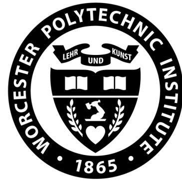
Improving QUIC Slow Start Behavior in Satellite Networks with SEARCH



WPI

An Undergraduate Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the Degree of Bachelor of Science in

Computer Science,
Electrical and Computer Engineering

By:

Amber Cronin

Project Advisors:

Mark Claypool
Alexander Wyglinski

Sponsored By:

Viasat, Inc.

Date: April 2024

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review. For more information about the projects program at WPI, see **https://www.wpi.edu/project-based-learning**.

Abstract

QUIC is increasingly being deployed on the Internet as an alternative to TCP. However, QUIC over satellite links faces particular challenges as high and variable round-trip times (RTTs) make it difficult to determine and then reach link capacity. Standard TCP slow start algorithms to detect link capacity can perform poorly over satellite links, often exiting slow start too early and limiting throughput or exiting too late and causing unnecessary packet loss. Slow start Exit At Right CHokepoint (SEARCH) aims to exit slow start after reaching link capacity but before incurring packet loss by tracking delivery rates and exiting when rates have not increased by the expected amount. SEARCH has shown benefits over traditional slow start for TCP connections but has yet to be implemented and evaluated in QUIC. This paper presents the design and implementation of SEARCH in an open-source QUIC library, with the code publicly available as a contribution. Evaluation of SEARCH over a geostationary satellite link show SEARCH successfully exits slow start before loss in the majority of cases, improving goodput compared to the baseline and improving download times by up to 3 seconds in the median case.

Acknowledgements

I would like to thank Professor Claypool and Professor Wyglinski for their continued support and advice during this project; they truly helped make this project a success. Our conversations were invaluable and I am thankful for my time spent with them. I would also like to thank Maryam Ataei Kachooei for her guidance when first implementing SEARCH, and her willingness to take the time to assist with various components of this project. Dr. Jae Chung, Dr. Feng Li, and Benjamin Peters provided important contributions and comments on the usage of the testbed, experimental design, and algorithm implementation, and I thank them all for their help.

Finally, I would like to thank my friends and parents for their unwavering support. Without them, I doubt I would have the confidence to have achieved this milestone.

Table of Contents

1 **Introduction** **1**

2 **Background** **3**

2.1 Satellite Internet Links . . . 3

2.1.1 Characteristics of GEO Internet Satellites . . . 3

2.1.2 Characteristics of LEO Internet Satellites . . . 3

2.1.3 Satellite Link Properties . . . 4

2.1.4 Performance Enhancing Proxies . . . 4

2.2 The QUIC Protocol . . . 5

2.3 QUIC vs. TCP . . . 6

2.4 Overview of Congestion Control . . . 6

2.5 Congestion Control Startup (Slow Start) . . . 8

2.6 HyStart . . . 9

3 **Related Works** **11**

3.1 Congestion Control Algorithms . . . 11

3.2 Performance Enhancing Proxies . . . 11

3.3 Slow Start Mechanisms . . . 12

3.4 QUIC Performance Evaluation . . . 12

4 **Methodology** **14**

4.1 Base QUIC Implementation . . . 14

4.2 Testbed Configuration . . . 14

4.3 Modular Slow Start . . . 15

4.4 Implementation of SEARCH . . . 16

4.5 Emulated Network Tests . . . 19

4.6 QUIC Max Data and Max Window . . . 20

4.7 Server Test Settings and Procedures . . . 20

4.8 Qperf Modifications . . . 22

4.9 End Goal Test Justification . . . 22

4.10 Experimental Design . . . 22

5 **Results** **24**

5.1 First-RTT and Wireless Loss . . . 24

5.2 Dataset Description . . . 24

5.3 Dataset Analysis . . . 25

6 Discussion **30**

6.1 SEARCH Performance . . . 30

6.2 Modular Slow Start . . . 31

6.3 First-RTT Losses . . . 31

6.4 Quicly Performance . . . 31

7 Future Work **33**

7.1 Congestion Control . . . 33

7.1.1 Slow Start . . . 33

7.1.2 Variable Congestion Window Growth . . . 34

7.2 New Quicly Features . . . 34

7.2.1 Pacing . . . 34

7.2.2 Jump Start . . . 34

7.3 Testing Procedures . . . 35

8 Conclusion **36**

References **37**

A SEARCH Code Listing **40**

List of Figures

1 Slow start exits: a) early exit, b) overshoot, c) at chokepoint. . . . 9

2 GEO Satellite Testbed 15

3 Delivered bytes vs. expected delivered bytes per RTT 17

4 General overview of the emulation testbed 19

5 Onion model of Quicly's implementation of sender limits. 21

6 Transmission rate and RTTs of a CUBIC connection in the emulated link, with default Quicly settings. 21

7 Transmission rate and RTTs of a CUBIC connection in the emulated link, with large maximum window growth. 21

8 Ratio of time between SEARCH declared exit and loss occurring, CDF of cleaned runs. Green dashed vertical line indicates exit one RTT before loss. 26

9 Scatterplot displaying behaviour of all runs in Dataset 1. 26

10 Median time to download [N] MB, shading depicts 25% and 75% quartiles 27

11 Difference in median run timing, shading depicts absolute difference between quartiles . 27

12 CDFs of the time taken to deliver [N] bytes, with surviving runs labeled in subtitles. . . 28

13 Ratio of the reduction of the CWND at SEARCH exit point to the original CWND value, SEARCH computation vs CUBIC β multiplicative decrease 29

List of Tables

1 Dataset 1 makeup, no CWND modification performed. 24

2 Dataset 2 makeup, CWND modification performed. 25

3 Baselines dataset makeup, composed of non-SEARCH exit runs from Dataset 1 and Dataset 2. 25

4 Count of runs delivering [N] MB, Dataset 1. 25

1 Introduction

QUIC is a transport-layer protocol, developed by Google since 2012 and standardized by IETF in 2021 [1]. QUIC implements transport control in userspace and uses UDP packets for communication between hosts. Further, QUIC encrypts all communications on the link as a fundamental part of the protocol, performing this setup in the first round trip. QUIC utilizes multiple streams over a single connection to reduce head-of-line blocking after loss, improving performance for use in applications like HTTP/3. QUIC has gained popularity since its public release, carrying nearly 30% of web traffic as of 2023 [2].

Satellites in Geostationary Equatorial Orbit (GEO) have a baseline round-trip time (RTT) of about 600 milliseconds (ms) and high bandwidths in the hundreds of megabits per second (Mb/s). These ”long, fat” links have bandwidth delay products (BDP) in the megabytes, compared to kilobytes for a typical low/medium latency terrestrial connection. TCP has traditionally faced challenges saturating connections with these properties due to the nature of TCP probing mechanisms [3].

A typical solution for increasing the throughput of TCP over these links is to implement a performance enhancing proxy (PEP) using middleboxes that break a TCP connection into two or more parts over the terrestrial and orbital sections, known as Split TCP [4]. This allows hosts on Earth to see low-latency responses, while the orbital transport can be implemented with a high-RTT optimized congestion control algorithm such as TCP Hybla [5]. Due to the encrypted nature of QUIC with which endpoints are cryptographically verified on connection, this solution is no longer feasible, forcing the protocol's own congestion controller to respond to the characteristics of the link. QUIC shares many similarities in its implementation of congestion control with TCP, and standard implementations include IETF-defined Reno, and many provide CUBIC among others.

One of the fundamental congestion control phases that can be improved over GEO satellite links is the slow start phase. Slow start implements the general bandwidth-seeking phase of the connection, during which the congestion controllers typically double the amount of data being pushed over the link every RTT until a packet loss occurs. On high-BDP links, the high RTT and the amount of data in-flight (sent, but not acknowledged) impacts the recovery time when packet losses do occur. Large queues on satellite groundstations, and bufferbloat [6] throughout the rest of the Internet exacerbate the BDP issue by not dropping packets immediately when link capacity is reached, increasing the penalty of slow start overshoot. To address this issue, we implement a new slow start algorithm called Slow start Exit At Right CHokepoint (SEARCH) [7] to limit the overshoot and exit the slow start phase and enter the congestion avoidance phase before loss occurs. While SEARCH has been implemented and evaluated in TCP, it has not been incorporated into QUIC. This project does so for

the open-source QUIC library, Quicly[1]. We contribute this code and evaluate our implementation over a GEO satellite link.

The organization of the rest of this paper is as follows: Section 2 discusses general topics of relevance to this paper in more depth, as a more substantial reference for the material needed to understand this work. Section 3 discusses prior work of interest to this paper. Section 4 discusses the implementation of SEARCH in QUIC, the presentation of a new method of abstracting slow start behaviour, and our experimental design. Section 5 discusses experimental results and takeaways. Section 6 discusses the implications of the presented results and reasoning behind behaviour observed. Section 7 discusses future work including changes to testbed, algorithm, and alternate methods to explore. Section 8 summarizes the presented work.

[1] `https://github.com/h2o/quicly`

2 Background

In this background, we discuss the technologies that form the basis for this project and provide an overview of the problem space. We include an overview of satellite networking technologies, TCP, QUIC, and congestion control theory.

2.1 Satellite Internet Links

Satellite links are one of many forms of wireless links that exist on the modern Internet and have unique properties that impact the performance of the protocols operating over them. Satellite internet links are important primary connection sources for rural Internet users, disaster response, military communications, ships, and secondary Internet sources for terrestrial users in more settled areas. Internet satellites exist in two primary locations above Earth: Geostationary Earth Orbit (GEO) and Low Earth Orbit (LEO).

2.1.1 Characteristics of GEO Internet Satellites

Satellites located in Geostationary Orbit (GEO) are located in an equatorial orbit approximately 35,786km above the Earth's surface, such that their orbital period is the length of one day and they appear fixed over a particular position on Earth [8]. Because of these properties, GEO satellites have a fixed position when viewed from the ground, and antennas to communicate with these satellites may be mono-directional in a fixed position. Due to the constant speed of light, signals transmitted from Earth to the GEO satellite or from the GEO satellite to Earth arrive approximately 120ms later. This constant requirement limits the minimum round-trip time (RTT) of an Internet link containing a GEO satellite link to 480ms, though in practice, this compounds with the additional latency of the link to form a minimum latency of around 600ms.

2.1.2 Characteristics of LEO Internet Satellites

The LEO zone does not have a fixed altitude, but refers to a zone between the edge of the Earth's atmosphere (100km) and Medium Earth Orbit (MEO, 2000km) [9]. Satellites located in LEO orbit the Earth faster than it rotates, and appear moving when viewed from the ground. LEO satellites do not have line-of-sight (required for high-frequency radio communication) to wide areas of the Earth's surface. Users intending to communicate with these satellites require antennas that can track individual satellite's movement as they pass over for high bandwidth applications, or do not require directional connection for low bandwidth applications like satellite phone. An Internet link containing a LEO satellite link has a minimum RTT of approximately 60ms, depending on the exact altitude of the

satellite constellation.

2.1.3 Satellite Link Properties

Satellite links are subject to a number of unique properties not experienced by wired or fiber links. When an endpoint initiates a connection with the satellite, a reservation must be made for some amount of bandwidth for use by the endpoint. This negotiation takes an amount of time not typically experienced by wired or other wireless links. As the data rate of the connection increases to the reserved bandwidth, the endpoint must request additional bandwidth (to the contracted service bandwidth) to support the additional data. To handle this behaviour, satellite links often implement larger-than-normal buffers on their ground stations to account for the allocation time. Over the course of a connection ramp-up, this creates oscillation of the measured RTTs as the higher data rate builds a queue during the allocation period, drains the queue as the connection continues ramping, and repeats until the link is fully allocated and a packet loss occurs.

As with all links, there is the possibility of connection errors causing packet loss prior to saturation. Communication speed is dependent on the signal-to-noise ratio (SNR), the supported physical layer (PHY) protocols, and physical obstructions including weather events and proper endpoint antenna orientation. These connection errors may be mitigated/addressed by link layer protocols for re-transmission on wireless errors (if supported), as performed by WiFi access points (802.11), but this is not guaranteed nor required.

2.1.4 Performance Enhancing Proxies

One of the most commmonly implemented solutions to mitigate the non-ideal behaviour of GEO satellite links are performance enhancing proxies (PEPs) [4]. PEPs perform a man-in-the-middle (MITM) attack on a TCP connection by programming the middleboxes on either end of the satellite connection to reply with faked TCP SYN+ACK and ACK packets. This behaviour is possible because TCP clients do not verify the endpoint they are connecting to cryptographically as QUIC does, so these middlebox responses appear to the client as if the server is located in terrestrial/LAN range while the true request is being forwarded on behalf of the middlebox to the final destination. The middlebox is unable to provide the true response from the server (unless it also implements caching), and as such the true latency of the connection is still limited to the physical behaviour of the underlying link. However, the benefit provided by implementing this behaviour comes from the speedup to the initial startup of the connection, as the fast SYN+ACK and ACK responses help grow the client's congestion window while the actual responses from the server are still in flight. QUIC's cryptographic behaviour sets up TLS encryption keys in the first salvo of packets, which verify that the responding endpoint

has control over the domain that it is responding as. This breaks the assumption that the PEP relies on, but does not remove other PEP-like constructions.

2.2 The QUIC Protocol

QUIC is a relatively new transport-layer protocol that provides some benefits over TCP. QUIC was initially developed for use on the web by Google in 2012, and was rolled out over time for use by Google services, with IETF standardization of the protocol completing in 2021 [10] [1]. QUIC is the underlying protocol supporting HTTP/3, a new version of HTTP designed specifically for operation with QUIC in order to take advantage of the features listed above for improving website load times and Internet performance. HTTP/3 (and QUIC) have seen a large growth on the web since the initial release of the protocol by Google. This growth has increased since the protocol was formalized by IETF, with Cloudflare estimating that the protocol handles nearly 30% of all web traffic [2].

QUIC operates at the application layer and transmits data using UDP packets rather than being implemented in the kernel. Due to the application-layer design, updates to the protocol are not limited to the release rate of operating system updates, and the protocol itself can carry out version negotiation between client and server to select a version of QUIC with shared support. QUIC connections are entirely encrypted after the initial handshake completes, including all plaintext-necessary header fields. Each QUIC packet contains one or more frames, and packet numbers are guaranteed to be unique over the course of the connection. QUIC provides the ability to create multiple non-blocking streams to mitigate head-of-line blocking in the connection caused by packet loss and data retransmission. Frames contained within a packet are each attributed to a stream, and frames from multiple streams may be included within a single packet. If a packet is lost in transit, only the streams which had data contained within the lost packet become blocked during the recovery periods triggered by the loss, and the QUIC controller will continue transmitting the data of other streams while managing the recovery of the streams that lost data. QUIC performs per-stream acking of received frames using a method analogous to TCP's SACK extension, with each ack frame containing a maximum sequence number received, and optional sub-fields listing the start and end of unacknowledged gaps in the sequence space. QUIC supports 1-RTT and 0-RTT data transmission, in which application data may be included in the initial or first round of packets to the server, before the connection has been fully established provided the client has cached cryptographic information from the server. Each QUIC connection has an associated Connection ID, which removes the assumption of the TCP four-tuple (source IP, destination IP, source port, destination port) and allows for more seamless routing from client to server after IP address or port changes due to middlebox behaviour or mobility.

2.3 QUIC vs. TCP

QUIC aims to address issues with TCP, namely protocol ossification due to unspecified/incorrect middlebox behaviour, head of line blocking, and lack of encryption by default. By encrypting many of the QUIC packet headers within its UDP carrier, middleboxes cannot inspect QUIC connection data as they can with TCP, and therefore cannot lock in a particular supported version. QUIC's implementation in userspace allows for fast iteration of the protocol, seen especially in early development during the testing phase by Google. While this is a net positive for end-to-end user privacy by default, with even the handshake packet encrypted, it makes it difficult for network administrators to implement traffic shaping and gather data about network usage as they could with TCP. One such case where traffic shaping helps improve TCP performance is by performing ack thinning techniques to drop redundant acks for a connection that provide little value and claw back bandwidth for use by other users. This is infeasible with QUIC, as the network administrator would have to either a) have the decryption key for the connection to determine which packets contain droppable acks, or b) drop "ack-likely" packets at random, degrading the connection and introducing the possibility of destroying non-ack frames contained within the packet.

QUIC finds some difficulty in the modern web due to issues with blocking/throttling of the UDP packets on which QUIC relies by network operators [10]. This behaviour presents as limiting the maximum size of UDP packets to smaller than path MTU, droping UDP packets more often than TCP packets, or failing to queue UDP packets alongside TCP packets due to assumptions about the traffic being carried. Often, applications that support QUIC will attempt to open a QUIC and TCP flows in parallel, with the intention of using QUIC but maintaining the TCP flow as a backup in networks where UDP packets are blocked or otherwise handled poorly.

2.4 Overview of Congestion Control

In the development of congestion control algorithms, two main behaviours exist. The most common, as proposed when the Internet was created, is loss-based congestion control. Algorithms following this model typically watch for loss as a primary signal of congestion, and lower sending rates when loss is observed on the connection. Modern proposals have been tested and deployed that are considered throughput/rate-based congestion control, which may take packet loss as a signal of congestion but do not necessarily lower their delivery rate in response to a loss.

The QUIC specification states that by default, the Reno congestion control algorithm should be used [1]. This algorithm is robust and simple, but performs worse than modern Internet congestion control algorithms due to demonstrably low throughput on high-bandwidth bottleneck links. Reno introduces the concept of additive increase/multiplicative decrease (AIMD) for link sharing, and im-

plements this concept after the slow start phase. Reno increases the congestion window by a set number of packets every RTT, to form linear growth, and lowers the congestion window by half when a loss is observed. However, due to its failure to saturate higher bandwidth links effectively, alternative congestion control algorithms are more common on the modern web.

The default congestion control used by most Linux distributions is CUBIC, proposed to address the issues with Reno failing to saturate high-bandwidth networks [11]. CUBIC probes for bandwidth through non-linear congestion window growth, defined by a cubic function between a minimum and maximum congestion window as defined by slow start loss and previous growth/loss cycles. CUBIC lowers the congestion window by a factor of 0.7 (compared to Reno's 0.5) on loss, allowing faster recovery. CUBIC implements a growth phase CUBIC's more aggressive behaviour allows better utilization of more lossy links, fairness with other CUBIC flows, and better competition with other congestion control algorithms.

To compensate for the difficult behaviours of many satellite links (and other lossy/variable RTT networks including 4G/5G and WiFi), other congestion control algorithms have been developed, though they are not widely deployed. Many of these algorithms require stronger constraints or additions to the underlying protocols to transmit further information, typically infeasible in most networks. For satellites, the most notable alternative congestion control algorithm is Hybla [5]. Hybla modifies both the slow start and congestion avoidance phases of the congestion controller to scale congestion window growth based on the ratio of the base link RTT to a minimum RTT of 25ms. By scaling the congestion window growth in both slow start and congestion avoidance based on this ratio, Hybla is capable of matching the growth of the congestion window seen on terrestrial networks in high BDP links.

Finally, there are alternative methods of implementing congestion control that have begun research in recent years. The main proponent of these alternative methods is Google, who has released several versions of BBR [12]. BBR shares a similar startup phase to seek link capacity by performing similar exponential doubling, but does not take loss as an explicit notification of congestion. Rather, BBR attempts to maintain low RTTs and high throughputs without regard for loss by attempting to minimize queue usage on the path. BBR has been shown to be effective in maximizing link usage, but initial versions failed to share bandwidth with existing CUBIC flows effectively due to using alternative methods of monitoring link capacity. Research on this congestion control algorithm is ongoing, and it is provided in some implementations of QUIC.

The open-source project used in this paper, Quicly, defaults to Reno but provides CUBIC and Pico (a combination of Reno and CUBIC behaviours) as options for the congestion control algorithm.

2.5 Congestion Control Startup (Slow Start)

The initial phase of the congestion control algorithm is the slow start phase, as proposed in RFC2001 [13]. This phase was proposed as a simple method of expanding the congestion window quickly at the beginning of a connection to approximate the link bandwidth, before the congestion avoidance phase begins. Slow start is also used when the congestion window drops below the slow start threshold due to inactivity on the connection or large amounts of loss. The default behaviour of the slow start algorithm proposed in RFC2001 is to, for every incoming acknowledgement (ack), increase the size of the congestion window by the number of bytes acknowledged (or in the case of tracking packets, by one packet). Because this ack was just received, it may be shown by definition that the number of bytes in flight is reduced by the number of bytes acknowledged. Thus, by increasing the size of the congestion window by that number of bytes, the number of bytes that can then be transmitted over the link in place of the acknowledged count is doubled. By performing this operation for every acknowledgement that is received, the size of the congestion window grows by two times every RTT, causing it to experience exponential (2^n) growth. Once a loss occurs, the congestion control algorithm saves the current size of the congestion window as the slow start threshold, lowers the congestion window by the amount specified by the congestion control algorithm, and enters the recovery phase to retransmit the data lost before continuing sending new data.

This default implementation of slow start has several major downsides. It must grow quickly to discover the size of the link (additive or even multiplicative increase of the congestion window will not converge on the link capacity effectively), but in doing so, sends up to twice the number of packets the link can process in the last round. It is unable to cope with transient losses (due to an inability to differentiate between loss due to congestion and loss due to wireless errors). Finally, the inclusion of larger and larger buffers on the Internet delay the point at which an endpoint actually experiences loss, allowing the congestion window to grow several times past the true link capacity, even though the delivery rate has stagnated. Especially on high BDP networks, like a GEO satellite connection, there may be tens to hundreds of megabytes of queueing in addition to the link inflight capacity. When a loss occurs, all inflight packets must be discarded by the client until retransmission begins, and no further application data may be transmitted over the link until the retransmissions conclude. For a satellite link, the period between loss and new data delivery is up to several seconds. In recognition of these flaws, the default slow start algorithm has been modified and extended in several ways to address the issue of overshoot.

This analysis yields the delineation of three main outcomes to an exit from slow start [7], depicted in Figure 1.

First, early exit defines an exit from slow start before the connection reaches link capacity. In

Figure 1: Slow start exits: a) early exit, b) overshoot, c) at chokepoint.

this case, the connection must grow in a more inefficient manner to reach capacity. This type of exit occurs when a loss occurs on the physical or link layer, transient congestion events cause a packet to be dropped, or a modification to the slow start algorithm causes a transition from slow start to congestion avoidance prematurely.

Second, late exit defines an exit occurring when the connection has exceeded link capacity. This is the typical behaviour of the slow start phase, as described above. This outcome does not require loss, as a heuristic that can detect the over-capacity and move to congestion avoidance will not incur the retransmission penalty until later in the growth phase of the connection.

Finally, the ideal case is on time/at capacity exit. Because no endpoint has forward knowledge of the link capacity, this outcome is the ideal but cannot realistically be attained. It has the theoretical highest performance of the three. Slow start algorithm design focuses on determining when the connection has passed the capacity point within the fewest round trips, to reduce the amount of excess data buffered on the link and over-capacity transmission. Several heuristics exist to this end, and are described below.

2.6 HyStart

HyStart is a popular modification to the slow start phase of congestion control, implemented by Linux and Windows (with some modifications) for TCP CUBIC [14]. HyStart adds additional checks during the slow start phase, attempting to determine the link capacity point before a packet loss occurs to signal the move to congestion avoidance. The two methods HyStart provides for early-exit are RTT increase, signaling queuing on the route; or ack train spacing, signaling complete utilization of the link. HyStart performs well over common links, and measurably improves connection goodput.

As discussed, the RTT of the connection progresses over Viasat links increase drastically compared to the initial RTT due to queuing during uplink reservations as the transmission rate increases up to the rate specified by the consumer plan. This variation in the RTT presents itself to HyStart as triggering one or both of the congestion signals, causing the connection to move from slow start to congestion avoidance after the first several round trips - far before the congestion window has grown

to the link capacity.

In the remainder of this paper, we discuss alternative modifications to slow start (and as an alternative to HyStart). In the next section, we discuss works related to the development of congestion control and slow start.

3 Related Works

3.1 Congestion Control Algorithms

Caini and Firrincieli [5] propose TCP Hybla, a replacement for both slow start and congestion avoidance sections of the congestion control algorithm. Hybla modifies the CWND growth in each phase by a scale factor derived from the link RTT, such that a Hybla connection finds the link capacity in the same time as in a comparable low-RTT network. Hybla is still a loss-based congestion control algorithm but is able to grow back to link capacity faster in congestion avoidance after a loss occurs [3].

Alrshah, Al-Maqri, and Othman [15] propose Elastic-TCP, which attempts to address issues faced by alternative congestion control algorithms on high-BDP networks. Elastic-TCP estimates the BDP of the link and attempts to grow the CWND aggressively. However, this algorithm still maintains the standard exponential slow start CWND growth, and does not address the initial large period of packet loss and retransmission after overshooting link capacity.

Jin, Wei, and Low [16] present Fast-TCP, an early delay-based congestion control algorithm. They show that this algorithm performs better than state-of-the-art, and scales window growth based on RTT.

Cardwell et al. [12] propose TCP BBR, a new congestion-based congestion control that performs probing of network links as an alternative to typical loss-based congestion control. BBR implements a binary search to increase data on the link until queue buildup is detected, before moving into the rest of the congestion control cycle. BBR has been shown to dominate links shared with CUBIC flows, though each algorithm's slow start performance remains similar [17].

3.2 Performance Enhancing Proxies

Kim et al. [18] propose TCP-GEN, a middlebox implementation to wrap High Assurance Internet Protocol Encryptions (HAIPE) flows with a PEP-compatible replacement. This proposes a method of encapsulating encrypted reliable transport protocol data within a standard TCP/IP flow, such that it is compatible with PEP devices. They also introduce the concept of TCP-over-TCP meltdown, which occurs when a reliable flow is encapsulated within another reliable flow, causing the congestion controller of the internal flow to consider the link as infinite bandwidth and overload the end host. The authors propose passing packet drops from the outer to inner flows to address this problem. This is of special importance due to QUIC flows being unable to be handled by standard PEP devices.

Pavur et al. [19] present QPEP, a PEP implementation utilizing QUIC as the PEP transport protocol typically implemented with a satellite-optimized TCP (such as Hybla). They show that a QUIC-based solution improves performance by utilizing streams within a single QPEP connection for

connection encapsulation, improving both speed and security.

3.3 Slow Start Mechanisms

Liu et al. [20] propose Jump Start, a replacement slow start mechanism that bypasses the exponential growth phase by pacing an initial burst of data after the completion of the handshake. Jump Start calculates the speed and size of the initial transmission based on the initial handshake RTT, data to be sent, and advertised window size from the receiver. Jump Start has been added to QUIC implementations, though it is off by default.

Ha and Rhee [14] present HyStart, a new slow start mechanism to detect congestion on a link and exit before loss occurs. They recognize that standard slow start has negative impacts on connections due to large amounts of packet loss that cause end hosts to become overloaded and waste valuable network resources. HyStart proposes two heuristics to detect congestion. ACK trains determine when the spacing between ACKs becomes consistent between rounds, indicating that there is no further capacity that can be taken up, and RTT delay samples indicate that queues are filling on the link, indicating that the endpoint has surpassed link capacity. This algorithm performs well on many terrestrial networks, but reads the variation in RTT seen on satellite networks as congestion early into the connection [21].

Balasubramanian, Huang, and Olson [22] present HyStart++, a modification to HyStart that works to address issues with variable RTTs by adding an intermediary state between slow start and congestion avoidance. This state is triggered when HyStart would declare an exit, and monitors the HyStart signals for several more rounds before officially declaring an exit to congestion avoidance if they continue to show congestion. HyStart++ additionally removes the ACK train condition included in HyStart.

Katchooei et al. [7] propose SEARCH, a modification to the slow start mechanism that tracks delivery rates with a sliding window to detect deviation between sending and delivery rates before packet loss occurs. SEARCH's performance benefits have been shown analytically with packet traces over GEO, Low Earth Orbit (LEO), and 4G LTE links for TCP. We choose this algorithm to focus on in QUIC.

3.4 QUIC Performance Evaluation

Custura et al. [23] present measurements of ACK frequency on QUIC, with a focus on lowering the impact on the return path, of considerable concern for highly-asymetric links. They show that reducing ACK ratio up to 10 has little effect on connection throughput, though recommend retaining the standard ratio of 2 for slow start period, and ensuring that ACKs are transmitted at least once every

quarter-RTT.

Volodina and Rathgeb [24] demonstrate a method of auto-tuning QUIC's ACK frequency to the characteristics of a link. They show that this method improves performance compared to static ACK frequency configurations.

Border et al. [25] present an analysis of web requests utilizing QUIC as compared to PEP-enhanced TCP flows. They find that gQUIC consistently performs worse than TCP-PEP, and better than native TCP in cases when packet loss on the link increases.

Endres et al. [26] provide measurements of QUIC implementation interoperability and performance over a high-RTT emulated satellite link. They show that some implementations are not interoperable over the network conditions introduced, and implementations overall provide low goodput over the link.

Kuhn et al. [27] present an overview of QUIC performance compared to native- and PEP-TCP over satellite networks. They find issues with long QUIC flows, and propose new base settings for QUIC to improve satellite performance.

Mishra and Leong [28] discuss the variety of congestion control algorithms in use across double-digit counts of QUIC implementations. They show that many QUIC implementations are not fair, despite usage of similar CCAs, and analyze possible improvements that may be made to improve them.

4 Methodology

In this section, we discuss in detail the testbed, modifications made to the Quicly library and Qperf, algorithmic implementation, takeaways from the Quicly codebase, testing justification and procedures, and experimental design.

4.1 Base QUIC Implementation

For this project, we select the Quicly project[2] as our QUIC implementation, developed by Fastly for the H2O webserver. Quicly is actively deployed and has been utilized for previous QUIC research. We select the Qperf project[3] as a wrapper for performing iperf3-like behavior with Quicly, and extend the project to implement necessary features to support our testing.

To confirm the behaviour of the satellite link, and ensure that the library and Qperf wrapper behaved appropriately over the satellite link, we downloaded the latest commit of Qperf and built it. This commit of Qperf was out of date with the latest work performed on Quicly, and one of the primary tasks was upgrading the Quicly library to the latest version. This merge was completed in mid-October 2023, and this version of the Quicly library (with any and all additional modifications performed) was used for the remainder of the project. Prior to the submission of this MQP, the modifications made to both Quicly and Qperf were submitted as pull requests to their respective upstream repositories, to share the modifications and findings of this report with the wider QUIC community.

4.2 Testbed Configuration

To test the default behaviour of Quicly/QUIC over the satellite link, we uploaded copies of the updated software to a client (glomma) and server (MLCnetX) machine, our basic testbed for this project. The testbed setup is displayed in Figure 2.

The client runs Ubuntu 18.04.6 LTS with kernel version 4.15.0-202-generic. It is equipped with a single Intel Core i7-5820k CPU at 3.3 GHz, 32 GB of RAM, and a single 4 TB HDD. The MLCnet servers are a set of four virtualized servers, denoted A, B, C, and D. They are virtualized using Kubernetes, running on shared resources. MLCnetC was the primary server used throughout the course of this project. MLCnetC runs Ubuntu 18.04.6 LTS with kernel version 4.15.0-194-generic. It is equipped with two cores of an Intel E312xx CPU (shared with other servers) at 2.5 GHz, 4 GB of RAM, and 42 GB of available HDD space. This hard drive space was expanded with 50GB of additional storage during later tests to accommodate the amount of data being generated.

[2] `https://github.com/h2o/quicly`
[3] `https://github.com/rbruenig/qperf`

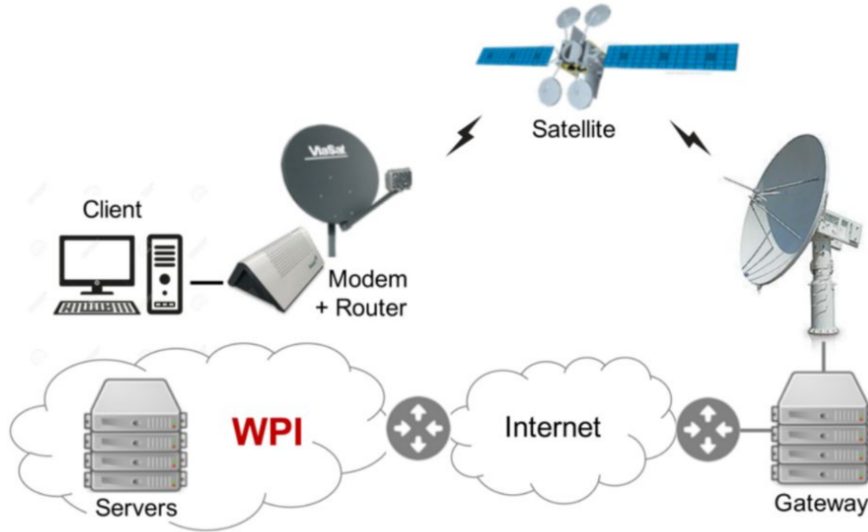


Figure 2: GEO Satellite Testbed

The server connects to our University LAN via Gb/s Ethernet. The campus network is connected to the Internet via several 10 Gb/s links, all throttled to 1 Gb/s.

The client connects to a Viasat GEO satellite terminal (with a dish and modem) via a Gb/s Ethernet connection. The client's downstream Viasat service plan provides a peak data rate of 144 Mb/s. The terminal communicates through a Ka-band outdoor antenna (RF amplifier, up/down converter, reflector, and feed) through the Viasat 2 satellite[4] to the larger Ka-band gateway antenna. The terminal supports adaptive coding and modulation using 16-APK, 8 PSK, and QPSK (forward) at 10 to 52 MSym/s and 8PSK, QPSK and BPSK (return) at 0.625 to 20 MSym/s.

The Viasat gateway performs per-client queue management, where the queue for each client can grow up to 36 MBytes. Queue lengths are controlled at the gateway by Active Queue Management (AQM) that randomly drops 25% of incoming packets when the queue is over half of the limit (i.e., 18 MBytes). While the gateway provides a GEO performance-enhancing proxy (PEP), this does not apply to this project due to the use of QUIC, as the encrypted UDP flow is unable to be proxied as discussed in Section 2.

Both the client and the server are controlled directly by logging into the machines via SSH and beginning tests manually. Remote operation was considered, but difficulties were encountered while attempting to start multiple background processes over remote, automated SSH connections.

4.3 Modular Slow Start

Rather than duplicating a single congestion control algorithm and adding our slow start modification into it, we propose a new method of modular slow start implementation. We extend the congestion

[4] `https://en.wikipedia.org/wiki/ViaSat-2`

controller structure definition with a new field, which points to a slow start function. A configuration setting in the startup context allows dynamic selection of the slow start algorithm, and each congestion control implementation may choose to utilize the slow start function pointer, allowing non loss-based controllers to be implemented as normal. This method reduces the time required to implement alternative slow start algorithms to existing congestion controllers, and decouples the slow start phase from the congestion avoidance phase in software to better match the behavior of most congestion controllers.

Quicly implements congestion controllers as a constant struct with a number of function pointers to each event that the congestion control algorithm responds to. A pointer to this struct is saved in the connection context, allowing different connections to implement different congestion controllers. The slow start method is saved in a struct in the congestion control struct as a variable field, to allow for modification of the slow start algorithm at runtime.

The signature of the slow start method is equivalent to the signature of `cc_on_acked`, as it is intended to be called during acknowledgement handling. The default slow start function is also provided in the Quicly default contexts, so a user of the Quicly library does not have to explicitly specify their slow start setting. The default slow start is RFC2001. By setting the slow start function in the context passed to the Quicly library startup, the library will set it in the congestion controller's context, if the selected congestion control algorithm supports custom slow start behaviour.

The congestion controller is stateful by use of a congestion control data structure. This structure contains all fields necessary for maintaining information necessary information about the connection. We extend this structure with a union of structs for slow start algorithm use. Each new slow start algorithm may declare its own struct within this union, with fields specific to that algorithm, and use those fields when it is called. This is a sister struct to the congestion controller state union, which implements structs for each flavour of congestion control.

This method of implementing slow start in a modular fashion is used to implement the default slow start exponential growth (RFC2001), HyStart, Hybla, and SEARCH. While HyStart and Hybla were implemented and lightly tested, RFC2001 as baseline and SEARCH were implemented as the main candidates for study.

4.4 Implementation of SEARCH

This project focused on the implementation of SEARCH, a new slow start mechanism [7]. During slow start, the congestion controller operates by doubling the number of bytes in flight every RTT, until the delivery rate reaches the link capacity, shown in Figure 3. The SEARCH algorithm operates by tracking the delivered bytes over time, and comparing the active delivery rate to the expected delivery rate from the previous RTT. Delivery rates for each period are computed over a window, set to 3.5x the

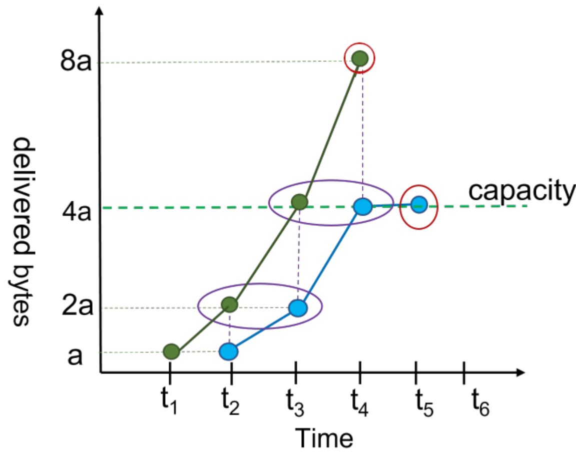


Figure 3: Delivered bytes vs. expected delivered bytes per RTT

initial RTT. When these rates deviate, SEARCH exits the slow start phase and moves to congestion avoidance. The algorithm utilized for these tests is given in Algorithm 1.

Our implementation requires several fields not presently available in the default congestion control data struct. Therefore, we extend the slow start union with a SEARCH-specific struct:

```
union {
    struct {
        /**
         * Bins for the byte count sent and the byte count delivered (instantiated on init)
         */
        uint64_t delv_bins[SEARCH20_SENT_BIN_COUNT];
        /**
         * Maintains the end time of the current bin
         */
        int64_t bin_end;
        /**
         * Holds the size of each bin (based on the handshake RTT)
         */
        uint32_t bin_time;
        /**
         * Counts the number of times that the bin has been incremented, so we know when to
         * start trying to watch for congestion
         */
        uint8_t bin_rounds;
    } search20delv;
} ss_state;
```

Several modifications were deemed necessary through initial rounds of testing to implement the algorithm correctly. We implement a check to ensure that the bin duration is always non-zero. When calculating the number of bins to shift to the past, we perform rounding to ensure we are inspecting the bins closest to the timeframe of interest, though discussions of utilizing interpolation to include the values of partial bins cut by an imperfect shift are ongoing. We perform a reset if the time between receiving sequential acknowledgements was larger than the number of bins available for storage, to ensure SEARCH is performed properly when an application continues sending after a pause. We set

Algorithm 1
SEARCH 2.0: Slow start Exit At Right CHokepoint.

```
1: Parameters:
2: WINDOW_SIZE = Initial_RTT × 3.5
3: W = 10
4: EXTRA_BINS = 15
5: NUM_BINS = W + EXTRA_BINS
6: BIN_DURATION = WINDOW_SIZE / W
7: THRESH = 0.35

8: LOWER_CWND = TRUE
9: Initialization:
10: bin[NUM_BINS]
11: curr = 0
12: bin_end = now + BIN_DURATION

13: Each acknowledgement:
14: if (now > bin_end) then
15:    bin_end += BIN_DURATION
16:    curr += 1
17:    bin[curr mod NUM_BINS] = 0

18:    prev = curr - (RTT / BIN_DURATION)
19:    if (prev ≥ W) and (curr - prev) ≤ EXTRA_BINS then
20:       // Check if SEARCH should exit
21:       curr_delv = Σ_{curr-W}^{curr} bin[i mod NUM_BINS]
22:       prev_delv = Σ_{prev-W}^{prev} bin[i mod NUM_BINS]
23:       norm_diff = (2 · prev_delv − curr_delv) / (2 · prev_delv)

24:       if (norm_diff ≥ THRESH) then
25:          // Exit slow start
26:          if (LOWER_CWND) then
27:             back = (Initial_RTT · 2) / BIN_DURATION

28:             overshoot = Σ_{curr - back}^{curr} bin[i mod NUM_BINS]
29:             set ssthresh to (cwnd - overshoot)
30:          else
31:             set ssthresh to cwnd
32:          end if
33:       end if
34:    end if
35: end if
36: bin[curr mod NUM_BINS] += bytes_delivered
```

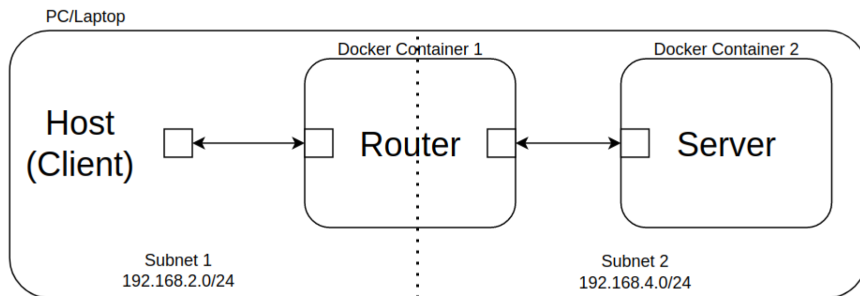


Figure 4: General overview of the emulation testbed

all bins that were skipped over to the appropriate value of zero when a gap in acknowledgements occurs that does not trigger the previous reset condition.

One significant modification to the original proposed algorithm is implemented as follows. When exiting slow start at the chokepoint, SEARCH is delayed in its prediction by at least one RTT, and the algorithm has knowledge of the exact amount of deviation between the predicted delivery rate and the actual delivery rate. Thus, the number of bytes excess added to the CWND since it bypassed the BDP of the link can be shown to be approximated by the number of bytes marked as delivered in the period of 2x initial RTT to exit. We perform tests with and without this modification added, and compare the values of the CWND as is, the SEARCH-predicted CWND (removing the two-RTT window byte count), and the CUBIC-predicted CWND (treating SEARCH exit as loss of zero bytes, and scaling CWND by the CUBIC β value).

We include a listing of our SEARCH algorithm in Appendix A.

4.5 Emulated Network Tests

To qualify the behavior of SEARCH in QUIC, we create a emulated test network consisting of two Docker containers, a router and a server, on two local subnets. The router is configured with the Linux traffic control to implement a 150Mbit link with a 300ms one-way delay and a 36MB queue on each outgoing virtual interface to the host machine and the server container. Corresponding virtual interfaces are added to the host machine and the server container to communicate through the router container. Finally, the router is configured to route all traffic between the subnets through the virtual interfaces, to emulate the satellite link. Tests using this configuration were performed on a Thinkpad T14 AMD Ryzen 7 PRO 4750U with 32GB of DDR4 RAM. Repeatable runs were performed over this emulated link to verify the Quicly configuration on the link and show that the QUIC flow was behaving properly.

We implemented this emulation testbed due to issues with the physical testbed Vorma, which had been used in previous MQPs. We provide this emulation testbed for future research groups.

This environment had several uses in the iterative implementation of the SEARCH algorithm. As a quiet environment, it removed alternate sources of congestion or less-friendly network operators from impacting the behaviour during initial tests of both the Quicly codebase and the SEARCH algorithm. This allowed us to determine when Quicly was behaving differently than expected by default and modify settings to ensure tests were stressing the slow start algorithm. It also allowed us to gather clean, repeatable traces of SEARCH's operation, which helped in debugging the implementation of the algorithm before runs over the satellite link were performed. While this environment did not emulate perfectly all of the behaviours seen while operating over the satellite link, it provided a good first pass to verify algorithm behaviour.

4.6 QUIC Max Data and Max Window

One shortcoming with Quicly that the emulated environment revealed was the impact that the QUIC window setting has. QUIC limits the cumulative number of bytes available to be sent by either endpoint on a connection as shared state, which exists external (and limits) the congestion control algorithm, as shown in Figure 5.

By default, Quicly implements the QUIC maximum window parameter with a size of 16MB. To grow the maximum window of the connection, a Max Data frame must be transmitted from receiver to sender to grow the window and enable more data to be sent on the link. Quicly's implementation to determine when these frames should be sent is not BDP aware, and grows the maximum window based on percent usage of the previously transmitted window. By default, the throughput and the RTTs on the connection appeared as in Figure 6. In this scenario, each spike of the RTT was the server sending the maximum allowed data as the congestion window grew to infinity, due to loss not occurring as the router queue buffered the entire window each time. This queue would drain as the server was blocked from sending until the next max data packet arrived, at which point the cycle would repeat. To fix this and ensure that the congestion controller was controlling the data rate and receiving loss signals, we increased the window parameter substantially. For further experiments, we set the maximum window parameter to 256MB, yielding flows that showed throughput and RTT values as expected, shown in Figure 7.

4.7 Server Test Settings and Procedures

We configured the server with several settings to ensure that packet loss was occuring from queueing loss rather than intermediary network operators dropping high-rate UDP flows unnecessarily. We found that by limiting the rate of the server to 200Mbps, the connection was able to progress to a point at which the sending rate and delivery rate deviated. Without a rate cap on the server, packet

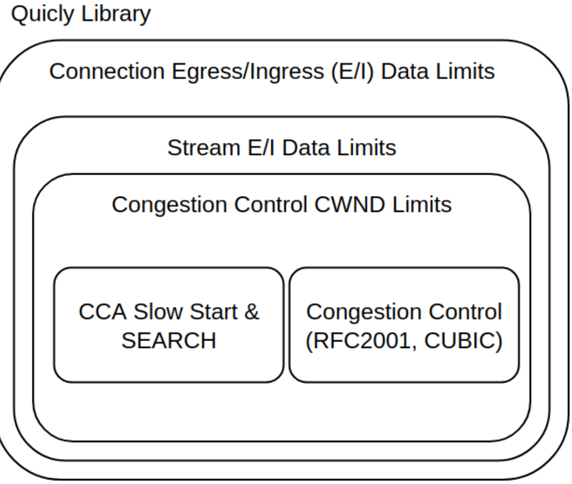


Figure 5: Onion model of Quicly's implementation of sender limits.

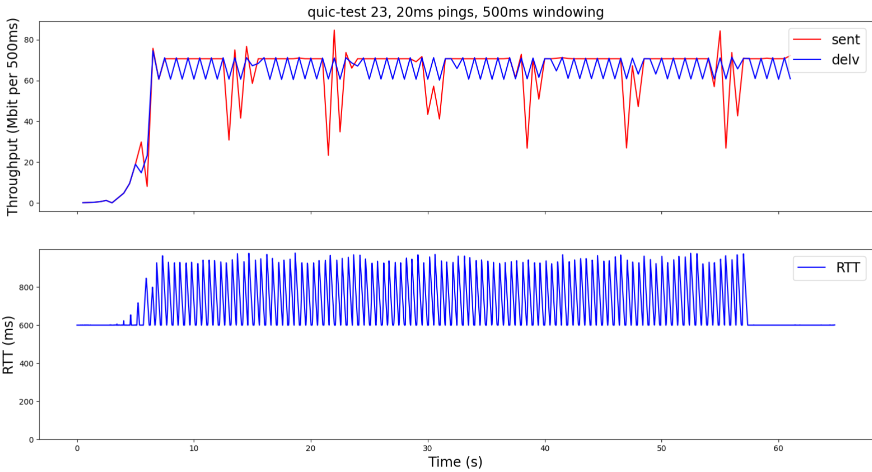


Figure 6: Transmission rate and RTTs of a CUBIC connection in the emulated link, with default Quicly settings.

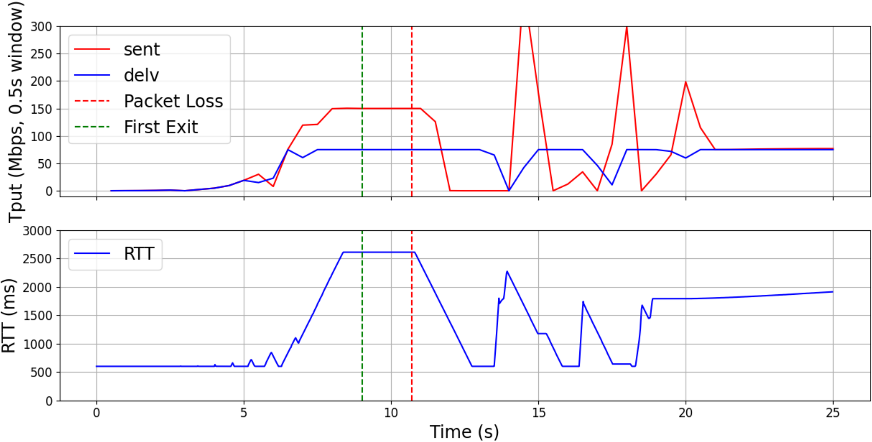


Figure 7: Transmission rate and RTTs of a CUBIC connection in the emulated link, with large maximum window growth.

loss would occur at or before the point of theoretical deviation. This rate limit was applied using the Linux traffic control (tc) utility's queue discipline settings. The server was configured with a token bucket filter (TBF) queue with a maximum rate of 200Mbps, a burst rate of 200Kb, and a latency of 1.5s.

4.8 Qperf Modifications

Several modifications were necessary to allow Qperf to function as needed for the purposes of this project. Qperf was necessary as the Quicly library itself, while it did provide a command line interface (CLI) testing interface, did not support more generalized test procedures similar to iperf3. Qperf provided a foundation for many simple procedures, but was lacking in specificity and did not provide flags to configure all available QUIC/Quicly settings. We expanded Qperf to expose several necessary settings, including the maximum window, and added the ability to select our slow start algorithm. We also added the ability to select a download size, as an alternative to the time limit option, for better measuring goodput over the link. Finally, we added an option if SEARCH was enabled to toggle whether it would exit slow start or continue standard slow start while logging SEARCH information.

4.9 End Goal Test Justification

In developing our experiments, we created several heuristics to help measure the validity of our tests. The first heuristic was measuring the time delta between when SEARCH first declared that the flow should exit slow start, and when the first congestion loss occurred. By computing the ratio of this time difference to the RTT at first exit, we generate a metric of how early SEARCH is predicting the exit, and likewise how much data a SEARCH-triggered exit is keeping from being lost on the link by detecting the congestion point before loss. Second, we measure the goodput of the link. The goodput is defined as the time taken to successfully deliver a number of bytes to the client application, and yields different results than throughput when considering real-world scenarios. For our tests, we measure the performance of SEARCH by measuring the time taken to successfully deliver [N] bytes, the time delta ratio as described, and in some tests the number of bytes removed from the congestion window by SEARCH on exit due to overgrowth.

4.10 Experimental Design

To test the behavior of the SEARCH algorithm, we record time to download to measure the goodput of the link with and without SEARCH enabled. This is accomplished by performing a 200MB download, and logging the time taken to deliver 1MB chunks. Runs were capped at 60 seconds and alternated between SEARCH and standard slow start enabled, with a 60 second sleep period before starting the

next run. Three sets of runs are presented: 20 hours, from a Sunday night to a Monday afternoon; 24 hours, from a Monday night to a Tuesday night; and 24 hours, from a Wednesday night to a Thursday night. The first set was capped to 20 hours due to disk usage issues. The second is considered a clean run. The third implemented congestion window lowering, and is thus analyzed separately from the previous two.

5 Results

We present the results of several runs performed on the Viasat testbed. We discuss the methods of data collection and analysis as relates to We show that SEARCH improves the performance measurably on larger downloads and compare how SEARCH performance develops compared to baseline as the size of the download increases.

5.1 First-RTT and Wireless Loss

When using qperf and Quicly to download data on our Viasat testbed, we found a non-insignificant number of runs reported loss in the first RTT after the connection establishment. This behavior is specific to the satellite and does not occur in the emulation setup. The root cause of these reported losses is not established. These runs occur whether or not SEARCH is enabled, and do not accurately represent the impact that SEARCH has on the connection. Thus, these runs are excluded from further analysis.

To ensure that we are measuring the impact of SEARCH itself, we discard runs that contain early loss - defined as runs that contain a packet loss before the congestion point. We find that runs that exit before 9 seconds have not entered congestion due to the underlying characteristics of the link (per the baseline RTT), and are considered to have been subject to wireless loss before congestion occurred. These runs are excluded (marked in the Wireless Loss column), unless SEARCH declared an exit before that point.

5.2 Dataset Description

Table 1 displays Dataset 1, made up of the first and second (20hr, 24hr) long runs. These runs were performed during a weekend in late March 2024, and during a weekday in early April 2024, respectively. During these runs, SEARCH was configured to exit slow start without modifying the congestion window.

	First-RTT Loss	Wireless Loss	Clean	**Total**
Baseline	103	12	544	659
SEARCH	116	8	535	659
Total	219	20	1079	1318

Table 1: Dataset 1 makeup, no CWND modification performed.

Table 2 displays Dataset 2, with runs performing congestion window modification, made up of the third (24hr) long run performed during a weekday in early April, 2024. Runs with SEARCH enabled are unable to be directly compared with dataset 1, but runs that only logged can be combined for analysis.

	First-RTT Loss	Wireless Loss	Clean	**Total**
Baseline	106	22	232	360
SEARCH	83	23	254	360
Total	189	45	486	720

Table 2: Dataset 2 makeup, CWND modification performed.

Table 3 displays the statistics of the Baselines dataset, the combined dataset of runs with SEARCH disabled. This increases the population size for analysis of baseline behaviour.

	First-RTT Loss	Wireless Loss	Clean	**Total**
Dataset 1	103	12	544	659
Dataset 2	106	22	232	360
Combined	209	34	776	1019

Table 3: Baselines dataset makeup, composed of non-SEARCH exit runs from Dataset 1 and Dataset 2.

For the remainder of this section, only the runs listed in the Clean columns of Tables 1, 2, and 3 are considered for analysis. We do not consider runs that fall into the other categories because they are considered transient loss that does not accurately reflect the impact of the changes to the slow start algorithm. For runs with first-RTT loss, neither SEARCH nor the baseline slow start algorithm would have impacted the performance of the run. For the runs meeting the wireless loss heuristic, the same reasoning is applied, though by explicitly including runs where SEARCH was the root cause of that early exit, we do not hide cases where SEARCH likely caused the run to perform worse than average.

5.3 Dataset Analysis

In Figure 8, we present a cumulative distribution function (CDF) of the cleaned baseline runs, all of which log SEARCH information without performing the early exit from slow start. We observe that for cleaned runs in this combined dataset, $\tilde{4}5\%$ of runs experience loss before SEARCH declares an exit. In the remaining $\tilde{5}5\%$, SEARCH declares an exit before loss occurs, and for $\tilde{3}5\%$ of those runs, that exit point is found greater than one RTT before loss.

Table 4 lists the number of runs from Dataset 1 that succeed in delivering the specified byte counts. We do not remove runs which fail to deliver the total desired byte count of 200MB in further evaluation.

	Total	50MB	100MB	150MB	200MB
Baseline	544	449	421	386	358
SEARCH	535	445	419	380	323
Total	1079	894	840	766	681

Table 4: Count of runs delivering [N] MB, Dataset 1.

Figure 9 depicts the time to deliver [N] bytes for each run, with each run color-coded by its slow start setting. This displays the runs that report loss in the first RTT trailing into the upper left, and

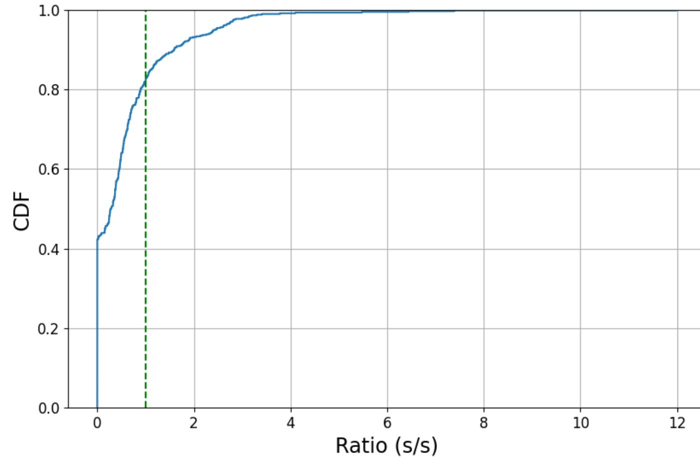


Figure 8: Ratio of time between SEARCH declared exit and loss occurring, CDF of cleaned runs. Green dashed vertical line indicates exit one RTT before loss.

shows the exponential growth of the slow start phase for the remainder of the runs.

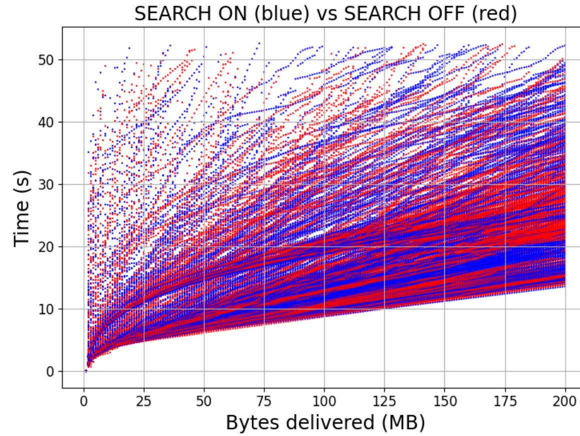


Figure 9: Scatterplot displaying behaviour of all runs in Dataset 1.

Figure 10 compares the median, 25%, and 75% quartiles of the time required to deliver the specified byte count for each set of runs. Prior to reaching 130MB, SEARCH and baseline perform nearly identically, as SEARCH does not modify the behavior of standard slow start. At approximately 135MB, the median run without SEARCH begins deviating from the median run with SEARCH enabled. The lower quartile of the baseline runs deviate from the lower quartile of the SEARCH runs at 175MB, showing the impact that continuing to double the sending rate has on the overall goodput.

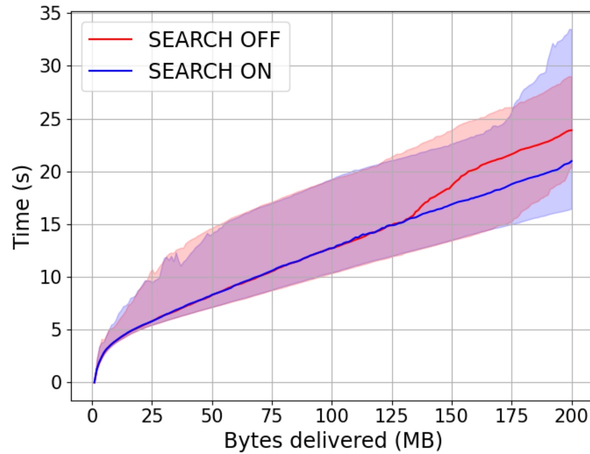


Figure 10: Median time to download [N] MB, shading depicts 25% and 75% quartiles

Figure 11 depicts the median time savings by enabling SEARCH. We display the spread between the difference of the quartiles as the shaded area. We find that SEARCH provides benefit by reducing the download time of files over 125MB by up to three seconds by extending the region before packet loss occurs compared to baseline.

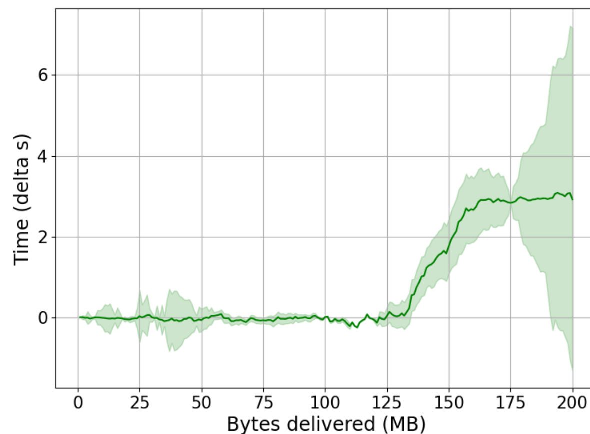


Figure 11: Difference in median run timing, shading depicts absolute difference between quartiles

In Figure 12, we show CDFs of the time to download specific byte counts for all included runs. The number of runs included in this analysis are presented in Table 4. We observe breakaway between SEARCH and baseline beginning to appear in the 125MB CDF, and becoming more apparent as the download progresses. We observe that 45% of runs have no visible difference in speed to download 150MB, an additional 40% of runs perform better with SEARCH by several seconds, and the remaining 15% form the tail of the CDF, where baseline performs slightly better than SEARCH. This difference continues as more data is downloaded, with SEARCH expanding the gap by multiple seconds for downloads in 40% of cases, while an additional 20-40% of runs lag behind baseline by similar amounts, depending on the CDF considered. We observe a knee in early CDFs that begins to drop away

as the runs continue, which may be due to a subset of runs with early loss not removed by the wireless/transient loss heuristic.

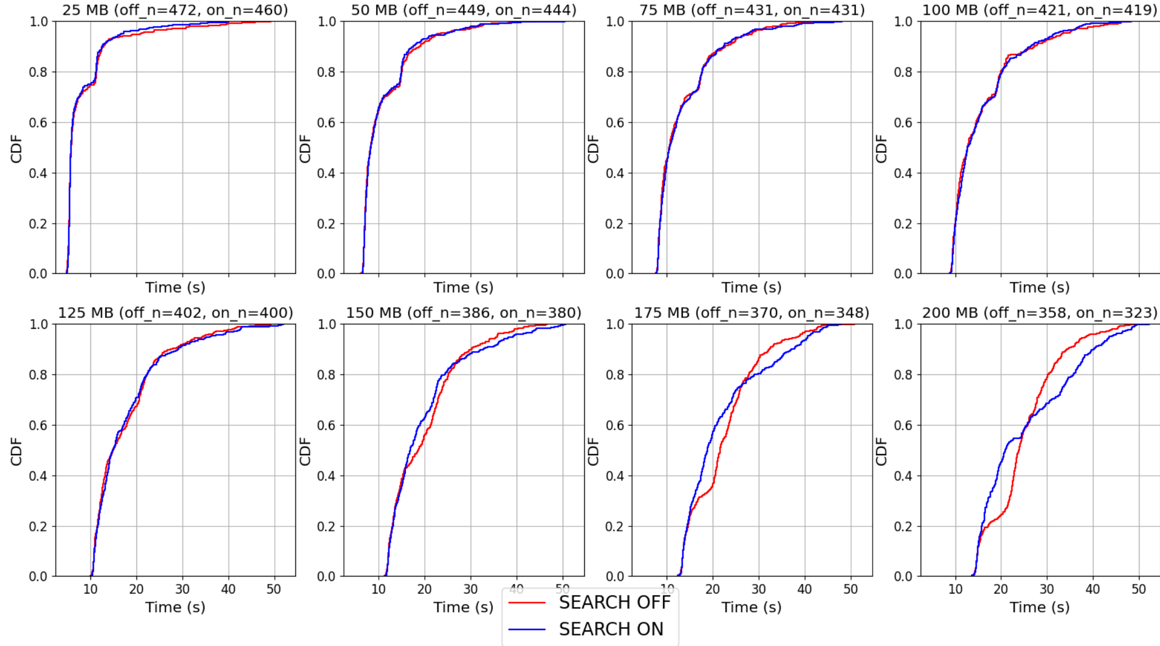


Figure 12: CDFs of the time taken to deliver [N] bytes, with surviving runs labeled in subtitles.

Finally, we present analysis comparing SEARCH CWND reduction to the equivalent CWND reduction if SEARCH declaring exit was treated as a packet loss, similar to exiting slow start due to explicit congestion notification (ECN) marks. This analysis is presented in Figure 13 as a CDF. This analysis stems from Dataset 2, with run statistics presented in Table 2. By reducing the CWND by only the amount of bytes transmitted since congestion began, SEARCH is able to maintain a higher sending rate that is more accurate to link capacity than it would if the exit was declared as a packet loss. The congestion window reduction of CUBIC, when represented as a fraction of the congestion window at exit, is a constant 0.3x. This is because CUBIC performs a constant reduction by β, set to 0.7 in most distributions. Only in a small fraction of runs does SEARCH remove more bytes from the congestion window than CUBIC would. This tail may be modified by a comparison to remove the minimum number of bytes between the two options, though more specific research would be required to determine if this action is appropriate.

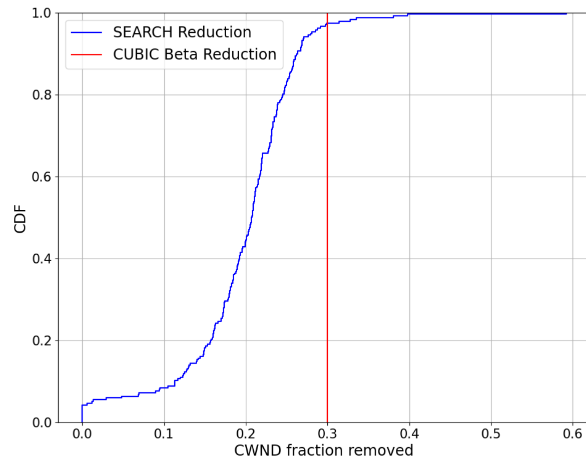


Figure 13: Ratio of the reduction of the CWND at SEARCH exit point to the original CWND value, SEARCH computation vs CUBIC β multiplicative decrease

6 Discussion

In this section, we discuss experimental results and analyze the impact of SEARCH on connections over the Viasat link.

6.1 SEARCH Performance

When analyzing the performance of SEARCH's chokepoint detection heuristic, we find that the long runs produced different results than prior experimentation had. A series of earlier pilot tests, collected in February and March 2024, yielded a much higher ratio of times when SEARCH detected exit more than one RTT before loss occurred. These tests were performed to help show the validity of the algorithm, ensure the testing framework was stable, and develop the data processing pipeline before longer tests were conducted. The results of these tests are not included in this paper due to concerns of statistical significance, though they shape some of our understanding of experimental results. The cause of this change is unknown, and may be due to weather-related events, as the sets of long runs used were collected during days with intermittent rain, a known factor in reducing the performance of the physical link. However, this is not definitively know.

By monitoring the quartiles, we avoid the impact that outliers have on the measurement of performance. This is useful for showing how well SEARCH performs against baseline to some degree, but do not present a complete view of the performance. Because runs were cut off at the 60s mark rather than allowing the entire download to finish every time, the population observed as the number of downloads increases self-selects for those runs that performed well the entire time, and avoids showing the worst-case performance.

Given this caveat, while we show that SEARCH does improve performance by a significant, measurable margin, we acknowledge the indicators that show SEARCH not performing better than standard slow start in all cases. The growth of the inversion of SEARCH performance and the base slow start implementation as the download size grows may indicate that, in the default exit configuration, SEARCH simply delays the realization of queuing loss. This theory is supported by the understanding that, because SEARCH ideally must trigger after the chokepoint, there has already been some amount of overshoot that is present on the link. This overshoot (excess bytes) only grows with the standard exit strategy, and lowering the CWND on exit is a promising future strategy. We show that SEARCH can compute a theoretical link capacity summing the previous RTTs of data, and that this sum is typically less aggressive than a standard CUBIC reduction. However, this does not show that SEARCH's prediction is entirely correct, which would require further measurements of the link capacity. Further, we posit that a reduction directly to the link capacity may not be as beneficial to connection per-

formance due to the overshoot, and that lowering the CWND further than the congestion point may enable those excess bytes to be cleared before congestion avoidance growth begins queuing additional packets.

6.2 Modular Slow Start

The usage and implementation of various slow starts in this project as modular components to the greater congestion control algorithm was useful in several ways. We found that the ability to separate the behaviour of slow start from the behaviour of congestion avoidance did speed up some development time when implementing different versions of SEARCH, and help clarify the difference between implementing a fully different congestion controller from implementing a small part of the same. It did present some technical problems during implementation, both in terms of design (where to place the slow start function pointers in the codebase), and implementation debugging (ensuring that the function pointers were not overwritten).

6.3 First-RTT Losses

One of the major problems faced during several stages of this project was the prevalence of losses reported by Quicly in the first seconds of the connection, which we deemed "first-RTT loss". We found that this type of loss affected between 10-25% of all runs, and was not directly linked to time of day, weather conditions that may affect link capacity, or server/client settings. The root cause of these losses were traced to the generation of "gap acks", QUIC's equivalent of the TCP SACK extension. We found that under some conditions, the client would generate a gap ack which the server would interpret as a loss, although no packets were seen missing when packet captures were inspected. We believe that this occurs due to particular variation in the link RTT causing a timeout on the client.

6.4 Quicly Performance

Another difficulty encountered with Quicly's QUIC implementation is the implementation of maximum window growth. Quicly implements maximum window growth independent of link BDP, only triggering a max data frame (the control frame that increases the maximum window of the connection) to be generated when the window is more than 50% utilized. When operating over a high-BDP link, following this method causes the client to not recognize that it needs to alert the server that it is willing to accept more data on the connection before the server becomes blocked. The server then sends a data blocked control frame to the client, indicating a request for the maximum window to be expanded, creating oscillatory behaviour.

Before updating the client during tests to operate with a max data setting of 256MB, we experimented with the addition of a Hybla-like maximum window growth. This change would grow the maximum window by the max data parameter multiplied by the ratio of the initial RTT over a base RTT (25ms). However, we rolled back this modification due to its violation of the max data parameter directly controlling the allowable growth of the maximum window. Therefore, we believe that the client should implement BDP-aware behaviour to determine when the server may be reaching the connection window limit based on the product of initial RTT and current throughput.

The impact on CPU usage with SEARCH enabled/disabled is believed to be minimal, though this could become more of a concern as the RTTs are lowered on terrestrial links. The majority of SEARCH computation is made up of accumulating delivered bytes, dependent on the delivery rate of the connection, and the threshold computation on binrolls is performed in constant time.

7 Future Work

In this section, we discuss future improvements to this project and some limitations of this work.

7.1 Congestion Control

On the whole, modern congestion control algorithms are inefficient when handling the traffic of connections including satellite links. The physical properties of the link are hostile towards non-adaptive algorithms like HyStart, and most congestion control algorithms are developed with the intention of usage over links with latencies of 100ms maximum. When the loss rate of satellite links are taken into account as well, loss-based congestion control is unable to maintain high throughput. We propose several methods for improving the performance of congestion control in

7.1.1 Slow Start

Because the behaviour of the default slow start algorithm is clocked by the RTT of the link, the link RTT controls the rate at which the congestion window can grow. Hybla was the primary alternative to this standard behaviour, but usage of this algorithm only exacerbates the post-slow start recovery period due to larger increases in throughput placing more data on the link during the period between reaching link capacity and loss occurring.

We propose several changes to SEARCH to be tested. First, modification to reduce CWND by the number of bytes sent in the previous two RTTs was not shown to significantly effect the behaviour of the algorithm compared to default SEARCH behaviour. We recommend both further testing of this modification in order to determine if it is effective at increasing goodput beyond the default SEARCH behaviour. Second, it is known that when SEARCH exits, the point at which the CWND surpassed link capacity was at least one RTT in the past. This data is still present in queues on the link, and to continue sending at the link rate will only grow these queues until loss occurs (though they do induce less loss when drops occur due to congestion avoidance). By reducing the CWND below the estimated link capacity by some amount, the losses may be delayed further. We recommend further investigation of this concept, both for QUIC and TCP implementations of SEARCH.

Finally, we believe that an analysis of fairness between multiple QUIC flows is necessary to show the impact of SEARCH on connections with the algorithm enabled, and how SEARCH causes flows to behave in the presence of competing flows that do not utilize the algorithm. This analysis may be completed for each iteration of SEARCH, as described above, which will additionally show the behaviour of SEARCH competing with variants of itself.

7.1.2 Variable Congestion Window Growth

This work implemented SEARCH in QUIC and showed that the usage of the algorithm measurably improved performance over baseline. We propose combining SEARCH and Hybla to produce a hybrid algorithm, to implement RTT-aware startup behaviour and mitigate the impact of higher-than-baseline losses introduced by Hybla with SEARCH's capabilities. We believe that this modification may allow Hybla to achieve greater performance for large downloads.

The RTT variation of the satellite link is a consequence of the satellite modem's bandwidth negotiation process during connection startup. CWND growth during slow start is typically performed by increasing the CWND by the number of bytes acked in each incoming packet. We propose investigating varying the rate of CWND growth during slow start by multiplying the number of bytes acked by some variable based on the RTT change in previous rounds. This modification would attempt to smooth CWND growth and queue usage during slow start, potentially increasing the effectiveness of SEARCH by reducing the noise in the signal.

7.2 New Quicly Features

Between the initial forking of Quicly (October, 2023) and the completion of this project (April, 2024), several features were added to the implementation that may affect future experimentation with this library.

7.2.1 Pacing

Pacing modifies the sending rate of the endpoint to reduce spikes in network usage and reduce the probability of transient packet drops and to improve compatibility and link sharing with other flows. Pacing should be analyzed for its impact on the slow start phase - by increasing the spacing between packets based on the initial RTT, the CWND may grow slower than it would if these packets were clumped in transmissions. It may also improve the stability of the link as stated by reducing the chance of spurious losses causing early exit for high-bandwidth flows.

7.2.2 Jump Start

Jump Start implements a fast-transmit option for a new non-initial application flows. This replaces the slow start mechanism and attempts to fill the link with all available application data, within limits defined by the congestion controller. This behaviour may cause issues with satellite terminals in particular, due to the necessity of bandwidth reservation for new flows. Jump Start, by replacing the slow start mechanism, may not implicitly benefit from the usage of SEARCH or other slow start oriented heuristic measures. We recommend a modification to Jump Start to allow SEARCH byte

tracking during the time that Jump Start is active, if Jump Start completes the fast-transmit without loss and enters the slow start phase, such that SEARCH can be active for the remainder of slow start.

7.3 Testing Procedures

In this work, we rate limit the outgoing link due to early testing with unlimited (1Gbps) throughput generating loss events early in the connection, at approximately the 10s mark. We recommend further work be done to understand the true cause of this loss, and determine if alternative rate-reducing methods like packet pacing improve performance.

QUIC provides specification for a logging format, qlog[5], to improve visibility into flow behaviour. This project did not utilize qlog, and we believe that future projects should include this in the workflow from the beginning. By utilizing this logging format, packet and frame headers and significant events are logged in a machine-readable format for later inspection. We believe that the usage of qlog will improve the ability to inspect unexpected flow behaviour and increase the efficiency of future QUIC-based projects. Quicly provides support for the format, though qlog is currently still in drafting by the IETF.

[5] `https://github.com/quicwg/qlog`

8 Conclusion

This work provides a solution to the difficult environment of networking over GEO satellite links. Due to the physical position of the satellite and infrastructure provided by it, these satellite links are high bandwidth with high, variable latencies. Traditional congestion controllers fail to efficiently utilize this link capacity due to their design and validation typically occurring over terrestrial Internet links. Particularly, the QUIC transport protocol faces additional challenges due to technical limitations with accelerating its flows using traditional proxy methods.

Of special concern is the slow start phase, which probes for link capacity during connection startup. Slow start exit is typically due to packet loss, which causes multi-second delays while retransmission occurs. Prior work has attempted to address this issue by a heuristic to determine the congestion point before loss occurs, but RTT variation over the satellite triggers this check within the first several round trips.

We introduce a new slow start algorithm, SEARCH, and a new method for implementing slow start algorithms into a larger congestion control algorithm to improve the performance of QUIC over satellite networks. We implement SEARCH in an open-source QUIC implementation, Quicly. We validate the performance of QUIC with SEARCH in an emulated network constructed with Docker containers on a host machine. We then validate the performance of QUIC with SEARCH on Viasat-2, a real GEO satellite link, and develop harnesses to enable large-scale tests of the system with our modifications. We show that after the first 125MB of a download, SEARCH lowers the median download time by up to 3 seconds. Finally, we discuss future modifications to SEARCH and future research directions for this project.

References

[1] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," num Pages: 151. [Online]. Available: https://datatracker.ietf.org/doc/rfc9000

[2] David Belson and Lucas Pardue. Examining HTTP/3 usage one year on. [Online]. Available: https://blog.cloudflare.com/http3-usage-one-year-on

[3] S. Claypool, J. Chung, and M. Claypool, "Comparison of TCP congestion control performance over a satellite network," in *Passive and Active Measurement*, O. Hohlfeld, A. Lutu, and D. Levin, Eds. Springer International Publishing, pp. 499–512.

[4] J. Griner, J. Border, M. Kojo, Z. D. Shelby, and G. Montenegro, "Performance enhancing proxies intended to mitigate link-related degradations," num Pages: 45. [Online]. Available: https://datatracker.ietf.org/doc/rfc3135

[5] C. Caini and R. Firrincieli, "TCP hybla: a TCP enhancement for heterogeneous networks," vol. 22, no. 5, pp. 547–566. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/sat.799

[6] J. Gettys and K. Nichols, "Bufferbloat: Dark buffers in the internet: Networks without effective AQM may again be vulnerable to congestion collapse." vol. 9, no. 11, pp. 40–54. [Online]. Available: https://dl.acm.org/doi/10.1145/2063166.2071893

[7] M. A. Kachooei, J. Chung, F. Li, B. Peters, and M. Claypool, "SEARCH: Robust TCP slow start performance over satellite networks," in *2023 IEEE 48th Conference on Local Computer Networks (LCN)*, pp. 1–4, ISSN: 2832-1421.

[8] Geostationary orbit | satellite, communications & telemetry | britannica. [Online]. Available: https://www.britannica.com/technology/low-Earth-orbit

[9] Low earth orbit (LEO) | definition, distance, & facts | britannica. [Online]. Available: https://www.britannica.com/technology/low-Earth-orbit

[10] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The QUIC transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* ACM, pp. 183–196. [Online]. Available: https://dl.acm.org/doi/10.1145/3098822.3098842

[11] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," vol. 42, no. 5, pp. 64–74. [Online]. Available: https://dl.acm.org/doi/10.1145/1400097.1400105

[12] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time," vol. 14, no. 5, pp. 20–53. [Online]. Available: https://dl.acm.org/doi/10.1145/3012426.3022184

[13] W. R. Stevens, "TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms," num Pages: 6. [Online]. Available: https://datatracker.ietf.org/doc/rfc2001

[14] S. Ha and I. Rhee, "Taming the elephants: New TCP slow start," vol. 55, no. 9, pp. 2092–2110. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1389128611000363

[15] M. A. Alrshah, M. A. Al-Maqri, and M. Othman, "Elastic-TCP: Flexible congestion control algorithm to adapt for high-BDP networks," vol. 13, no. 2, pp. 1336–1346. [Online]. Available: https://ieeexplore.ieee.org/document/8642512/

[16] C. Jin, D. Wei, and S. Low, "FAST TCP: motivation, architecture, algorithms, performance," in *IEEE INFOCOM 2004*, vol. 4, pp. 2490–2501 vol.4, ISSN: 0743-166X. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1354670

[17] P. Zhao, B. Peters, J. Chung, and M. Claypool, "Competing TCP congestion control algorithms over a satellite network," in *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE Press, pp. 132–138. [Online]. Available: https://doi.org/10.1109/CCNC49033.2022.9700541

[18] Y. Kim, J.-Y. Jo, R. Harkanson, and K. Pham, "TCP-GEN framework to achieve high performance for HAIPE-encrypted TCP traffic in a satellite communication environment," in *2018 IEEE International Conference on Communications (ICC)*, pp. 1–7, ISSN: 1938-1883.

[19] J. Pavur, M. Strohmeier, V. Lenders, and I. Martinovic, "QPEP: A QUIC-based approach to encrypted performance enhancing proxies for high-latency satellite broadband." [Online]. Available: http://arxiv.org/abs/2002.05091

[20] D. Liu, M. Allman, S. Jin, and L. Wang, "Congestion control without a startup phase." [Online]. Available: https://www.semanticscholar.org/paper/Congestion-Control-Without-a-Startup-Phase-Liu-Allman/2c2e25a8d869f2c954807f1dc6f964cca77836ce

[21] B. Peters, P. Zhao, J. Chung, and M. Claypool, "TCP HyStart performance over a satellite network." [Online]. Available: https://www.

semanticscholar.org/paper/TCP-HyStart-Performance-over-a-Satellite-Network-Peters-Zhao/6e0d58fe936e8d48661ae8ef214577898fdb497e

[22] P. Balasubramanian, Y. Huang, and M. Olson, "HyStart++: Modified slow start for TCP," p. RFC9406. [Online]. Available: https://www.rfc-editor.org/info/rfc9406

[23] A. Custura, T. Jones, R. Secchi, and G. Fairhurst, "Reducing the acknowledgement frequency in IETF QUIC," vol. 41, no. 4, pp. 315–330, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sat.1466. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sat.1466

[24] E. Volodina and E. P. Rathgeb, "The ACK policy optimization algorithm for QUIC," in *2023 IEEE 48th Conference on Local Computer Networks (LCN)*, pp. 1–8, ISSN: 2832-1421. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10223366

[25] J. Border, B. Shah, C.-J. Su, and R. Torres, "Evaluating QUIC's performance against performance enhancing proxy over satellite link," in *2020 IFIP Networking Conference (Networking)*, pp. 755–760. [Online]. Available: https://ieeexplore.ieee.org/document/9142718

[26] S. Endres, J. Deutschmann, K.-S. Hielscher, and R. German, "Performance of QUIC implementations over geostationary satellite links." [Online]. Available: http://arxiv.org/abs/2202.08228

[27] N. Kuhn, F. Michel, L. Thomas, E. Dubois, E. Lochin, F. Simo, and D. Pradas, "QUIC: Opportunities and threats in SATCOM," vol. 40, no. 6, pp. 379–391, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sat.1432. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sat.1432

[28] A. Mishra and B. Leong, "Containing the cambrian explosion in QUIC congestion control," in *Proceedings of the 2023 ACM on Internet Measurement Conference*, ser. IMC '23. Association for Computing Machinery, pp. 526–539. [Online]. Available: https://dl.acm.org/doi/10.1145/3618257.3624811

A SEARCH Code Listing

```
#include "quicly/ss.h"
#include <stdint.h>

/*
This slow start algorithm implements the SEARCH algorithm, as derived in
https://web.cs.wpi.edu/~claypool/papers/search-wowmom-24/
*/

void ss_search20_delv_reset(quicly_cc_t *cc, const quicly_loss_t *loss, uint32_t bytes,
    int64_t now);

void ss_search20_delv_reset(quicly_cc_t *cc, const quicly_loss_t *loss, uint32_t bytes,
    int64_t now)
{
       // Handy pointers to the cc struct
       uint64_t* delv = cc->ss_state.search20delv.delv_bins;
       int64_t* bin_end = &cc->ss_state.search20delv.bin_end;
       uint32_t* bin_time = &cc->ss_state.search20delv.bin_time;
       uint8_t* bin_rounds = &cc->ss_state.search20delv.bin_rounds;

       // bin time is the size of each of the sent/delv bins
       *bin_time = MAX((loss->rtt.latest * SEARCH20_WINDOW_MULTIPLIER) / (
           SEARCH20_DELV_BIN_COUNT), 1);
       *bin_end = now + *bin_time;
       delv[0] = 0;
       *bin_rounds = 0;
}

// bytes is the number of bytes acked in the last ACK frame
// inflight is sentmap->bytes_in_flight + bytes
void ss_search20_delv(quicly_cc_t *cc, const quicly_loss_t *loss, uint32_t bytes, uint64_t
    largest_acked, uint32_t inflight,
                     uint64_t next_pn, int64_t now, uint32_t max_udp_payload_size)
{
       // Handy pointers to the cc struct
       uint64_t* delv = cc->ss_state.search20delv.delv_bins;
       int64_t* bin_end = &cc->ss_state.search20delv.bin_end;
       uint32_t* bin_time = &cc->ss_state.search20delv.bin_time;
       uint8_t* bin_rounds = &cc->ss_state.search20delv.bin_rounds;

       // struct initializations, everything else important has already been reset to 0
       if(*bin_time == 0) {
              ss_search20_delv_reset(cc, loss, bytes, now);
       }

       // bin_shift is the number of bins to shift backwards, based on the latest RTT
       uint8_t bin_shift = loss->rtt.latest / *bin_time;
       if(bin_shift == 0) {
              bin_shift = 1;
       }
       else if(loss->rtt.latest % *bin_time > (*bin_time / 2)) {
              // round to the nearest bin (not doing interpolation yet)
              bin_shift++;
       }

       // Possibly add some code here for dirty reset - run when no data has been sent on the
            connection
       // for a very long time, but application never received a loss (and so is still in
```

```
    slow-start)
// This is likely handled by the prior binroll while loop, but that might add
    unnecessary latency
// dependant on how long ago the last packet was acknowledged.
if (((now - *bin_end) / *bin_time) > SEARCH20_SENT_BIN_COUNT) {
        ss_search20_delv_reset(cc, loss, bytes, now);
}


// perform prior binrolls before updating the latest bin to run SEARCH on if necessary
while((now - *bin_time) > (*bin_end)) {
        *bin_end += *bin_time;
        *bin_rounds += 1;
        delv[(*bin_rounds % (SEARCH20_SENT_BIN_COUNT))] = 0;
}
// perform current binroll
if((now > (*bin_end))) {
        // only perform SEARCH if there is enough data in the sent bins with the
            current RTT
        // bin_rounds tracks how many times we've rolled over, and a single window is
            the entire
        // delivered bin count (because of the definition of how bin_time is calculated
            )
        // thus, the number of rounds must be >= than the delv bin count + the bin
            shift
        if((*bin_rounds) >= ((SEARCH20_DELV_BIN_COUNT) + bin_shift)
                && bin_shift < (SEARCH20_SENT_BIN_COUNT - SEARCH20_DELV_BIN_COUNT)) {
                // do SEARCH
                double shift_delv_sum = 0, delv_sum = 0;
                for (int i = *bin_rounds; i > (*bin_rounds - (SEARCH20_DELV_BIN_COUNT));
                     i--) {
                        // the value of bin_shift will always be at least 1, so the
                            current sent bin is never used
                        shift_delv_sum += delv[((i - bin_shift) % (
                            SEARCH20_SENT_BIN_COUNT))];
                        delv_sum += delv[(i % (SEARCH20_SENT_BIN_COUNT))];
                }
                if (shift_delv_sum >= 1) {
                        shift_delv_sum *= 2;
                        double normalized_diff = (shift_delv_sum - delv_sum) /
                            shift_delv_sum;
                        if (normalized_diff > SEARCH20_THRESH) {
                                // exit slow start
                                // TODO: Proposal to lower cwnd by tracked previously
                                    sent bytes
                                if (cc->cwnd_maximum < cc->cwnd)
                                        cc->cwnd_maximum = cc->cwnd;
                                cc->ssthresh = cc->cwnd;
                                cc->cwnd_exiting_slow_start = cc->cwnd;
                                cc->exit_slow_start_at = now;
                                return;
                        }
                }
        }
        else if(bin_shift >= (SEARCH20_SENT_BIN_COUNT - SEARCH20_DELV_BIN_COUNT)) {
                /* TODO: Double bin_time and consolidate for high RTT operation */
        }

        *bin_end += *bin_time;
        *bin_rounds += 1;
        delv[(*bin_rounds % (SEARCH20_SENT_BIN_COUNT))] = 0;
}
```

```
        // fill (updated) bin with latest acknowledged bytes
        // TCP implementation has a method of tracking total delivered bytes to avoid this per
            -packet
        // computation, but we aren't doing that (yet). loss->total_bytes_sent looks
            interesting, but
        // does not seem to guarantee a match with conn->egress.max_data.sent (see loss.c)
        delv[(*bin_rounds % (SEARCH20_SENT_BIN_COUNT))] += bytes;

        // perform standard SS doubling
        cc->cwnd += bytes;
        if (cc->cwnd_maximum < cc->cwnd)
                cc->cwnd_maximum = cc->cwnd;

        return;
}

quicly_ss_type_t quicly_ss_type_search20_delv = { "search", ss_search20_delv };
```