The pacing timestamp can then be set on the control message header of the packet message to the operating system as shown in listing 9.

```
let mut packets = vec![];
loop {
  let (write, send_info) = match inner.conn.send(&mut out) {
    Ok(v) => v,
    // There are no more packets to send
    Err(quiche::Error::Done) => {
      break;
    },
    Err(e) => {
      self.set_error(e.into()).await;
      break 'outer;
    }
  };
  packets.push((send_info, (&out[..write]).to_vec()));
}

for (send_info, packet) in &packets {
  if let Err(e) = inner.socket.send_dgram(packet, send_info).await {
    self.set_error(e.into()).await;
    break;
  }
  trace!("{:?} sent {} bytes", inner.scid, packet.len());
}
```

Listing 4: Sending packets out of the Quiche state machine to the network

4.2 HTTP Version 3

To test this work in a more real-world environment an implementation of HTTP/3 was developed on top of the Tokio wrapper library. HTTP/3 is by far the most common application layer above QUIC [4], so it made sense to test these modifications to QUIC using the application for which they'll most likely be used.

The HTTP/3 library is written in a way to be as agnostic as possible to the QUIC library used underneath. As long as the QUIC library provides streams as Tokio async read/write objects and a few basic connection management functions on the QUIC connection itself it would take a small amount of work to switch the backing QUIC library.

To setup a HTTP/3 connection, first the settings and QPACK encoder/decoder streams are created and configured. This happens first to allow time for the same to happen on the peer's side and for the setup data to start making their way across the network. The library then listens for new streams opened by the peer until all required streams for a HTTP/3 connection are setup and configured (control, QPACK encoder, QPACK decoder).

Once all streams are setup, a new task is spawned to handle the control stream over the duration of the connection. No requests are transferred over the control steam, only background information relating to management of

```
let (control_tx, control_rx) = tokio::sync::mpsc::unbounded_channel();
let (new_stream_tx, new_stream_rx) = tokio::sync::mpsc::channel(25);
let (new_token_tx, new_token_rx) = tokio::sync::mpsc::channel(25);
let (new_cr_event_tx, new_cr_event_rx) = tokio::sync::watch::channel(None);

let shared_connection_state = std::sync::Arc::new(SharedConnectionState {
  connection_established: std::sync::atomic::AtomicBool::new(false),
  connection_established_notify: tokio::sync::Mutex::new(Vec::new()),
  connection_closed: std::sync::atomic::AtomicBool::new(false),
  connection_closed_notify: tokio::sync::Mutex::new(Vec::new()),
  connection_error: tokio::sync::RwLock::new(None),
  application_protocol: tokio::sync::RwLock::new(Vec::new()),
  dcid: tokio::sync::RwLock::new(None),
  peer_token: tokio::sync::RwLock::new(None),
  transport_parameters: tokio::sync::RwLock::new(None),
});
```

Listing 5: Setting up communication channels between the host application and Quiche

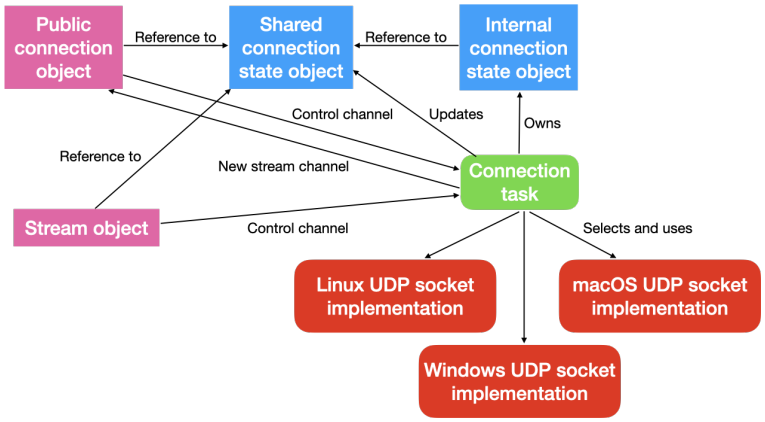

Figure 4.1: Interactions of different components in the Quiche Wrapper Library

the entire HTTP/3 connection. Similar tasks are spawned to accept and respond to messages on the QPACK streams. If the connection is configured as a server an additional task is spawned to handle incoming requests, decode their headers, and pass this to the host application for further processing. The setup function then returns control to the host application for it to continue as it pleases.

Each request in HTTP/3 runs on its own stream. Therefore each requested is turned into its own object (listing 10) with storage for headers and trailers, as well as taking ownership of the stream for that request to allow for body data transfer. The request object provides a few helper functions for transferring headers and trailers, but overall HTTP/3 works as a generic binary stream transfer protocol once headers are exchanged.

```
if inner.conn.is_closed() {
  if let Some(err) = inner.conn.peer_error() {
    self.connection_error
      .write()
      .await
      .replace(err.clone().into());
  } else if let Some(err) = inner.conn.local_error() {
    self.connection_error
      .write()
      .await
      .replace(err.clone().into());
  } else if inner.conn.is_timed_out() {
    self.connection_error
      .write()
      .await
      .replace(std::io::ErrorKind::TimedOut.into());
  } else {
    self.connection_error
      .write()
      .await
      .replace(std::io::ErrorKind::ConnectionReset.into());
  }
  self.set_closed().await;
  break;
}
```

Listing 6: Checking for error conditions on the Quiche connection

A diagrammatic overview of the interactions between objects implementing HTTP/3 is presented in figure 4.2.

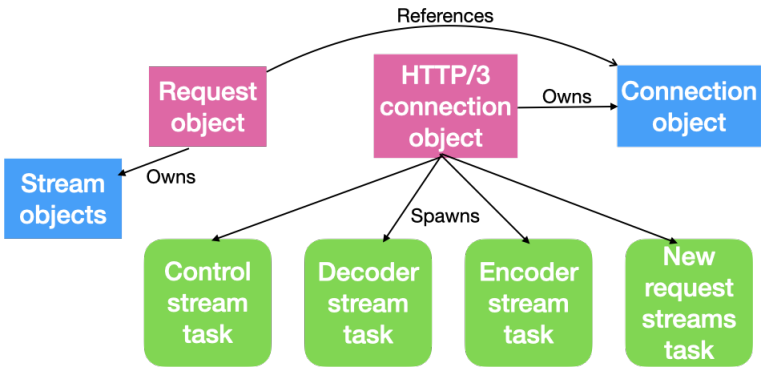

Figure 4.2: Interactions of different components in the HTTP/3 library

Implementing QPACK was a more technically interesting challenge due it being much more algorithm based and less about purely moving data from A to B. Header lines can be compressed in a multitude of ways with QPACK:

- Static table indexed
- Dynamic table indexed

```
pub async fn setup_careful_resume(
  &self, previous_rtt: Duration, previous_cwnd: usize
) -> ConnectionResult<()> {
  let (tx, rx) = tokio::sync::oneshot::channel();
  self.send_control(Control::SetupCarefulResume {
    previous_rtt,
    previous_cwnd,
    resp: tx
  }).await?;
  match rx.await {
    Ok(r) => r,
    Err(_) => Err(self.make_error().await)
  }
}
```

Listing 7: A helper function to pass the Careful Resume setup message into Quiche

```
unsafe {
  libc::setsockopt(
    fd.as_raw_fd(), libc::SOL_SOCKET, libc::SO_TXTIME,
    std::mem::transmute(&tx_time_opts),
    std::mem::size_of::<libc::sock_txtime>() as libc::socklen_t
  )
}
```

Listing 8: Enabling pacing on a Linux socket

- Dynamic table offset indexed (post base indexed)
- Literal value with indexed name (static, dynamic, or dynamic offset)
- Literal value and name

Literal values can be encoded as their raw byte value, or can be compressed with a Huffman table specifically engineered for HTTP/3 [16]. The static table is a list of common header names and values so that they do not have to be repeated constantly on the wire. The dynamic table is built by each sender for fields that it thinks will be repeated many times in many requests over the HTTP/3 connection. The dynamic table is the only piece of global state that has to be synchronised between sender and receiver, all other forms of encoding headers can be decoded with reference to some well known data tables.

For a more efficient operation of the static table it is stored in the output library as a binary tree. However, this would be difficult to maintain in the source code so the Rust `build.rs` facility was used to turn an easily editable list into a binary tree on each compile. If there is a filed called `build.rs` in the project's root directory the Rust compilation suite will run this file before compiling the rest of the source code, allowing complex code/data generation operations to

```
let cmsg: &mut libc::cmsghdr = unsafe {
  std::mem::transmute(libc::CMSG_FIRSTHDR(&msg))
};
cmsg.cmsg_level = libc::SOL_SOCKET;
cmsg.cmsg_type = libc::SCM_TXTIME;
cmsg.cmsg_len = unsafe {
  libc::CMSG_LEN(std::mem::size_of::<u64>() as u32) as usize
};
unsafe {
  *(libc::CMSG_DATA(cmsg) as *mut u64) = send_time;
}
```

Listing 9: Enabling pacing on a Linux socket

```
pub struct Message {
    headers: qpack::Headers<'static>,
    trailers: Option<qpack::Headers<'static>>,
    stream: tokio::io::BufStream<quiche_tokio::Stream>,
    shared_state: std::sync::Arc<SharedConnectionState>,
}
```

Listing 10: Data stored for each request

take place during the compilation stage. A similar approach was used to convert the list of entries in the Huffman table (as it is presented in the standards document) into a binary tree for decoding and a lookup table for encoding. An extract of the input data to the build script is presented in listing 11 and an extract of the output binary tree is presented in listing 12.

```
// Tuple contents: huffman encoding, length of encoding in bits,
// target byte value
const HUFFMAN_TABLE: &[(u32, u8, u8)] = &[
    (0x1ff8, 13, 0),
    (0x7fffd8, 23, 1),
    (0xfffffe2, 28, 2),
    (0xfffffe3, 28, 3),
    (0xfffffe4, 28, 4),
```

Listing 11: Input data to the build script

4.3 Modifications to Quiche

Given the close interaction between Careful Resume and QUIC it was infeasible to implement some parts of this project as an overlay of Quiche, some changes had to be made to Quiche itself. Careful Resume itself had to be implemented, functions to pass around address validation tokens added, and new QLog events defined in code.

Quiche, in the state when work was started, had incomplete support for address validation tokens in Initial packets. That is, it only fully implemented

```
static HUFFMAN_TREE: &Node = &Node::Node(
  &Node::Node(
    &Node::Node(
      &Node::Node(
        &Node::Node(
          &Node::Terminal(48),
          &Node::Terminal(49)
        ),
        &Node::Node(
          &Node::Terminal(50),
          &Node::Terminal(97)
        )
      ),
```

Listing 12: Binary tree output from the build script

address validation through Retry packets. Some new fields (listing 13) had to be added to the Quiche connection object to support the tokens. Support functions such as `set_token`, `send_new_token`, `peer_token`, and `recv_new_token` were added to allow a host application to interact with this new functionality. Additionally the transport parameters encoding and decoding was updated to support the new field to signal extensible tokens.

```
/// Address verification token to send in the INITIAL packet
token: Option<Vec<u8>>,
/// Address verification token to send in next
/// packet in a NEW_TOKEN frame
new_token: Option<Vec<u8>>,
/// Received address verification token from the INITIAL packet.
peer_token: Option<Vec<u8>>,
/// Queue of tokens received in NEW_TOKEN frames
received_tokens: VecDeque<Vec<u8>>,
```

Listing 13: New fields to support address validation tokens

Congestion control in Quiche is handled by the `recovery` submodule. A new `resume.rs` file was added to this module to implement Careful Resume. This file provides an implementation of the Careful Resume state machine with hooks to be called from the main congestion controller. The Careful Resume state machine is given the opportunity to update itself every time a packet is acknowledged (function `process_ack`), a packet is sent on the network (function `send_packet`), or congestion is detected (function `congestion_event`). Corresponding modifications to the congestion controller were made to call these functions at the correct time.

This file additionally implements the mechanism for deciding when to save

updates in congestion control parameters and send them to the peer for storage. The exact conditions for this are not defined in the Careful Resume specification and are instead left to individual preference. The one constraint from Careful Resume is that the congestion control parameters are not saved if the congestion window is less than 4 times the initial window.

The criteria I have decided upon for my implementation has two triggers. First, if there has not been an update in 60 seconds then there must be an update at the next available opportunity. Otherwise the following conditions (where t is the seconds since last update) are checked, and if any are true an update is made:

$$RTT_{min_{new}} < RTT_{min_{old}} - \frac{RTT_{min_{old}}}{t} \tag{4.1}$$

$$RTT_{min_{new}} > RTT_{min_{old}} + \frac{RTT_{min_{old}}}{t} \tag{4.2}$$

$$CWND_{new} < CWND_{old} - \frac{CWND_{old}}{t} \tag{4.3}$$

$$CWND_{new} > CWND_{old} + \frac{CWND_{old}}{t} \tag{4.4}$$

This allows smaller changes to be saved without spamming updates in a short space of time if the values are very jittery. Once the congestion controller has decided that an update should be made it is passed to the host application via the `next_cr_event` function, for it to do with this update as it pleases.

Other more minor modifications to Quiche included adding the ability to configure flow control limits on individual streams. It was discovered during development that the flow control limits for individual streams climbed in step with the congestion window under normal operations, but when Careful Resume makes its large jump the stream flow limits are unable to keep up and the extra capacity provided by Careful Resume is therefore underutilised on a connection with a low number of streams. Quiche was therefore modified to allow setting these higher (based on knowledge from saved Congestion Control parameters) via the `setup_default_stream_window` and `stream_max_data` functions.

A final modification worthy of note is that by default Quiche assumes the RTT to be $333ms$ until it has data on the actual RTT. Given that the previous RTT is known when using Careful Resume Quiche was modified to allow the host application to set the assumed RTT based on its own data using the `set_initial_rtt` function. This can help with more accurate pacing of the first few packets of a connection.

4.4 Extensible Tokens

Extensible Tokens are implemented in their own library, allowing independence of this piece of code from underlying QUIC implementations. The layout for an Extensible Token is shown in listing 14; this format with an unknown extension type allows for host applications to add their own code for handling experimental/private token extensions. At present the BDP token is the only defined and implemented extension available. Functions to encode to/decode from the wire format are also provided.

```
#[derive(Debug, Clone)]
pub struct ExToken {
    pub address_validation_data: Vec<u8>,
    pub extensions: std::collections::BTreeMap<ExtensionType, Vec<u8>>
}

#[repr(u64)]
#[derive(Debug, Copy, Clone, Ord, PartialOrd,
  Eq, PartialEq, num_enum::FromPrimitive,
  num_enum::IntoPrimitive)]
pub enum ExtensionType {
    BDPToken = 1,
    #[num_enum(catch_all)]
    Unknown(u64)
}
```

Listing 14: Layout of the Extensible Token object

```
#[derive(Debug)]
pub struct PrivateBDPData {
    ip: std::net::IpAddr,
    expiry: chrono::DateTime<chrono::Utc>,
}
```

Listing 15: Private data in the BDP Token

In this implementation, the private data encoded is the IP address to which the token was sent and the expiry after which the token will be ignored (as shown in listing 15). This data is put into the Private Data field in a BDP Token after encryption with the ChaCha20-Poly1305 cipher. This cipher provides both encryption (as otherwise private data such as the IP address would be visible on the wire), and data authentication [62] ensuring the token hasn't been modified.

The ChaCha cipher additionally is an Authenticated Encryption with Associated Data (AEAD) cipher, meaning that plaintext (unencrypted) data can be authenticated as unmodified. This features allows for smaller BDP tokens as the previous capacity and RTT fields used for client interaction can be authenticated and thus don't need to be repeated in the encrypted data.

4.5 Starlink Monitoring

As discussed in section 2.9 the Starlink user terminal provides an (officially unsupported) gRPC interface to retrieve various statistics about its state. Starlink gRPC tools [41] were used to collect these statistics at a regular interval. This piece of software is distributed as a Docker container image allowing easy deployment on the machine used for testing. Stats are collected every 60 seconds by the tool and fed into an InfluxDB bucket.

InfluxDB is an industry-standard time series database for storing, querying, and manipulating large amounts of data across long timespans [63]. It stores the instantaneous data point from the Starlink terminal to allow them to be inspected at a later date, or refined to more meaningful data. Grafana, another industry-standard tool, is used to plot the data stored in InfluxDB and to allow for its graphical exploration [1]. An example of the output from Grafana is provided in figure 4.3.

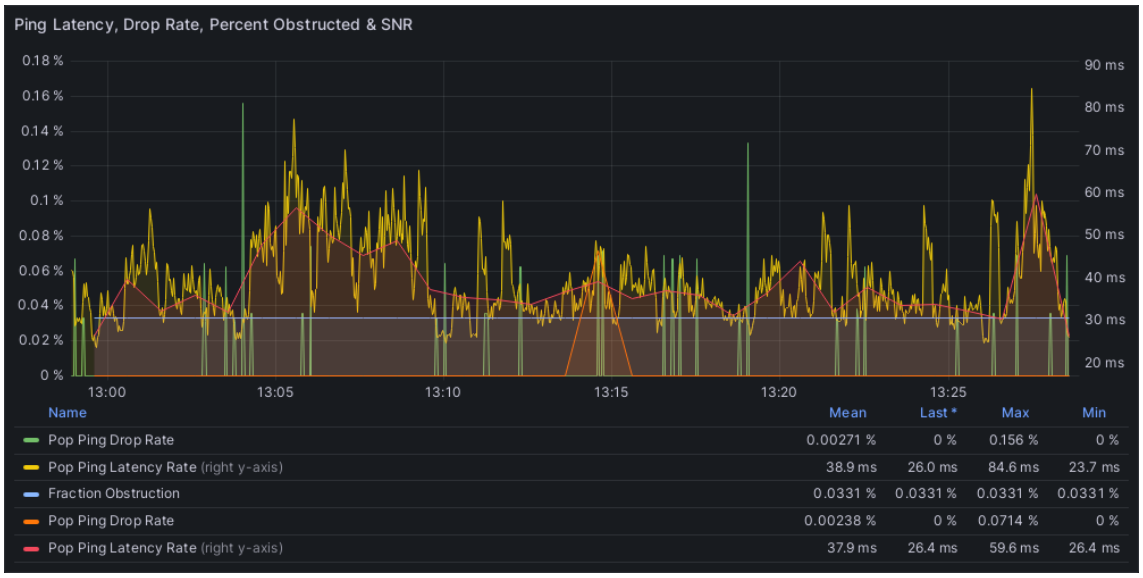

Figure 4.3: A plot from Grafana of Starlink terminal data

Ultimately, after using Grafana to explore the data provided by the terminal, it was discovered that none of the values would provide a suitable indication of available bandwidth for client interaction. The most promising field initially was `downlink_throughput_bps`, however it turns out this only contains the current utilised throughput, not the predicted available throughput. No further work was done on data from the Starlink monitoring interface.

The user terminal does however provide constant measurements of RTTs to various parts of the Starlink network (satellite, ground station, and the wider Internet). Using the RTT measured as a heuristic to determine changes in bandwidth was not considered due to lack of time to adequately research the effects

[1] The Grafana dashboard is available to view at `https://grafana.sat-vm.y-cwmwl.co.uk/public-dashboards/c440608b1e7c49eebd10951ea3479ad8`

of changes in Starlink's RTT on available bandwidth.

Chapter 5

Testing

When developing something intended to be deployed at an Internet scale it is vital to be certain of its behaviour before letting it loose on millions of computers. This chapter documents the methodology used to test the code and the behaviour of Careful Resume. Results are discussed in more detail in chapter 6.

5.1 Unit-testing

5.1.1 Careful Resume

Given the importance of having a well behaved congestion controller on the Internet, the first priority in testing was unit testing the implementation of Careful Resume to ensure its behaviour in code matches with what the specification says should happen. This in important especially for ensuring compatibility with existing implementations of QUIC. The specification sets out how an implementation is interoperable, and these unit tests ensure the code conforms to these interoperability rules.

Rust provides a neat way of implementing such unit tests. Tests are standard functions, marked with a test decorator. A test passes if it does not crash, so to make a test fail the function need only make the programme crash - using the likes of `panic!()`, `.unwrap()` and asserts. An example of this is provided in listing 16. Given the test code is at the same namespace level as the code under test there are no problems accessing private fields to test the internal state of objects.

A simple test for Careful Resume is provided in listing 17. This test calls functions on the Careful Resume implementation as they would be called by the Congestion Controller during a connection and validates that the implementation ends up in the correct state. Longer and more comprehensive tests create an entire congestion controller with Careful Resume enabled and call functions for packets arriving to and departing on the network, again verifying the state at the end is as expected.

The full unit-test suite for Careful Resume is available in the `resume.rs` file.

During testing, one very specific edge case was discovered in the specification itself for Careful Resume. There was, at the time, no protection against

```rust
#[test]
fn a_rust_test() {
  // This test will pass as 1 is obviously 1
  assert_eq!(1, 1);
}
```

Listing 16: An example of how a unit test is written in Rust

```rust
// for a set rtt that meets the conditions and assuming cwnd = jump
// window already, check we move to unvalidated
#[test]
fn valid_rtt() {
    let mut r = Resume::new("");
    r.setup(Duration::from_millis(50), 80_000);
    let jump = r.send_packet(Some(Duration::from_millis(60)), 20_500, 20, false);
    assert_eq!(jump, 19_500);

    assert_eq!(r.cr_state, CrState::Unvalidated(20));
    assert_eq!(r.pipesize, 20_500);
}
```

Listing 17: A simple test for Careful Resume functionality

Careful Resume setting the congestion window to 0 (or a similarly low value) and completely stalling the congestion. Careful Resume only enters the unvalidated phase when there is more data to send than the Congestion Window allows. It then keeps track of how much data is sent during the unvalidated phase as the amount of data it knows is safe to send over the network. On leaving the unvalidated phase the congestion window is reset to this safe known value. In certain circumstances there could be little to no data sent during unvalidated phase, meaning the congestion window is reset to a very low value. The fix for this was to change the specification to say that the congestion window is reset to the maximum of the known safe data and its value before entering the unvalidated phase.

The circumstances that caused the discovery of this bug were also a bug. In an effort to make efficiency improvements in the handling of packets in Quiche, a bug was introduced in which acknowledgements were counted twice, resulting in the congestion controller thinking very little data was actually in flight on the network.

5.1.2 QPACK

The QPACK RFC [16] provides in its Appendix B a comprehensive list of encoder and decoder instruction example encodings, and their corresponding meaning. This allowed tests such as those in listing 18 to be written. These tests take one of the example encodings and run them through the decoder to check that a) the decoder does not crash/error and b) the decode is correct. The decoded object is then passed through the encoder and compared to check that

it is identical to the input binary data.

Additionally, some entire header blocks are provided in the standard. These were used to check higher level features of the QPACK implementation such as interactions with the static and dynamic tables. The Huffman compression implementation was tested by passing various strings into and out of the compressor, and ensuring their decompressed value was the same as the value before compression.

```
#[tokio::test]
async fn test_set_dynamic_table_capacity() {
  let data = hex::decode("3fbd01").unwrap();
  let mut reader = tokio_bitstream_io::BitReader::new(std::io::Cursor::new(&data));
  let encoder_instruction = EncoderInstruction::decode(&mut reader).await.unwrap().unwrap()
  if let EncoderInstruction::SetDynamicTableCapacity(c) = encoder_instruction {
      assert_eq!(c, 220);
  } else {
      panic!("Expected SetDynamicTableCapacity");
  }

  let mut writer = tokio_bitstream_io::BitWriter::new(std::io::Cursor::new(vec![]));
  encoder_instruction.encode(&mut writer).await.unwrap();
  let encoded = writer.into_writer().into_inner();
  assert_eq!(data, encoded);
}
```

Listing 18: A test for the QPACK encoder and decoder

5.2 Integration Testing

After ensuring that each piece of code worked in isolation to implement Careful Resume it was then important to test that all parts working together produce the intended result. This entails a server and a client communicating over a network and exchanging BDP tokens to conduct Careful Resume on their next connection. Each test involves the client making a HTTP GET request for a file, and measurements about the file's transfer time and other characteristics of the transfer being taken.

5.2.1 Linux Network Emulator

In order to have complete control over the network in which these tests were conducted the Linux Network Emulator (NetEm) was used; a real network with real routers and switches would be too difficult to control. NetEm allows the existing Quality of Service features of the Linux Kernel to be used to add delay, packet loss, bandwidth constrictions, and other characteristics to a network interface [64].

They key use of NetEm was to add high amounts of delay to connections, simulating a high BDP connection. In addition to this NetEm was used to simulate different path bandwidths and packet loss rates to simulate congestion

on the network, and test the response of Careful Resume to these undesirable conditions.

5.2.2 Docker

To conduct a proper integration test it was necessary to have independent network endpoints talking to each other. Traditionally this would have been achieved with full blown Virtual Machines (VMs) connected over a physical network switch. This, however, can be very slow for running a large suite of tests as the time it takes to setup and boot a VM is the same as that of a normal computer because a full kernel boot process is involved. It is also difficult to orchestrate VMs to perform a fully automated test suite; custom programmes would have to be written to handle communication to an orchestrator and to configure the VM kernel appropriately.

As an alternative to this the Docker containerisation system[1] was used to host each end of the connection. With Docker containers the host Linux kernel is used with its namespacing feature used to separate programmes out into their own network and process domains. Virtual switches can be created between these containers, and NetEm used to simulate network conditions. Because of this very lightweight way of simulating computers tests can be spun up and torn down much quicker, allowing for more extensive testing to take place with greater ease. Docker provides an extensive API[2] suitable for orchestrating the kinds of tests needed for this project.

5.2.3 Testing Orchestration

The test suite is defined by a YAML file containing a list of tests to run, and various network conditions for each test be run under. The variables than can be configured for each test are:

1. Test name
2. Number of repeats
3. Congestion Controller used (Reno, CUBIC, BBR, etc.)
4. Enable/disable pacing
5. The file to be transferred (i.e. files of different sizes)
6. RTT
7. RTT volatility
8. Network bandwidth limit
9. Packet loss percentage

[1] `https://www.docker.com`
[2] `https://docker-py.readthedocs.io/`

A Python script runs through this test file and creates Docker containers for every test. It creates a new directory for each test that is mounted into the container to collect result data files, and configures NetEm as appropriate for each test. Each run contains data for the first connection (before BDP tokens have been exchanged and stored) and a second connection (with BDP tokens available from the previous). The result data consists of a QLog file for each connection from the perspective of the server - the server has the large amount of data to transfer and so is the party actually performing Careful Resume - and a JSON file consisting of some summary statistics such as the average data transfer rate over the connection.

5.2.4 QLog

To assist in understanding the internals of a QUIC connection, QVis[3] is usually used to visualise QLog files. This had two problems for this project: firstly it does not support the new QLog elements added for Careful Resume (although this could be patched in) but more importantly is existence as a web app makes it very difficult to orchestrate. A different solution that would allow automatic generation of plots from the QLog for analysis was needed.

To this end, another Python script was written to read QLog files and feed them into the Matplotlib[4] library. This script is configurable and can plot the following values from the QLog files:

- Careful Resume Phase (as a change in background colour)
- Careful Resume Pipesize
- Congestion Window
- Bytes in flight on the network
- Smoothed RTT
- Connection flow credit data limits
- Stream flow credit data limits
- Total data sent
- Pack loss events (as vertical lines)

The script coalesces these plots into a PDF with each test connection and labels them with the test conditions, to allow for easier review of behaviour. Examples of these plots are included in chapter 6.

[3] `https://qvis.quictools.info`
[4] `https://matplotlib.org`

Chapter 6

Results

Figure 6.1 shows a download of a 1MiB file over a simulated 10Mbits 800ms RTT link, without the use of Careful Resume. We can see the doubling effect of SlowStart throughout the connection, indicating the congestion controller never achieves the full BDP of the link. It takes nearly 7 seconds to transfer this file, at an average rate of 1.12Mbits - a mere tenth of the available link bandwidth.

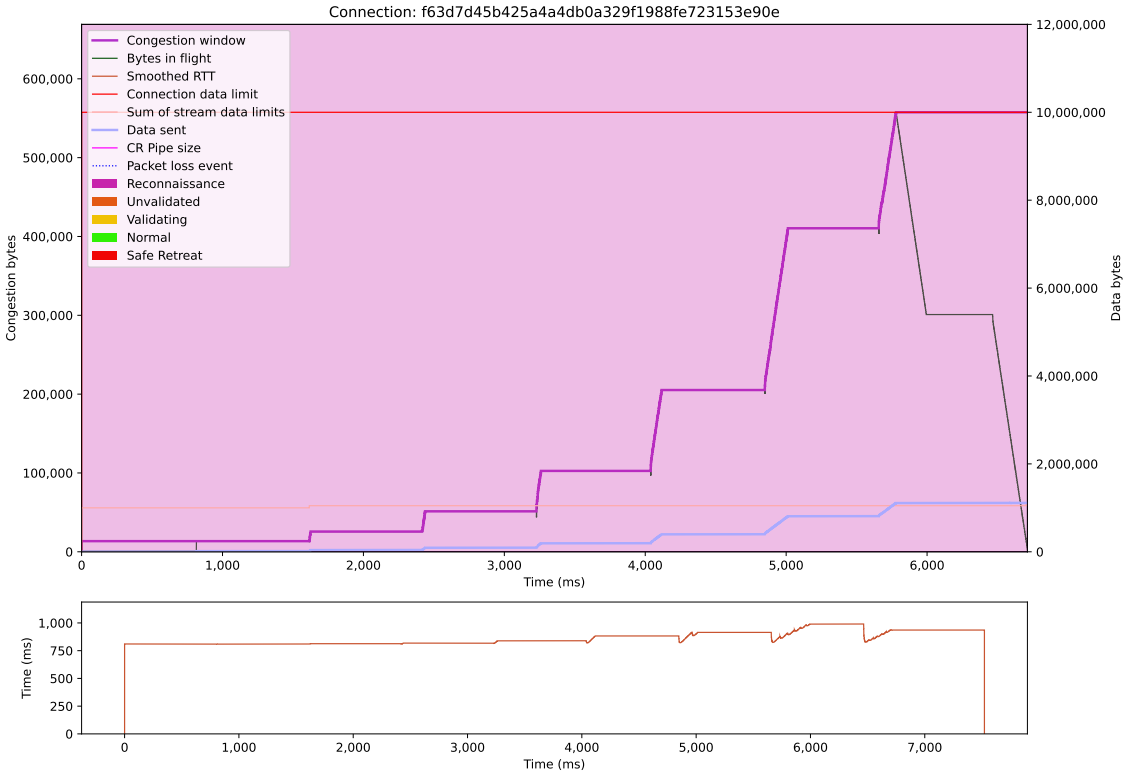

Figure 6.1: A QUIC connection downloading a 1MiB file over a 10Mbits link with 800ms of RTT, without Careful Resume

Figure 6.2 shows a repeat of the same test, but utilising BDP tokens from the previous test to enable Careful Resume. The transfer completes in a much quicker time of around 4.5 seconds, with an average data transfer rate of 1.88Mbits. A significant portion of the time is still spent in connection setup (around 1.5 seconds) and draining of buffers after the transfer (also around 1.5

seconds). For a larger file transfer the effect of Careful Resume on the effective data transfer rate would be greater than this near doubling.

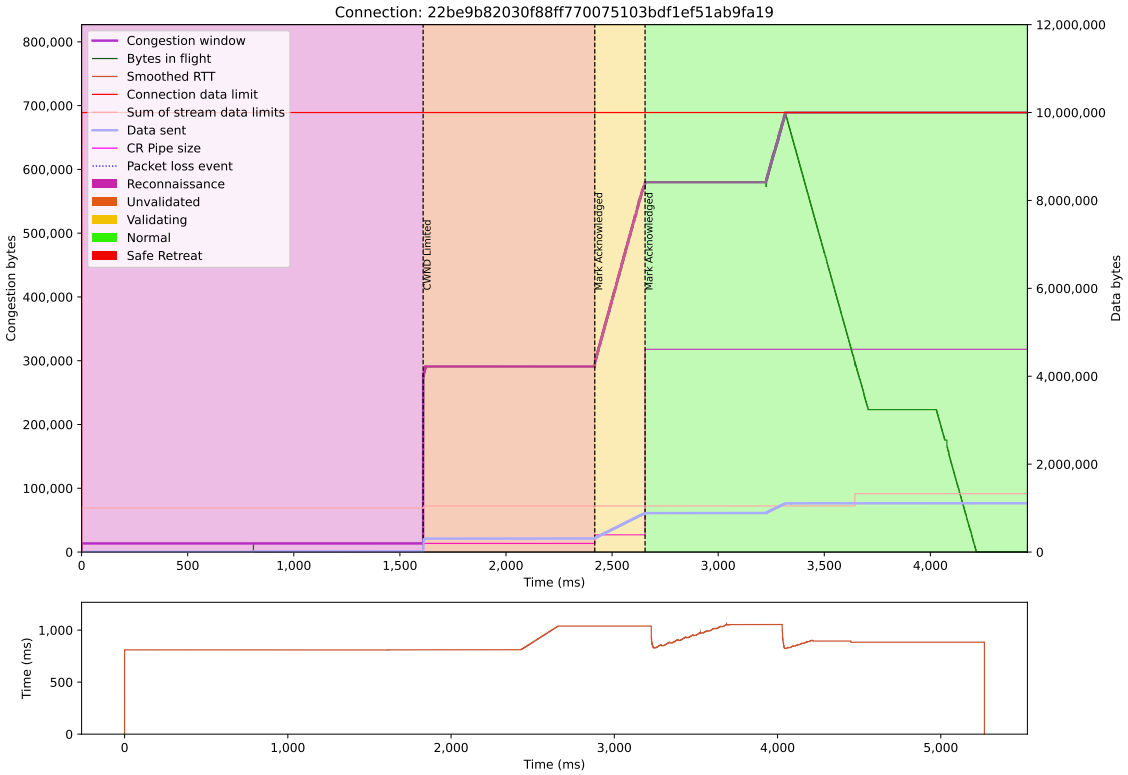

Figure 6.2: A QUIC connection downloading a 1MiB file over a 10Mbits link with 800ms of RTT, with Careful Resume

Figure 6.3 shows the larger effect Careful Resume has on connection data rates as RTT increases. All tests for this figure were conducted with 1MiB files over a 10Mbits link. Connection 1 is conducted without stored BDP tokens and thus without Careful Resume. Connection 2 has stored tokens from Connection 1, and thus uses Careful Resume. Below around 100ms of RTT the effect of Careful Resume isn't really noticeable, however an improvement on the absolute minimum and maximum is observed for 100ms, improving the experience in the extremes. Above 200ms the BDP of the connection becomes greater than the congestion controller can handle on its own and Careful Resume begins to have a significant effect. Beyond 300ms a roughly doubling in the effective data transfer rate can be observed.

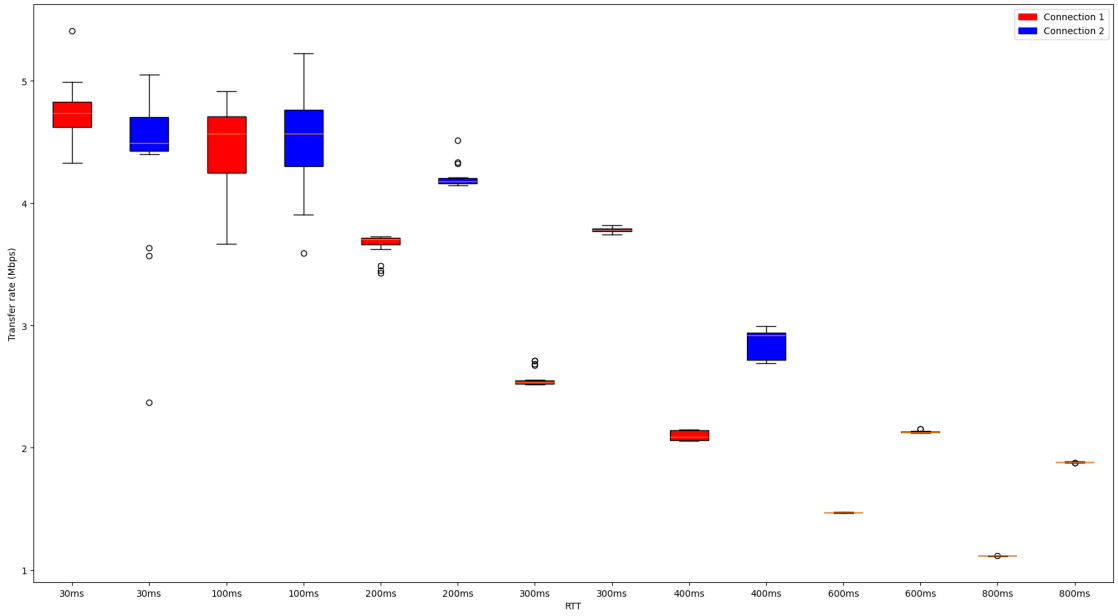

Figure 6.3: Transfer rates of different connections with and without Careful Resume with a 1MiB file

With figure 6.4 the tests were conducted under identical circumstances as figure 6.3 but with a 10MiB file being transferred instead of a 1MiB file. The differentiation between connections with and without Careful Resume becomes apparent later (at around 200ms instead of 100ms) and drastic changes in effective transfer rates do not happen until around 400ms. This is because the much larger files provides a longer transfer, and so more opportunity for the congestion controller to adapt to the link's BDP on its own. This can be see in figure 6.5; Careful Resume is only relevant at the start of the connection to help it get up to speed. The congestion controller having more time to adapt, and more importantly guess, is also the cause of the larger variation on effective transfer times. This test shows that even with longer running connections with larger files than is typical of MPEG-DASH[1] Careful Resume is still useful at increasing overall transfer rate and thus improving user experience.

[1] A 1080p 3Mbits stream put through NGINX's RTMP module generated MPEG Transport Stream files chunks of roughly 800KiB each

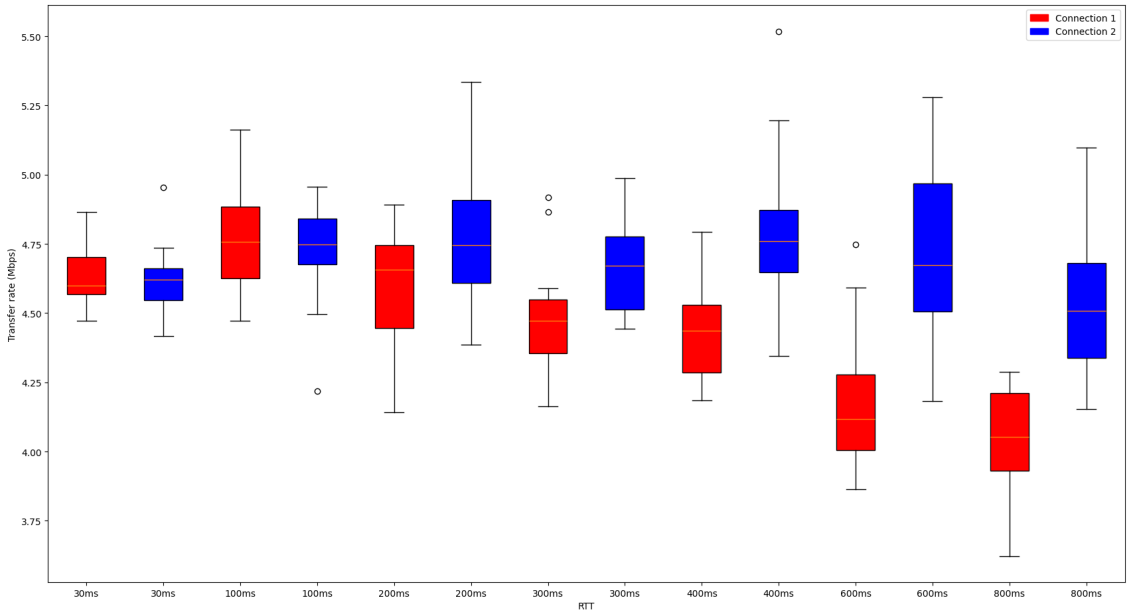

Figure 6.4: Transfer rates of different connections with and without Careful Resume with a 10MiB file

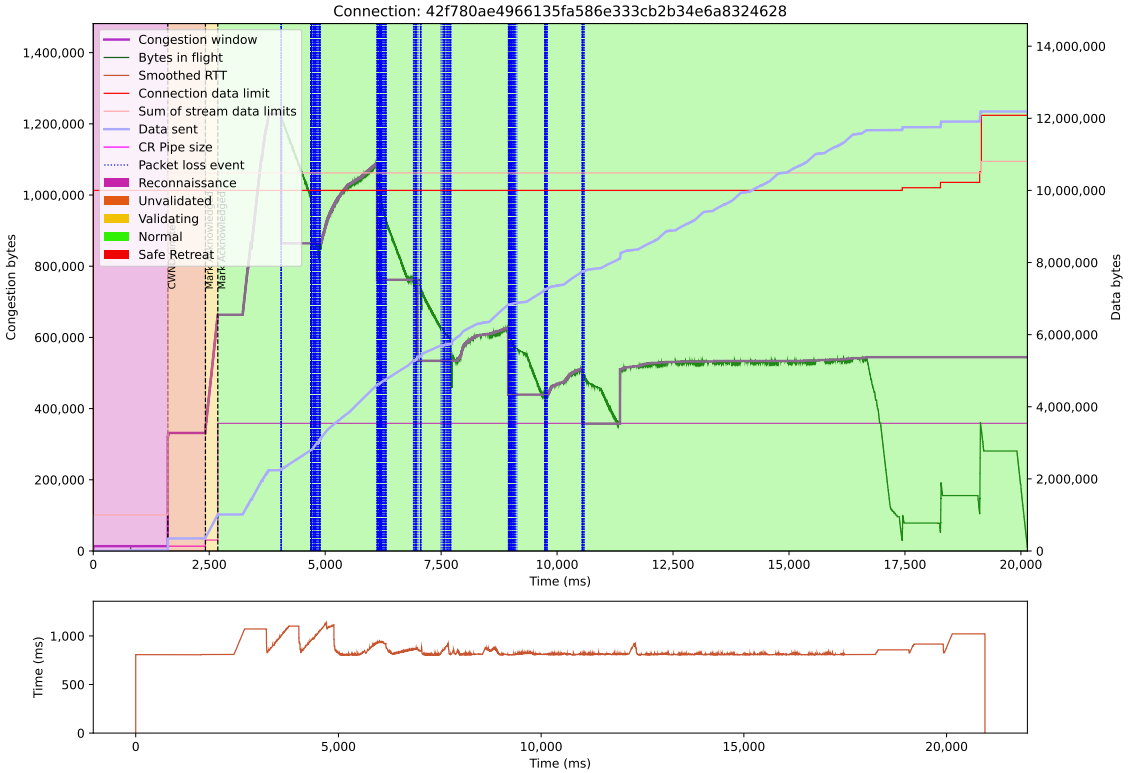

Figure 6.5: A QUIC connection downloading a 1MiB file over a 10Mbits link with 800ms of RTT, with Careful Resume

All tests so far have been under ideal conditions, with the same conditions in each connection. It is all well and good to say that Careful Resume and BDP Tokens work in ideal conditions but the Internet is not always ideal. It is also important to show that these modifications don't harm connections in less than

ideal conditions.

Figure 6.6 shows the data transfer rates of connections with and without Careful Resume on over a connection with 5% packet loss - indicative of severe congestion. Careful Resume no longer provides an appreciable benefit in transfer rates, but more importantly it does not harm the transfer rates or cause persistent congestion. It recognises the jump in the congestion window is ill-advised given the circumstances and proceeds as normal.

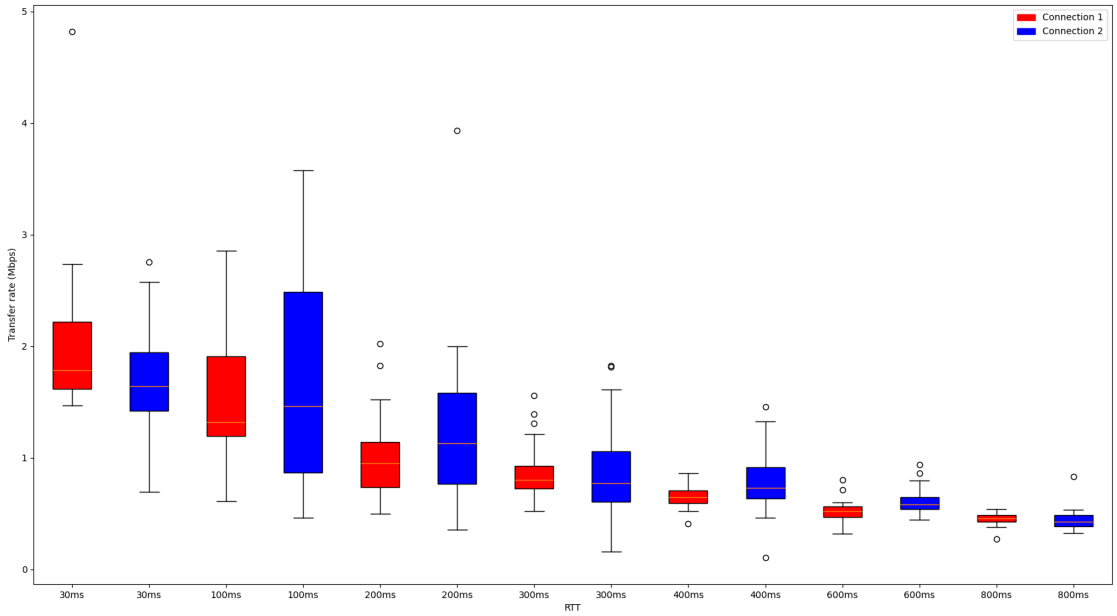

Figure 6.6: Transfer rates of different connections with and without Careful Resume with a 1MiB file, and 5% packet loss

Figure 6.7 shows the behaviour when the first connection has a bandwidth of 10Mbits but the second only has a bandwidth of 3Mbps. There was no difference in the RTT so Careful Resume was attempted. Initially the same large jump is made, but Careful Resume realises this was a mistake when packet loss happens. The Congestion Window is collapsed to what Careful Resume thinks is a safe bandwidth, and this is close to the value the congestion controller also settles in on. There is a large amount of packet loss at the start because of the overshoot, but very little after that.

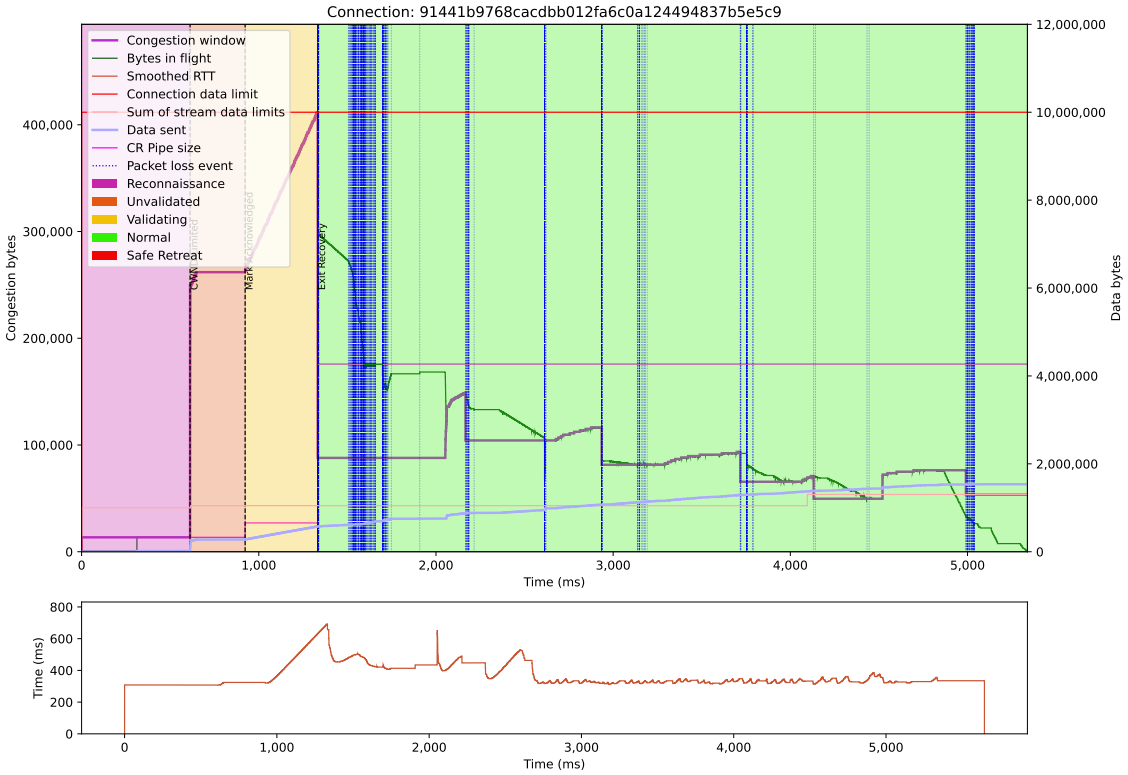

Figure 6.7: A QUIC connection downloading a 1MiB file over a 3Mbits link with 800ms of RTT, with Careful Resume assuming a 10Mbits link

Chapter 7

Discussion

This project has demonstrated the current issues in deploying QUIC over high BDP links, and the impacts this can have on real-world Internet applications. Careful Resume provides a safe way to increase data transfer rates on these connections, but is incomplete without signalling of when it should be used. The improvement is greatest on geostationary Satellite Internet connections, but on small transfers the impact can still be seen on low-earth orbit Satellite Internet connections.

The BDP Token presented here provides this signalling, and offloads the burden of data storage from the server/CDN to each client, in a way similar to HTTP cookies. The token is extendible to allow for future work in this space, and also backwards compatible, removing the chicken-and-egg problem common to new protocols. The token has an extremely low burden of additional data on the connection, and the implementation presented here further reduces its size by using AEAD encryption. Careful Resume with the BDP Token has been shown to work well, even in adverse circumstances, without reducing data transfer rates or being an issue to the safety of congestion control on the network.

The implementation created in Rust for this project builds on the robust foundations of Cloudflare's work on QUIC. It is a fully tested implementation conforming to all aspects of the specification, and is likely ready for deployment into a production environment.

7.1 Future work

Unfortunately, there was not enough time to explore all desired aspects. Little time was given to client interaction, and future work should investigate means to detect available bandwidth on the client side for this to be signalled to the server. The protocol presented in this work is ready for such data, as and when clients make it available.

Additional work could including adapting Careful Resume and BDP Tokens for use in the datacenter. Datacenters often have very high bandwidth links that may not be fully utilised by conventional congestion control methods. Inside a datacenter there is generally more trust of peers, and greater knowledge

of available link bandwidth. A token could be created and signed by the client inside the DC to give to a server in the same DC with exact details about the utilised link for the congestion controller.

There is scope to explore the deployment of Careful Resume on 5G and 6G mobile connections. Mobile connections are starting to have multi-gigabit throughputs, but still with the latency associated with a wireless interface. It is possible that Careful Resume could be used to aid in making full use of the very high speeds available on these connections.

This project did not adequately research the methods for deciding when to send new BDP tokens to the client (discussed in section 4.3). The implementation provided in this project is functional, but may not be ideal. Further work should be conducted to verify it, or to devise a better solution.

Bibliography

[1] N. Kuhn, S. Emile, G. Fairhurst, R. Secchi, and C. Huitema, "Convergence of Congestion Control from Retained State," Internet Engineering Task Force, Internet-Draft draft-ietf-tsvwg-careful-resume-07, Feb. 2024, Work in Progress, 31 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tsvwg-careful-resume/07/.

[2] L. Schumann, T. V. Doan, T. Shreedhar, R. Mok, and V. Bajpai, *Impact of Evolving Protocols and COVID-19 on Internet Traffic Shares*, 2022. arXiv: 2201.00142 [cs.NI].

[3] W. Eddy, *Transmission Control Protocol (TCP)*, RFC 9293, Aug. 2022. DOI: 10.17487/RFC9293. [Online]. Available: https://www.rfc-editor.org/info/rfc9293.

[4] L. Thomas, E. Dubois, N. Kuhn, and E. Lochin, "Google QUIC performance over a public SATCOM access," *International Journal of Satellite Communications and Networking*, vol. 37, no. 6, pp. 601–611, 2019. DOI: https://doi.org/10.1002/sat.1301. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sat.1301. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sat.1301.

[5] B. Trammell and M. Kühlewind, *The Wire Image of a Network Protocol*, RFC 8546, Apr. 2019. DOI: 10.17487/RFC8546. [Online]. Available: https://www.rfc-editor.org/info/rfc8546.

[6] M. Thomson, *Version-Independent Properties of QUIC*, RFC 8999, May 2021. DOI: 10.17487/RFC8999. [Online]. Available: https://www.rfc-editor.org/info/rfc8999.

[7] "Why use https?" Cloudflare, Inc. (Unknown), [Online]. Available: https://www.cloudflare.com/learning/ssl/why-use-https/.

[8] M. Thomson and S. Turner, *Using TLS to Secure QUIC*, RFC 9001, May 2021. DOI: 10.17487/RFC9001. [Online]. Available: https://www.rfc-editor.org/info/rfc9001.

[9] *NCP/TCP transition plan*, RFC 801, Nov. 1981. DOI: 10.17487/RFC0801. [Online]. Available: https://www.rfc-editor.org/info/rfc801.

[10] J. Iyengar and M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport*, RFC 9000, May 2021. DOI: 10.17487/RFC9000. [Online]. Available: https://www.rfc-editor.org/info/rfc9000.

[11] T. Pauly, E. Kinnear, and D. Schinazi, *An Unreliable Datagram Extension to QUIC*, RFC 9221, Mar. 2022. DOI: 10.17487/RFC9221. [Online]. Available: https://www.rfc-editor.org/info/rfc9221.

[12] E. Sy, C. Burkert, T. Mueller, H. Federrath, and M. Fischer, "Quicker connection establishment with out-of-band validation tokens," in *2019 IEEE 44th Conference on Local Computer Networks (LCN)*, 2019, pp. 105–108. DOI: 10.1109/LCN44214.2019.8990785.

[13] E. Sy, "Surfing the Web Quicker Than QUIC via a Shared Address Validation," in *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2019, pp. 1–6. DOI: 10.23919/SOFTCOM.2019.8903753.

[14] M. Bishop, *HTTP/3*, RFC 9114, Jun. 2022. DOI: 10.17487/RFC9114. [Online]. Available: https://www.rfc-editor.org/info/rfc9114.

[15] R. T. Fielding, M. Nottingham, and J. Reschke, *HTTP Semantics*, RFC 9110, Jun. 2022. DOI: 10.17487/RFC9110. [Online]. Available: https://www.rfc-editor.org/info/rfc9110.

[16] C. B. Krasic, M. Bishop, and A. Frindell, *QPACK: Field Compression for HTTP/3*, RFC 9204, Jun. 2022. DOI: 10.17487/RFC9204. [Online]. Available: https://www.rfc-editor.org/info/rfc9204.

[17] R. T. Braden, *Requirements for Internet Hosts - Communication Layers*, RFC 1122, Oct. 1989. DOI: 10.17487/RFC1122. [Online]. Available: https://www.rfc-editor.org/info/rfc1122.

[18] W. Eddy and M. Welzl, *Congestion Control in the RFC Series*, RFC 5783, Feb. 2010. DOI: 10.17487/RFC5783. [Online]. Available: https://www.rfc-editor.org/info/rfc5783.

[19] S. Floyd, D. K. K. Ramakrishnan, and D. L. Black, *The Addition of Explicit Congestion Notification (ECN) to IP*, RFC 3168, Sep. 2001. DOI: 10.17487/RFC3168. [Online]. Available: https://www.rfc-editor.org/info/rfc3168.

[20] S. Floyd, *Congestion Control Principles*, RFC 2914, Sep. 2000. DOI: 10.17487/RFC2914. [Online]. Available: https://www.rfc-editor.org/info/rfc2914.

[21] *Congestion Control in IP/TCP Internetworks*, RFC 896, Jan. 1984. DOI: 10.17487/RFC0896. [Online]. Available: https://www.rfc-editor.org/info/rfc896.

[22] *On Packet Switches With Infinite Storage*, RFC 970, Dec. 1985. DOI: 10.17487/RFC0970. [Online]. Available: https://www.rfc-editor.org/info/rfc970.

[23] L. Zhang, D. C. Partridge, S. Shenker, *et al.*, *Recommendations on Queue Management and Congestion Avoidance in the Internet*, RFC 2309, Apr. 1998. DOI: 10.17487/RFC2309. [Online]. Available: https://www.rfc-editor.org/info/rfc2309.

[24] *TCP extensions for long-delay paths*, RFC 1072, Oct. 1988. DOI: 10.17487/RFC1072. [Online]. Available: https://www.rfc-editor.org/info/rfc1072.

[25] E. Blanton, D. V. Paxson, and M. Allman, *TCP Congestion Control*, RFC 5681, Sep. 2009. DOI: 10.17487/RFC5681. [Online]. Available: https://www.rfc-editor.org/info/rfc5681.

[26] D. Katabi, M. Handley, and C. Rohrs, "Congestion Control for High Bandwidth-Delay Product Networks," in *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '02, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2002, pp. 89102, ISBN: 158113570X. DOI: 10.1145/633025.633035. [Online]. Available: https://doi.org/10.1145/633025.633035.

[27] T. Lakshman and U. Madhow, "The performance of TCP/IP for networks with high bandwidth-delay products and random loss," *IEEE/ACM Transactions on Networking*, vol. 5, no. 3, pp. 336–350, 1997. DOI: 10.1109/90.611099.

[28] D. C. Partridge, M. Allman, and S. Floyd, *Increasing TCP's Initial Window*, RFC 3390, Nov. 2002. DOI: 10.17487/RFC3390. [Online]. Available: https://www.rfc-editor.org/info/rfc3390.

[29] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis, *Increasing TCP's Initial Window*, RFC 6928, Apr. 2013. DOI: 10.17487/RFC6928. [Online]. Available: https://www.rfc-editor.org/info/rfc6928.

[30] J. Ruth and O. Hohlfeld, "Demystifying TCP Initial Window Configurations of Content Distribution Networks," in *2018 Network Traffic Measurement and Analysis Conference (TMA)*, IEEE, Jun. 2018. DOI: 10.23919/tma.2018.8506549. [Online]. Available: http://dx.doi.org/10.23919/TMA.2018.8506549.

[31] S. Weiler, D. Ward, and R. Housley, *The rsync URI Scheme*, RFC 5781, Feb. 2010. DOI: 10.17487/RFC5781. [Online]. Available: https://www.rfc-editor.org/info/rfc5781.

[32] Craig Partridge and D. Rockwell and Mark Allman and R. Krishnan and James Sterbenz, "A Swifter Start for TCP," BBN Technologies, 2002. [Online]. Available: https://www.icir.org/mallman/pubs/PRAKS02/PRAKS02.pdf.

[33] P. Balasubramanian, Y. Huang, and M. Olson, *HyStart++: Modified Slow Start for TCP*, RFC 9406, May 2023. DOI: 10.17487/RFC9406. [Online]. Available: https://www.rfc-editor.org/info/rfc9406.

[34] L. Pearce, *Spotlight: Direct Satellite-to-Device Mobile Services*, CCS Insight, 2023.

[35] S. Endres, J. Deutschmann, K.-S. Hielscher, and R. German, *Performance of QUIC Implementations Over Geostationary Satellite Links*, 2022. arXiv: 2202.08228 [cs.NI].

[36] M. Trevisan, "Fact checking starlinks performance figures," APNIC, 2022. [Online]. Available: https://blog.apnic.net/2022/11/28/fact-checking-starlinks-performance-figures/.

[37] D. J. D. Touch, *TCP Control Block Interdependence*, RFC 2140, Apr. 1997. DOI: 10.17487/RFC2140. [Online]. Available: https://www.rfc-editor.org/info/rfc2140.

[38] D. J. D. Touch, M. Welzl, and S. Islam, *TCP Control Block Interdependence*, RFC 9040, Jul. 2021. DOI: 10.17487/RFC9040. [Online]. Available: https://www.rfc-editor.org/info/rfc9040.

[39] L. Guo and J. Y. B. Lee, "Stateful-TCP A New Approach to Accelerate TCP Slow-Start," *IEEE Access*, vol. 8, pp. 195 955–195 970, 2020. DOI: 10.1109/ACCESS.2020.3034129.

[40] P. Mohapatra and R. Fernando, "BGP Link Bandwidth Extended Community," Internet Engineering Task Force, Internet-Draft draft-ietf-idr-link-bandwidth-07, Mar. 2018, Work in Progress, 5 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-idr-link-bandwidth/07/.

[41] sparky8512, *starlink-grpc-tools*, 2023. [Online]. Available: https://github.com/sparky8512/starlink-grpc-tools.

[42] R. Jung, "Understanding and evolving the rust programming language," 2020.

[43] K. Bierhoff, "Iterator specification with typestates," in *Proceedings of the 2006 conference on Specification and verification of component-based systems*, 2006, pp. 79–82.

[44] B. Stroustrup, *The design and evolution of C++*. Pearson Education India, 1994.

[45] L. Pardue. "QUIC Version 1 is live on Cloudflare," Cloudflare, Inc. (2021), [Online]. Available: https://blog.cloudflare.com/quic-version-1-is-live-on-cloudflare.

[46] R. Marx, L. Niccolini, M. Seemann, and L. Pardue, "Main logging schema for qlog," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-qlog-main-schema-08, Mar. 2024, Work in Progress, 44 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/08/.

[47] R. Marx, L. Niccolini, M. Seemann, and L. Pardue, "QUIC event definitions for qlog," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-qlog-quic-events-07, Mar. 2024, Work in Progress, 60 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-quic-events/07/.

[48] R. Marx, L. Niccolini, M. Seemann, and L. Pardue, "HTTP/3 qlog event definitions," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-qlog-h3-events-07, Mar. 2024, Work in Progress, 22 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-h3-events/07/.

[49] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446, Aug. 2018. DOI: 10.17487/RFC8446. [Online]. Available: https://www.rfc-editor.org/info/rfc8446.

[50] D. Peck. "SSL added and removed here ;-)." (2013), [Online]. Available: https://blog.encrypt.me/2013/11/05/ssl-added-and-removed-here-nsa-smiley/.

[51] "BoringSSL Headers," Google, LLC. (2024), [Online]. Available: https://commondatastorage.googleapis.com/chromium-boringssl-docs/ssl.h.html.

[52] D. H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, RFC 2104, Feb. 1997. DOI: 10.17487/RFC2104. [Online]. Available: https://www.rfc-editor.org/info/rfc2104.

[53] F. Gont, S. Krishnan, D. T. Narten, and R. P. Draves, *Temporary Address Extensions for Stateless Address Autoconfiguration in IPv6*, RFC 8981, Feb. 2021. DOI: 10.17487/RFC8981. [Online]. Available: https://www.rfc-editor.org/info/rfc8981.

[54] Z. Sarker, V. Singh, X. Zhu, and M. A. Ramalho, *Test Cases for Evaluating Congestion Control for Interactive Real-Time Media*, RFC 8867, Jan. 2021. DOI: 10.17487/RFC8867. [Online]. Available: https://www.rfc-editor.org/info/rfc8867.

[55] "Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats," International Organization for Standardization, Geneva, CH, Standard, Aug. 2022.

[56] C. Newman and G. Klyne, *Date and Time on the Internet: Timestamps*, RFC 3339, Jul. 2002. DOI: 10.17487/RFC3339. [Online]. Available: https://www.rfc-editor.org/info/rfc3339.

[57] H. Birkholz, C. Vigano, and C. Bormann, *Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures*, RFC 8610, Jun.

2019. DOI: 10.17487/RFC8610. [Online]. Available: https://www.rfc-editor.org/info/rfc8610.

[58] C. Bormann and P. E. Hoffman, *Concise Binary Object Representation (CBOR)*, RFC 7049, Oct. 2013. DOI: 10.17487/RFC7049. [Online]. Available: https://www.rfc-editor.org/info/rfc7049.

[59] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 8259, Dec. 2017. DOI: 10.17487/RFC8259. [Online]. Available: https://www.rfc-editor.org/info/rfc8259.

[60] A. Chanda, *Network Programming with Rust: Build fast and resilient network servers and clients by leveraging Rust's memory-safety and concurrency features*. Packt Publishing Ltd, 2018.

[61] W. de Bruijn and E. Dumazet, "Optimizing UDP for content delivery: GSO, pacing and zerocopy," in *Linux Plumbers Conference*, 2018.

[62] Y. Nir and A. Langley, *ChaCha20 and Poly1305 for IETF Protocols*, RFC 8439, Jun. 2018. DOI: 10.17487/RFC8439. [Online]. Available: https://www.rfc-editor.org/info/rfc8439.

[63] J. Turnbull, *The art of monitoring*. James Turnbull, 2014, ISBN: 978-0-9888202-4-1.

[64] A. Jurgelionis, J.-P. Laulajainen, M. Hirvonen, and A. I. Wang, "An Empirical Study of NetEm Network Emulation Functionalities," in *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)*, 2011, pp. 1–6. DOI: 10.1109/ICCCN.2011.6005933.

Appendix A

Internet-Draft: draft-misell-quic-ex-token

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 5 September 2024

Q. Misell
AS207960

4 March 2024

Extensible Address Validation Tokens for QUIC
draft-misell-quic-ex-token-latest

Abstract

This document describes a method to extend QUIC address validation tokens to include structured data that a client can parse and make use of. The initial application envisioned in this document is signalling congestion control parameters for use with Careful Resume.

Table of Contents

1. Introduction

This document defines a method to extend QUIC address validation tokens to have structure, so that a client can parse them, and optionally make use of additional information within the token, depending on the nature of the extensions in use.

The initial application for this is envisioned to be allowing a QUIC server to send calculated Congestion Control parameters to a client for storage and later use on a future connection with Careful Resume [I-D.ietf-tsvwg-careful-resume].

1.1. Requirements Language

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, NOT RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in [BCP14] when, and only when, they appear in all capitals, as shown here.

2. Extensible Tokens

QUIC [RFC9000] defines an address validation token as an opaque blob that the client should not inspect. This document extends this by providing structure to the token, allowing arbitrary fields to be defined on the token by this and future documents.

A QUIC client unaware of this document will still be able to make use of extensible tokens without modification, although it will do nothing but pass the entire token back to the server unmodified.

The format of an extensible token is defined as follows:

```
BDP Token {
    Address Validation Length (i),
    Address Validation (..),
    Extension ID (i),
    Extension Data Length (i),
    Extension Data (..),
    ...
}
```

Figure 1: Extensible Token

The fields are as follows:

Address Validation Length A variable-length integer specifying the length of the Address Validation field, in bytes. A value of 0 indicates that the server is not using address validation and is using the token purely for extensible information.

Address Validation Opaque information specific to the server that it will use for address validation. The construction of this field MUST comply with the requirements of [RFC9000] section 8.1.

Extension ID A variable-length integer specifying the ID of the extension. The registry for these IDs is defined in the IANA Considerations section of this document.

Extension Data Length A variable-length integer specifying the length of the Extension Data field, in bytes.

Extension Data A byte field containing the data for the extension.

Extension ID, Extension Data Length, and Extension Data can be repeated as many times as needed to include desired extensions. Extension IDs MUST NOT appear multiple times in a token.

2.1. Extensible token signalling

The server can send extended tokens to all clients without further negotiation. However a client needs some way to know that there is meaningful structure to a token its received from the server. To this end a new transport parameter is defined.

ex_token (04143414213370002) The ex_token transport parameter is a boolean value that indicates that the server is using extensible tokens. It can have the following values:
- 0, default value: Extensible Tokens are not in use.
- 1: Extensible Tokens are in use.

2.2. Invalid Extensible tokens

If the server is unable to decode the token received from the client, or vise versa, this MUST be treated as the connection error EX_TOKEN_ERROR (0x4143414213370002).

2.3. Extension design

As an extensible token is designed to be presented to clients that may not be aware of what an extensible token is, all extensions defined for extensible tokens MUST be designed such that it can handle its data being echoed back to the server unmodified on a future connection.

3. BDP Tokens

This document defines the BDP_TOKEN extension, designed to Congestion Control information for use in Careful Resume [I-D.ietf-tsvwg-careful-resume].

The format of a token extension containing BDP information is defined as follows:

```
BDP Token {
    Private Data Length (i),
    Private Data (..),
    Capacity (i),
    RTT (i),
    Requested Capacity (i),
}
```

Figure 2: BDP Token

The fields are as follows:

Private Data Length A variable-length integer specifying the length of the Private Data field, in bytes. A value of 0 indicates that the server does not intend to store Congestion Control data on the client, and that the Capacity and RTT fields are merely informative.

Private Data Opaque information specific to the server that it will use to prime its congestion controller state. This field SHOULD contain expiration/lifetime information, and any additional information that the server may need to validate that the same path is being used, such as an endpoint token as defined in Careful Resume [I-D.ietf-tsvwg-careful-resume]. The data MUST be

signed, or otherwise protected against modification by the client. If the data contains information that could potentially be used to fingerprint the client, it MUST be encrypted, or otherwise protected against inspection by wire observers. An example method for protecting the Private Data field is provided in Appendix A.

Capacity The estimated capacity of the path in bytes (congestion window), encoded as a variable-length integer.

RTT The RTT of the path in microseconds, encoded as a variable-length integer.

Requested capacity In a token sent by the server, this field is set to the same value as the Capacity field. If the client becomes aware of a change in the available bandwidth of the path, it can adjust this field to request a lower capacity be used by the server when priming its congestion controller state.

3.1. Client interaction

If the client becomes aware of a change in the available bandwidth of the path, it can use the Requested Capacity field to signal to the server a change it its available bandwidth. The server MUST not accept a value higher than that of the Capacity field, as this could cause an overload of the network path.

If the client sets the Requested Bandwidth field to 0 then it is signalling that the server should not attempt to prime its congestion controller from previous state and should instead treat this connection as an entirely new congestion control context.

3.2. Invalid BDP tokens

If a server or client is unable to decode the token received as a valid BDP token then this MUST be treated as a connection error BDP_TOKEN_ERROR (0x4143414213370003). A token which is merely expired MUST NOT trigger a connection error, instead it should be silently discarded.

4. IANA Considerations

4.1. QUIC Transport Parameters

Per this document, one new entry has been added to the "QUIC Transport Parameters" registry defined in [RFC9000] section 22.3. This entry is defined below:

Value	Status	Specification	Parameter name
0x4143414213370002	Provisional	This document	ex_token

Table 1: New Transport Parameter entries

4.2. QUIC Transport Error Codes

Per this document, two new entries have been added to the "QUIC Transport Error Codes" registry defined in [RFC9000] section 22.5. This entry is defined below:

Value	Status	Code	Description	Specification
0x4143414213370002	Provisional	EX_TOKEN_ERROR	The extensible	This document

			token received is invalid.		
0x41434142 13370003	Provisional	BDP_TOKEN_ERROR	The BDP token received is invalid.	This document	

Table 2: New Transport Error Code entries

4.3. QUIC Extensible Token Extension IDs

Per this document, a new registry for "QUIC Extensible Token Extension IDs" is created under the "QUIC" heading defined in [RFC9000] section 22.

In addition to the fields listed in [RFC9000] section 22.1.1, permanent registrations in this registry MUST include the following field:

Extension name A short mnemonic for the extension type.

Value	Extension name	Specification
0x01	BDP_TOKEN	Section 3

Table 3: Initial QUIC Extensible Token Extension IDs Entries

Each value of the form 31 * N + 26 for integer values of N (that is, 26, 57, 88, ...) are reserved for private use; these values are to be treated as opaque blobs meaningful only to the server operator.

Each value of the form 31 * N + 27 for integer values of N (that is, 27, 58, 89, ...) are reserved; these values MUST NOT be assigned by IANA and MUST NOT appear in the listing of assigned values.

5. Security Considerations

The Congestion Control data MUST be protected against manipulation by malicious or mis-behaving clients. A client that can modify Congestion Control data could cause an overload of the network path.

6. References

6.1. Normative References

[BCP14] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017.

<https://www.rfc-editor.org/info/bcp14>

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

6.2. Informative References

[I-D.ietf-tsvwg-careful-resume]

Kuhn, N., Emile, S., Fairhurst, G., Secchi, R., and C. Huitema, "Convergence of Congestion Control from Retained State", Work in Progress, Internet-Draft, draft-ietf-tsvwg-careful-resume-07, 16 February 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-tsvwg-careful-resume-07>.

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/info/rfc8439>.

Appendix A. Example method for protecting the BDP Token's Private Data Field

This section is non-normative.

Consider the following private data structure:

```
Private Data {
    IP Address (16 bytes),
    Expiry Timestamp (8 bytes)
}
```

Figure 3: BDP Private Data

This structure contains information that could be used to track a client (its previous IP address), and therefore MUST be protected from inspection by wire observers. Additionally the Capacity and RTT fields MUST be protected from modification by a client.

To this end this example uses the ChaCha20-Poly1305 [RFC8439] cipher with additional data. The ChaCha20-Poly1305 cipher encrypts data, and protects the encrypted data as well as additional unencrypted data against modification.

The additional data has the following structure:

```
Additional Data {
    Saved Capacity (8 bytes),
    Saved RTT (16 bytes)
}
```

Figure 4: BDP Additional Data

After encryption the 12 byte nonce is appended to the Private Data structure to allow the encryption to be reversed when the token is later used.

Author's Address

Q Misell
AS207960 Cyfyngedig
13 Pen-y-lan Terrace
Caerdydd
CF23 9EU
United Kingdom
Email: q@as207970.net, q@magicalcodewit.ch
URI: https://magicalcodewit.ch

Appendix B

User Manual, Maintenance Manual and Code Listing

Due to the nature of this project the traditional User Manual, Maintenance Manual and Code Listing is not suitable. A theory of operations for the software is provided in the design and implementation sections.

The `README.md` file within the code submission provides guidance on which files are of interest and the general structure of the submitted code. The code is not intended to be used by an end user as this was a research project, and as such a User Manual is not provided. Even if it were to be used by an end user it would be as a library in a larger programme such as a web browser, not as a standalone piece of software. The user would be oblivious to the use of this project's code.

The changes to Cloudflare's Quiche may eventually be upstreamed, however the rest of the code serves mostly to facilitate testing of hypotheses. This is not code that is intended to be maintained or worked upon in future. For the code that may be upstreamed to Cloudflare, comments are provided throughout to document function interfaces and theories of operation where code is not self-explanatory.

The code listing of the totality of the code needed for this project is nearly 1,500 pages long and almost entirely useless. It is provided only because the process requires it. The folder structure of the submitted code is a far better format to separate what is important code, and what is merely ancillary or lightly modified code.