The QUIC Transport Protocol: Design and Internet-Scale Deployment

Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, Zhongyi Shi *
Google
quic-sigcomm@google.com

ABSTRACT

We present our experience with QUIC, an encrypted, multiplexed, and low-latency transport protocol designed from the ground up to improve transport performance for HTTPS traffic and to enable rapid deployment and continued evolution of transport mechanisms. QUIC has been globally deployed at Google on thousands of servers and is used to serve traffic to a range of clients including a widely-used web browser (Chrome) and a popular mobile video streaming app (YouTube). We estimate that 7% of Internet traffic is now QUIC. We describe our motivations for developing a new transport, the principles that guided our design, the Internet-scale process that we used to perform iterative experiments on QUIC, performance improvements seen by our various services, and our experience deploying QUIC globally. We also share lessons about transport design and the Internet ecosystem that we learned from our deployment.

CCS CONCEPTS

• **Networks** → **Network protocol design**; **Transport protocols**; *Cross-layer protocols*;

ACM Reference format:
Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, Zhongyi Shi . 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098842

1 INTRODUCTION

We present QUIC, a new transport designed from the ground up to improve performance for HTTPS traffic and to enable rapid deployment and continued evolution of transport mechanisms. QUIC replaces most of the traditional HTTPS stack: HTTP/2, TLS, and TCP (Figure 1). We developed QUIC as a user-space transport with UDP as a substrate. Building QUIC in user-space facilitated its deployment as part of various applications and enabled iterative changes to occur at application update timescales. The use of UDP allows QUIC packets to traverse middleboxes. QUIC is an encrypted transport: packets are authenticated and encrypted, preventing modification and limiting ossification of the protocol by middleboxes. QUIC uses a cryptographic handshake that minimizes handshake latency for most connections by using known server credentials on repeat connections and by removing redundant handshake-overhead at multiple layers in the network stack. QUIC eliminates head-of-line blocking delays by using a lightweight data-structuring abstraction, *streams*, which are multiplexed within a single connection so that loss of a single packet blocks only streams with data in that packet.

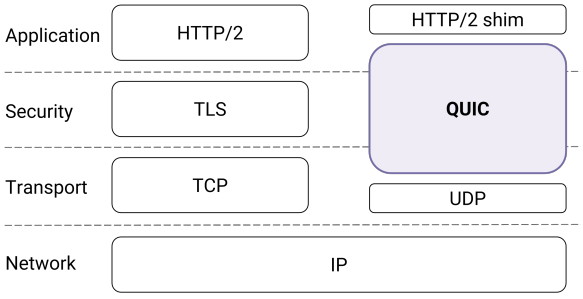

Figure 1: QUIC in the traditional HTTPS stack.

On the server-side, our experience comes from deploying QUIC at Google's front-end servers, which collectively handle billions of requests a day from web browsers and mobile apps across a wide range of services. On the client side, we have deployed QUIC in Chrome, in our mobile video streaming YouTube app, and in the Google Search app on Android. We find that on average, QUIC reduces latency of Google Search responses by 8.0% for desktop users and by 3.6% for mobile users, and reduces rebuffer rates of YouTube playbacks by 18.0% for desktop users and 15.3% for mobile users[1]. As shown in Figure 2, QUIC is widely deployed: it currently accounts for over 30% of Google's total egress traffic in bytes and consequently an estimated 7% of global Internet traffic [61].

We launched an early version of QUIC as an experiment in 2013. After several iterations with the protocol and following our deployment experience over three years, an IETF working group was formed to standardize it [2]. QUIC is a single monolithic protocol in

*Fedor Kouranov is now at Yandex, and Jim Roskind is now at Amazon. Author names are in alphabetical order.

SIGCOMM '17, August 21-25, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-4653-5/17/08.
https://doi.org/10.1145/3098822.3098842

[1]Throughout this paper "desktop" refers to Chrome running on desktop platforms (Windows, Mac, Linux, etc.) and "mobile" refers to apps running on Android devices.

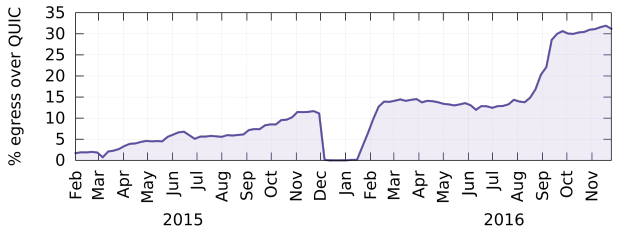

Figure 2: Timeline showing the percentage of Google traffic served over QUIC. Significant increases and decreases are described in Section 5.1.

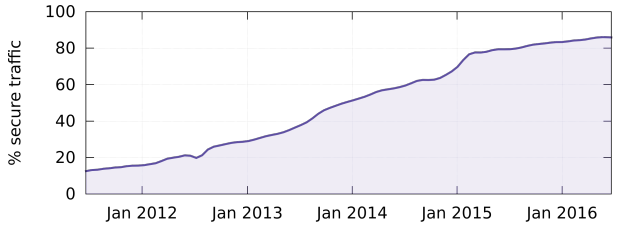

Figure 3: Increase in secure web traffic to Google's front-end servers.

our current deployment, but IETF standardization will modularize it into constituent parts. In addition to separating out and specifying the core protocol [33, 34], IETF work will describe an explicit mapping of HTTP on QUIC [9] and separate and replace QUIC's cryptographic handshake with the more recent TLS 1.3 [55, 63]. This paper describes pre-IETF QUIC design and deployment. While details of the protocol will change through IETF deliberation, we expect its core design and performance to remain unchanged.

In this paper, we often interleave our discussions of the protocol, its use in the HTTPS stack, and its implementation. These three are deeply intertwined in our experience. The paper attempts to reflect this connectedness without losing clarity.

2 MOTIVATION: WHY QUIC?

Growth in latency-sensitive web services and use of the web as a platform for applications is placing unprecedented demands on reducing web latency. Web latency remains an impediment to improving user-experience [21, 25], and tail latency remains a hurdle to scaling the web platform [15]. At the same time, the Internet is rapidly shifting from insecure to secure traffic, which adds delays. As an example of a general trend, Figure 3 shows how secure web traffic to Google has increased dramatically over a short period of time as services have embraced HTTPS. Efforts to reduce latency in the underlying transport mechanisms commonly run into the following fundamental limitations of the TLS/TCP ecosystem.

Protocol Entrenchment: While new transport protocols have been specified to meet evolving application demands beyond TCP's simple service [40, 62], they have not seen wide deployment [49, 52, 58]. Middleboxes have accidentally become key control points in the Internet's architecture: firewalls tend to block anything unfamiliar for security reasons and Network Address Translators (NATs) rewrite the transport header, making both incapable of allowing traffic from new transports without adding explicit support for them. Any packet content not protected by end-to-end security, such as the TCP packet header, has become fair game for middleboxes to inspect and modify. As a result, even modifying TCP remains challenging due to its ossification by middleboxes [29, 49, 54]. Deploying changes to TCP has reached a point of diminishing returns, where simple protocol changes are now expected to take upwards of a decade to see significant deployment (see Section 8).

Implementation Entrenchment: As the Internet continues to evolve and as attacks on various parts of the infrastructure (including the transport) remain a threat, there is a need to be able to deploy changes to clients rapidly. TCP is commonly implemented in the Operating System (OS) kernel. As a result, even if TCP modifications were deployable, pushing changes to TCP stacks typically requires OS upgrades. This coupling of the transport implementation to the OS limits deployment velocity of TCP changes; OS upgrades have system-wide impact and the upgrade pipelines and mechanisms are appropriately cautious [28]. Even with increasing mobile OS populations that have more rapid upgrade cycles, sizeable user populations often end up several years behind. OS upgrades at servers tend to be faster by an order of magnitude but can still take many months because of appropriately rigorous stability and performance testing of the entire OS. This limits the deployment and iteration velocity of even simple networking changes.

Handshake Delay: The generality of TCP and TLS continues to serve Internet evolution well, but the costs of layering have become increasingly visible with increasing latency demands on the HTTPS stack. TCP connections commonly incur at least one round-trip delay of connection setup time before any application data can be sent, and TLS adds two round trips to this delay[2]. While network bandwidth has increased over time, the speed of light remains constant. Most connections on the Internet, and certainly most transactions on the web, are short transfers and are most impacted by unnecessary handshake round trips.

Head-of-line Blocking Delay: To reduce latency and overhead costs of using multiple TCP connections, HTTP/1.1 recommends limiting the number of connections initiated by a client to any server [19]. To reduce transaction latency further, HTTP/2 multiplexes multiple objects and recommends using a single TCP connection to any server [8]. TCP's bytestream abstraction, however, prevents applications from controlling the framing of their communications [12] and imposes a "latency tax" on application frames whose delivery must wait for retransmissions of previously lost TCP segments.

In general, the deployment of transport modifications for the web requires changes to web servers and clients, to the transport stack in server and/or client OSes, and often to intervening middleboxes. Deploying changes to all three components requires incentivizing and coordinating between application developers, OS vendors, middlebox vendors, and the network operators that deploy these middleboxes. QUIC encrypts transport headers and builds transport functions atop UDP, avoiding dependence on vendors and network operators and moving control of transport deployment to the applications that directly benefit from them.

[2]TCP Fast Open [11, 53] and TLS 1.3 [55] seek to address this delay, and we discuss them later in Section 8.

3 QUIC DESIGN AND IMPLEMENTATION

QUIC is designed to meet several goals [59], including deployability, security, and reduction in handshake and head-of-line blocking delays. The QUIC protocol combines its cryptographic and transport handshakes to minimize setup RTTs. It multiplexes multiple requests/responses over a single connection by providing each with its own stream, so that no response can be blocked by another. It encrypts and authenticates packets to avoid tampering by middleboxes and to limit ossification of the protocol. It improves loss recovery by using unique packet numbers to avoid retransmission ambiguity and by using explicit signaling in ACKs for accurate RTT measurements. It allows connections to migrate across IP address changes by using a Connection ID to identify connections instead of the IP/port 5-tuple. It provides flow control to limit the amount of data buffered at a slow receiver and ensures that a single stream does not consume all the receiver's buffer by using per-stream flow control limits. Our implementation provides a modular congestion control interface for experimenting with various controllers. Our clients and servers negotiate the use of the protocol without additional latency. This section outlines these elements in QUIC's design and implementation. We do not describe the wire format in detail in this paper, but instead refer the reader to the evolving IETF specification [2].

3.1 Connection Establishment

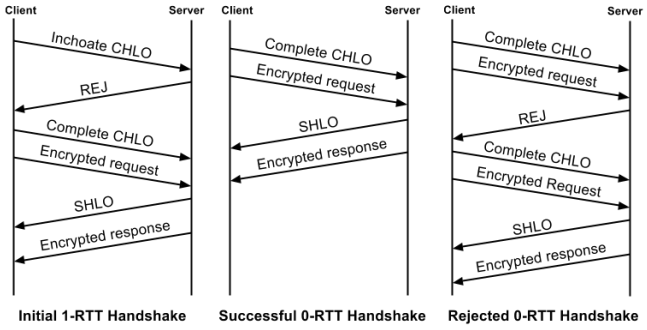

Figure 4: Timeline of QUIC's initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.

QUIC relies on a combined cryptographic and transport handshake for setting up a secure transport connection. On a successful handshake, a client caches information about the *origin*[3]. On subsequent connections to the same origin, the client can establish an encrypted connection with no additional round trips and data can be sent immediately following the client handshake packet without waiting for a reply from the server. QUIC provides a dedicated reliable stream (streams are described below) for performing the cryptographic handshake. This section summarizes the mechanics of QUIC's cryptographic handshake and how it facilitates a *zero round-trip time* (0-RTT) connection setup. Figure 4 shows a schematic of the handshake.

Initial handshake: Initially, the client has no information about the server and so, before a handshake can be attempted, the client sends an *inchoate client hello (CHLO)* message to the server to elicit a *reject (REJ)* message. The REJ message contains: (i) a *server config* that includes the server's long-term Diffie-Hellman public value, (ii) a certificate chain authenticating the server, (iii) a signature of the server config using the private key from the leaf certificate of the chain, and (v) a *source-address token*: an authenticated-encryption block that contains the client's publicly visible IP address (as seen at the server) and a timestamp by the server. The client sends this token back to the server in later handshakes, demonstrating ownership of its IP address. Once the client has received a server config, it authenticates the config by verifying the certificate chain and signature. It then sends a *complete CHLO*, containing the client's ephemeral Diffie-Hellman public value.

Final (and repeat) handshake: All keys for a connection are established using Diffie-Hellman. After sending a complete CHLO, the client is in possession of *initial keys* for the connection since it can calculate the shared value from the server's long-term Diffie-Hellman public value and its own ephemeral Diffie-Hellman private key. At this point, the client is free to start sending application data to the server. Indeed, if it wishes to achieve 0-RTT latency for data, then it must start sending data encrypted with its initial keys before waiting for the server's reply.

If the handshake is successful, the server returns a *server hello (SHLO)* message. This message is encrypted using the initial keys, and contains the server's ephemeral Diffie-Hellman public value. With the peer's ephemeral public value in hand, both sides can calculate the *final* or *forward-secure keys* for the connection. Upon sending an SHLO message, the server immediately switches to sending packets encrypted with the forward-secure keys. Upon receiving the SHLO message, the client switches to sending packets encrypted with the forward-secure keys.

QUIC's cryptography therefore provides two levels of secrecy: initial client data is encrypted using initial keys, and subsequent client data and all server data are encrypted using forward-secure keys. The initial keys provide protection analogous to TLS session resumption with session tickets [60]. The forward-secure keys are ephemeral and provide even greater protection.

The client caches the server config and source-address token, and on a repeat connection to the same origin, uses them to start the connection with a complete CHLO. As shown in Figure 4, the client can now send initial-key-encrypted data to the server, without having to wait for a response from the server.

Eventually, the source address token or the server config may expire, or the server may change certificates, resulting in handshake failure, even if the client sends a complete CHLO. In this case, the server replies with a REJ message, just as if the server had received an inchoate CHLO and the handshake proceeds from there. Further details of the QUIC handshake can be found in [43].

Version Negotiation: QUIC clients and servers perform version negotiation during connection establishment to avoid unnecessary delays. A QUIC client proposes a version to use for the connection in the first packet of the connection and encodes the rest of the handshake using the proposed version. If the server does not speak the client-chosen version, it forces version negotiation by sending back a Version Negotiation packet to the client carrying all of the server's supported versions, causing a round trip of delay before connection establishment. This mechanism eliminates round-trip latency when the client's optimistically-chosen version is spoken

[3] An *origin* is identified by the set of URI scheme, hostname, and port number [5].

by the server, and incentivizes servers to not lag behind clients in deployment of newer versions. To prevent downgrade attacks, the initial version requested by the client and the list of versions supported by the server are both fed into the key-derivation function at both the client and the server while generating the final keys.

3.2 Stream Multiplexing

Applications commonly multiplex units of data within TCP's single-bytestream abstraction. To avoid head-of-line blocking due to TCP's sequential delivery, QUIC supports multiple streams within a connection, ensuring that a lost UDP packet only impacts those streams whose data was carried in that packet. Subsequent data received on other streams can continue to be reassembled and delivered to the application.

QUIC streams are a lightweight abstraction that provide a reliable bidirectional bytestream. Streams can be used for framing application messages of arbitrary size—up to 2^{64} bytes can be transferred on a single stream—but they are lightweight enough that when sending a series of small messages a new stream can reasonably be used for each one. Streams are identified by *stream IDs*, which are statically allocated as odd IDs for client-initiated streams and even IDs for server-initiated streams to avoid collisions. Stream creation is implicit when sending the first bytes on an as-yet unused stream, and stream closing is indicated to the peer by setting a "FIN" bit on the last stream frame. If either the sender or the receiver determines that the data on a stream is no longer needed, then the stream can be canceled without having to tear down the entire QUIC connection. Though streams are reliable abstractions, QUIC does not retransmit data for a stream that has been canceled.

A QUIC packet is composed of a common header followed by one or more *frames*, as shown in Figure 5. QUIC stream multiplexing is implemented by encapsulating stream data in one or more *stream frames*, and a single QUIC packet can carry stream frames from multiple streams.

The rate at which a QUIC endpoint can send data will always be limited (see Sections 3.5 and 3.6). An endpoint must decide how to divide available bandwidth between multiple streams. In our implementation, QUIC simply relies on HTTP/2 stream priorities [8] to schedule writes.

3.3 Authentication and Encryption

With the exception of a few early handshake packets and reset packets, QUIC packets are fully authenticated and mostly encrypted. Figure 5 illustrates the structure of a QUIC packet. The parts of the QUIC packet header outside the cover of encryption are required either for routing or for decrypting the packet: Flags, Connection ID, Version Number, Diversification Nonce, and Packet Number[4]. Flags encode the presence of the Connection ID field and length of the Packet Number field, and must be visible to read subsequent fields. The Connection ID serves routing and identification purposes; it is used by load balancers to direct the connection's traffic to the right server and by the server to locate connection state. The version number and diversification nonce fields are only present in early packets. The server generates the diversification nonce and sends it to the client in the SHLO packet to add entropy into key generation. Both

[4]The details of which fields are visible may change during QUIC's IETF standardization.

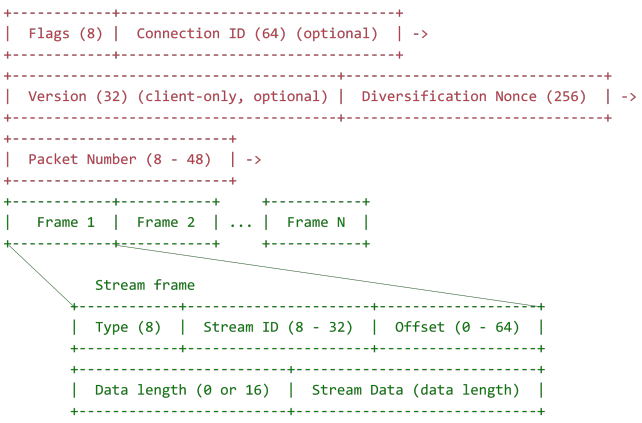

Figure 5: Structure of a QUIC packet, as of version 35 of Google's QUIC implementation. Red is the authenticated but unencrypted public header, green indicates the encrypted body. This packet structure is evolving as QUIC gets standardized at the IETF [2].

endpoints use the packet number as a per-packet nonce, which is necessary to authenticate and decrypt packets. The packet number is placed outside of encryption cover to support decryption of packets received out of order, similar to DTLS [56].

Any information sent in unencrypted handshake packets, such as in the Version Negotiation packet, is included in the derivation of the final connection keys. In-network tampering of these handshake packets causes the final connection keys to be different at the peers, causing the connection to eventually fail without successful decryption of any application data by either peer. Reset packets are sent by a server that does not have state for the connection, which may happen due to a routing change or due to a server restart. As a result, the server does not have the connection's keys, and reset packets are sent unencrypted and unauthenticated[5].

3.4 Loss Recovery

TCP sequence numbers facilitate reliability and represent the order in which bytes are to be delivered at the receiver. This conflation causes the "retransmission ambiguity" problem, since a retransmitted TCP segment carries the same sequence numbers as the original packet [39, 64]. The receiver of a TCP ACK cannot determine whether the ACK was sent for the original transmission or for a retransmission, and the loss of a retransmitted segment is commonly detected via an expensive timeout. Each QUIC packet carries a new packet number, including those carrying retransmitted data. This design obviates the need for a separate mechanism to distinguish the ACK of a retransmission from that of an original transmission, thus avoiding TCP's retransmission ambiguity problem. Stream offsets in stream frames are used for delivery ordering, separating the two functions that TCP conflates. The packet number represents an explicit time-ordering, which enables simpler and more accurate loss detection than in TCP.

QUIC acknowledgments explicitly encode the delay between the receipt of a packet and its acknowledgment being sent. Together with monotonically-increasing packet numbers, this allows for precise

[5]QUIC connections are susceptible to off-path third-party reset packets with spoofed source addresses that terminate the connection. IETF work will address this issue.

network round-trip time (*RTT*) estimation, which aids in loss detection. Accurate RTT estimation can also aid delay-sensing congestion controllers such as BBR [10] and PCC [16]. QUIC's acknowledgments support up to 256 ACK blocks, making QUIC more resilient to reordering and loss than TCP with SACK [46]. Consequently, QUIC can keep more bytes on the wire in the presence of reordering or loss.

These differences between QUIC and TCP allowed us to build simpler and more effective mechanisms for QUIC. We omit further mechanism details in this paper and direct the interested reader to the Internet-draft on QUIC loss detection [33].

3.5 Flow Control

When an application reads data slowly from QUIC's receive buffers, flow control limits the buffer size that the receiver must maintain. A slowly draining stream can consume the entire connection's receive buffer, blocking the sender from sending data on other streams. QUIC ameliorates this potential head-of-line blocking among streams by limiting the buffer that a single stream can consume. QUIC thus employs *connection-level flow control*, which limits the aggregate buffer that a sender can consume at the receiver across all streams, and *stream-level flow control*, which limits the buffer that a sender can consume on any given stream.

Similar to HTTP/2 [8], QUIC employs credit-based flow-control. A QUIC receiver advertises the absolute byte offset within each stream up to which the receiver is willing to receive data. As data is sent, received, and delivered on a particular stream, the receiver periodically sends *window update frames* that increase the advertised offset limit for that stream, allowing the peer to send more data on that stream. Connection-level flow control works in the same way as stream-level flow control, but the bytes delivered and the highest received offset are aggregated across all streams.

Our implementation uses a connection-level window that is substantially larger than the stream-level window, to allow multiple concurrent streams to make progress. Our implementation also uses flow control window auto-tuning analogous to common TCP implementations (see [1] for details.)

3.6 Congestion Control

The QUIC protocol does not rely on a specific congestion control algorithm and our implementation has a pluggable interface to allow experimentation. In our deployment, TCP and QUIC both use Cubic [26] as the congestion controller, with one difference worth noting. For video playback on both desktop and mobile devices, our non-QUIC clients use two TCP connections to the video server to fetch video and audio data. The connections are not designated as audio or video connections; each chunk of audio and video arbitrarily uses one of the two connections. Since the audio and video streams are sent over two streams in a single QUIC connection, QUIC uses a variant of mulTCP [14] for Cubic during the congestion avoidance phase to attain parity in flow-fairness with the use of TCP.

3.7 NAT Rebinding and Connection Migration

QUIC connections are identified by a 64-bit Connection ID. QUIC's Connection ID enables connections to survive changes to the client's IP and port. Such changes can be caused by NAT timeout and rebinding (which tend to be more aggressive for UDP than for TCP [27]) or by the client changing network connectivity to a new IP address. While QUIC endpoints simply elide the problem of NAT rebinding by using the Connection ID to identify connections, client-initiated connection migration is a work in progress with limited deployment at this point.

3.8 QUIC Discovery for HTTPS

A client does not know *a priori* whether a given server speaks QUIC. When our client makes an HTTP request to an origin for the first time, it sends the request over TLS/TCP. Our servers advertise QUIC support by including an *"Alt-Svc"* header in their HTTP responses [48]. This header tells a client that connections to the origin may be attempted using QUIC. The client can now attempt to use QUIC in subsequent requests to the same origin.

On a subsequent HTTP request to the same origin, the client races a QUIC and a TLS/TCP connection, but prefers the QUIC connection by delaying connecting via TLS/TCP by up to 300 ms. Whichever protocol successfully establishes a connection first ends up getting used for that request. If QUIC is blocked on the path, or if the QUIC handshake packet is larger than the path's MTU, then the QUIC handshake fails, and the client uses the fallback TLS/TCP connection.

3.9 Open-Source Implementation

Our implementation of QUIC is available as part of the open-source Chromium project [1]. This implementation is shared code, used by Chrome and other clients such as YouTube, and also by Google servers albeit with additional Google-internal hooks and protections. The source code is in C++, and includes substantial unit and end-to-end testing. The implementation includes a test server and a test client which can be used for experimentation, but are not tuned for production-level performance.

4 EXPERIMENTATION FRAMEWORK

Our development of the QUIC protocol relies heavily on continual Internet-scale experimentation to examine the value of various features and to tune parameters. In this section we describe the experimentation frameworks in Chrome and our server fleet, which allow us to experiment safely with QUIC.

We drove QUIC experimentation by implementing it in Chrome, which has a strong experimentation and analysis framework that allows new features to be A/B tested and evaluated before full launch. Chrome's experimentation framework pseudo-randomly assigns clients to experiments and exports a wide range of metrics, from HTTP error rates to transport handshake latency. Clients that are opted into statistics gathering report their statistics along with a list of their assigned experiments, which subsequently enables us to slice metrics by experiment. This framework also allows us to rapidly disable any experiment, thus protecting users from problematic experiments.

We used this framework to help evolve QUIC rapidly, steering its design according to continuous feedback based on data collected at the full scale of Chrome's deployment. Monitoring a broad array of metrics makes it possible to guard against regressions and to avoid

imposing undue risks that might otherwise result from rapid evolution. As discussed in Section 5, this framework allowed us to contain the impact of the occasional mistake. Perhaps more importantly and unprecedented for a transport, QUIC had the luxury of being able to directly link experiments into analytics of the application services using those connections. For instance, QUIC experimental results might be presented in terms of familiar metrics for a transport, such as packet retransmission rate, but the results were also quantified by user- and application-centric performance metrics, such as web search response times or rebuffer rate for video playbacks. Through small but repeatable improvements and rapid iteration, the QUIC project has been able to establish and sustain an appreciable and steady trajectory of cumulative performance gains.

We added QUIC support to our mobile video (YouTube) and search (Google Search) apps as well. These clients have similar experimentation frameworks that we use for deploying QUIC and measuring its performance.

Google's server fleet consists of thousands of machines distributed globally, within data centers as well as within ISP networks. These front-end servers terminate incoming TLS/TCP and QUIC connections for all our services and perform load-balancing across internal application servers. We have the ability to toggle features on and off on each server, which allows us to rapidly disable broken or buggy features. This mechanism allowed us to perform controlled experiments with QUIC globally while severely limiting the risk of large-scale outages induced by these experiments. Our servers report performance data related to current and historic QUIC connections. This data is collected by a centralized monitoring system that aggregates it and provides visualizations and alerts.

5 INTERNET-SCALE DEPLOYMENT

The experimentation framework described in Section 4 enabled safe global deployment of QUIC to our users. We first present QUIC's deployment timeline. We then describe the evolution of one of several metrics that we monitored carefully as we deployed QUIC globally.

5.1 The Road to Deployment

QUIC support was added to Chrome in June 2013. It was enabled via an optional command-line flag so usage was effectively limited to the QUIC development team. In early 2014, we were confident in QUIC's stability and turned it on via Chrome's experimentation framework for a tiny fraction (< 0.025%) of users. As QUIC proved to be performant and safe, this fraction was increased. As of January 2017, QUIC is turned on for almost[6] all users of Chrome and the Android YouTube app.

Simultaneously developing and deploying a new secure protocol has its difficulties however, and it has not been completely smooth sailing to get to this point. Figure 2 shows QUIC traffic to our services from February 2015 to December 2016. We now describe the two notable events seen in the graph.

Unencrypted data in 0-RTT requests: In December 2015, we discovered a vulnerability in our implementation of the QUIC handshake. The vulnerability was traced to a bug in the client code, which could result in 0-RTT requests being sent unencrypted in an exceedingly rare corner case. Our immediate response was to disable QUIC globally at our servers, using the feature toggle mechanism described in Section 4. This turndown can be seen as the drop to zero, and is also visible in Figures 6 and 14. The bug was fixed and QUIC traffic was restored as updated clients were rolled out.

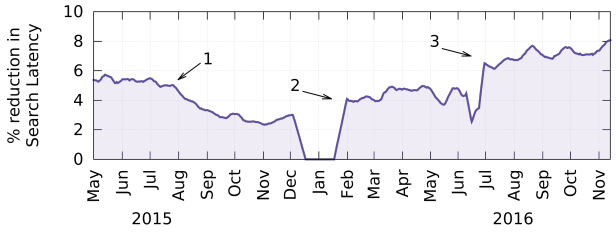

Figure 6: Search Latency reduction for users in the QUIC experiment over an 18-month period. Numbered events are described in Section 5.2.

Increasing QUIC on mobile: A substantial fraction of our users access our services through mobile phones, often using dedicated applications (apps). The majority of our mobile users perform searches via our mobile search app. Similarly, the majority of mobile video playbacks are performed through the YouTube app. The YouTube app started using QUIC in September 2016, doubling the percentage of Google's egress traffic over QUIC, from 15% to over 30%.

5.2 Monitoring Metrics: Search Latency

Our server infrastructure gathers performance data exported by front-end servers and aggregates them with service-specific metrics gathered by the server and clients, to provide visualizations and alerts. We will use *Search Latency* as an example of such a metric. Search Latency is defined as the delay between when a user enters a search term into the client and when all the search-result content is generated and delivered to the client, including images and embedded content. We analyze the evolution of Search Latency improvement for users in the QUIC experiment[7] versus those using TLS/TCP over an 18-month period, as shown in Figure 6.

Over the 18-month period shown in Figure 6 there are two notable regressions, and one improvement. The first regression started in July 2015 (labeled '1' in Figure 6) and lasted for about 5 months. This regression was attributed to changes in our serving infrastructure and to a client configuration bug, both of which inadvertently caused a large number of clients in the QUIC experiment to gradually stop speaking QUIC.

The second regression in December 2015 lines up with the complete disabling of QUIC described in Section 5.1. When QUIC was re-enabled in February 2016 (labeled '2' in Figure 6), the client configuration bug had been fixed, and Search Latency improved, albeit not to the same extent as earlier, since the infrastructure changes hadn't been resolved.

We take a slight detour to describe *restricted edge locations (RELs)* and *UDP proxying*. A large number of our servers are deployed inside ISPs, and we refer to these as *RELs*. For technical, commercial and other considerations, RELs do not terminate TLS sessions to a number of domains. For these domains, RELs therefore simply act as TCP-terminating proxies, proxying the TCP payload to

[6] A small holdback experiment of a few percent allows us to compare QUIC vs. TLS/TCP performance over time.

[7] The QUIC experiment is described in Section 6.1

an unrestricted front-end server for termination of the TLS session and further processing. There is no QUIC equivalent to a TCP-terminating proxy, since the transport session cannot be terminated separately from the rest of the cryptographic session. *UDP-proxying* therefore simply forwards incoming client UDP packets to the unrestricted front-end servers. This allows users getting served at the RELs to use QUIC, since without UDP-proxying the RELs would only be able to speak TCP.

QUIC's performance improvement in July 2016 is attributed to the deployment of UDP-proxying at our RELs (labeled '3' in Figure 6). As a result of UDP-proxying, QUIC's average overall improvement in Search Latency increased from about 4% to over 7%, showing that for this metric, QUIC's latency reductions more than made up for improvements from TCP termination at the RELs.

6 QUIC PERFORMANCE

In this section, we define three key application metrics that drove QUIC's development and deployment, and we describe QUIC's impact on these metrics. We also describe QUIC's CPU utilization at our servers and outline known limitations of QUIC's performance.

Though we use "mobile" as shorthand throughout this paper, we note that it refers to both a difference in operating environment as well as a difference in implementation. The Google Search and YouTube apps were developed independently from Chrome, and while they share the same network stack implementation, they are tuned specifically for the mobile environment. For example, the Google Search app retrieves smaller responses, whose content has been tailored to reduce latency in mobile networks. Similarly, the YouTube app pre-warms connections to reduce video playback latency and uses an Adaptive Bit Rate (ABR) algorithm that is optimized for mobile screens.

Tables 1 and 2 summarize the difference between QUIC users and TLS/TCP users on three metrics: Search Latency, Video Playback Latency, and Video Rebuffer Rate. For each metric, the tables show QUIC's performance impact as a percent reduction between using TLS/TCP and using QUIC. If QUIC decreased Search Latency from 100 seconds to 99 seconds, it would be indicated as a 1% reduction. We describe QUIC's performance on these metrics further below but briefly discuss our experiment setup first.

6.1 Experiment Setup

Our performance data comes from QUIC experiments deployed on various clients, using the clients' frameworks for randomized experimental trials. Users are either in the QUIC experimental group ($QUIC_g$) or in the TLS/TCP control group (TCP_g). Unless explicitly specified, we show QUIC performance as the performance of users in $QUIC_g$, which includes users who were unable to speak QUIC due to failed handshakes. This group also includes data from TLS/TCP usage prior to QUIC discovery as described in Section 3.8. Most users in this group however are able to speak QUIC (see Section 7.2), and most of their traffic is in fact QUIC. Clients capable of using QUIC use TLS/TCP for only 2% of their HTTP transactions to servers which support QUIC. The size of the $QUIC_g$ and TCP_g populations are equal throughout.

Clients that do not use QUIC use HTTP/2[8] over a single TLS/TCP connection for Search and HTTP/1.1 over two TLS/TCP connections for video playbacks. Both QUIC and TCP implementations use a paced form of the Cubic algorithm [26] for congestion avoidance. We show data for desktop and mobile users, with desktop users accessing services through Chrome, and mobile users through dedicated apps with QUIC support. Since TCP Fast Open is enabled at all Google servers, results include such connections. However TCP Fast Open has seen limited deployment at clients (seen Section 8).

Unless otherwise noted, all results were gathered using QUIC version 35 and include over a billion samples. All search results were gathered between December 12, 2016 and December 19, 2016, and all video playback results were gathered between January 19, 2017 and January 26, 2017.

6.2 Transport and Application Metrics

Before diving into application performance, we first discuss transport-level handshake latency as a microbenchmark that QUIC seeks to improve. We then discuss our choice of application metrics used in the rest of this section.

Handshake latency is the amount of time taken to establish a secure transport connection. In TLS/TCP, this includes the time for both the TCP and TLS handshakes to complete. We measured handshake latency at the server as the time from receiving the first TCP SYN or QUIC client hello packet to the point at which the handshake is considered complete. In the case of a QUIC 0-RTT handshake, latency is measured as 0 ms. Figure 7 shows the impact of QUIC's 0-RTT and 1-RTT handshakes on handshake latency.

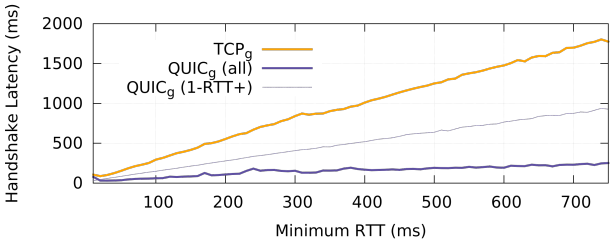

Figure 7: Comparison of handshake latency for $QUIC_g$ and TCP_g versus the minimum RTT of the connection. Solid lines indicate the mean handshake latency for all connections, including 0-RTT connections. The dashed line shows the handshake latency for only those $QUIC_g$ connections that did not achieve a 0-RTT handshake. Data shown is for Desktop connections, mobile connections look similar.

With increasing RTT, average handshake latency for TCP/TLS trends upwards linearly, while QUIC stays almost flat. QUIC's handshake latency is largely insensitive to RTT due to the fixed (zero) latency cost of 0-RTT handshakes, which constitute about 88% of all QUIC handshakes. The slight increase in QUIC handshake latency with RTT is due to the remaining connections that do not successfully connect in 0-RTT. Note that even these remaining connections complete their handshake in less time than the 2- or 3-RTT TLS/TCP handshakes.

We do not show microbenchmarks to characterize transport-level impact of QUIC's improved loss recovery, but this improvement

[8]Google's SPDY protocol [3] has been subsumed by the HTTP/2 standard [8].

manifests itself as higher resilience to loss in general and lower latency for short connections.

Microbenchmarks such as the ones above are a useful measure of whether a transport change is working correctly, but application and user-relevant metrics are a measure of the usefulness of the change. The impact of QUIC's improvements on different application metrics is discussed in the rest of this section, but we offer two insights about the impact of networking changes on applications.

First, networking remains just one constituent of end-to-end application measures. For instance, handshake latency contributes to well under 20% of Search Latency and Video Latency. An almost complete elimination of handshake latency will still yield only a small percentage of total latency reduction. However, even a small change in end-to-end metrics is significant, since this impact is often directly connected to user-experience and revenue. For instance, Amazon estimates that every 100 ms increase in latency cuts profits by 1% [24], Google estimates that increasing web search latency by 100 ms reduces the daily number of searches per user measurably [36], and Yahoo demonstrated that users are more likely to perform clicks on a result page that is served with lower latency [6].

Second, the sensitivity of application metrics to networking changes depends on the maturity of the application. In the rest of this section we describe QUIC's impact on Google Search and YouTube. These are highly-optimized and mature Web applications, and consequently improving end-to-end metrics in them is difficult.

We chose these applications for two reasons. First, improving performance for these highly optimized applications has direct revenue impact. Second, they represent diverse transport use-cases: Search represents a low-load latency-sensitive application, and YouTube represents a heavy-load bandwidth-sensitive application.

6.3 Search Latency

Recall that Search Latency is the delay between when a user enters a search term and when all the search-result content is generated and delivered to the client by Google Search, including all corresponding images and embedded content. On average, an individual search performed by a user results in a total response load of 100 KB for desktop searches and 40 KB for mobile searches. As a metric, Search Latency represents delivery latency for small, delay-sensitive, dynamically-generated payloads.

As shown in Table 1, users in QUIC_g experienced reduced mean Search Latency. The percentile data shows that QUIC_g's improvements increase as base Search Latency increases. This improvement comes primarily from reducing handshake latency, as demonstrated in Figure 9 which shows desktop latency reduction for users in QUIC_g[9] as a function of the client's minimum RTT to the server. As the user's RTT increases, the impact of saving handshake round trips is higher, leading to larger gains in QUIC_g. Figure 8 further shows that users with high RTTs are in a significant tail: more than 20% of all connections have a minimum RTT larger than 150ms, and 10% of all connections have a minimum RTT larger than 300ms. Of the handshake improvements, most of the latency reduction comes from the 0-RTT handshake: about 88% of QUIC connections from desktop achieve a 0-RTT handshake, which is at least a 2-RTT latency saving over TLS/TCP. The remaining QUIC connections still benefit from a 1-RTT handshake.

[9]For the sake of brevity we show only desktop data for these supporting graphs. Mobile trends are similar.

		% latency reduction by percentile						
		Lower latency				Higher latency		
	Mean	1%	5%	10%	50%	90%	95%	99%
Search								
Desktop	8.0	0.4	1.3	1.4	1.5	5.8	10.3	16.7
Mobile	3.6	-0.6	-0.3	0.3	0.5	4.5	8.8	14.3
Video								
Desktop	8.0	1.2	3.1	3.3	4.6	8.4	9.0	10.6
Mobile	5.3	0.0	0.6	0.5	1.2	4.4	5.8	7.5

Table 1: Percent reduction in global Search and Video Latency for users in QUIC_g, at the mean and at specific percentiles. A 16.7% reduction at the 99th percentile indicates that the 99th percentile latency for QUIC_g is 16.7% lower than the 99th percentile latency for TCP_g.

		% rebuffer rate reduction by percentile				
		Fewer rebuffers			More rebuffers	
	Mean	< 93%	93%	94 %	95%	99%
Desktop	18.0	*	100.0	70.4	60.0	18.5
Mobile	15.3	*	*	100.0	52.7	8.7

Table 2: Percent reduction in global Video Rebuffer Rate for users in QUIC_g at the mean and at specific percentiles. An 18.5% reduction at the 99th percentile indicates that the 99th percentile rebuffer rate for QUIC_g is 18.5% lower than the 99th percentile rate for TCP_g. An $*$ indicates that neither QUIC_g nor TCP_g have rebuffers at that percentile.

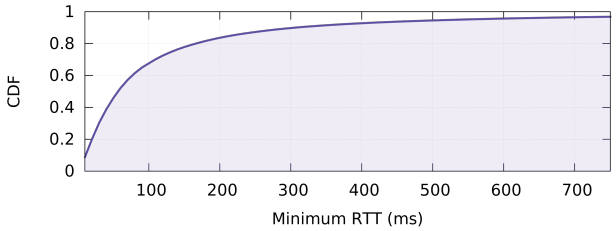

Figure 8: Distribution of connection minimum RTTs for TCP connections to our servers. These results were gathered from video playbacks. The distribution is similar for search connections.

We believe that QUIC's loss recovery mechanisms may also play a role in decreasing Search latency at higher RTTs. Recall that QUIC includes richer signaling than TCP, which enables QUIC loss recovery to be more resilient to higher loss rates than TCP (see Section 3.4). Figure 10 shows the relationship between TCP retransmission rates measured at our servers against minimum client RTTs. Employing TCP retransmission rate as a proxy for network loss, this figure shows that network quality is highly correlated with the client's minimum RTT. Consequently, QUIC's improved loss recovery may also contribute to Search Latency improvement at higher RTTs.

Table 1 shows that Search Latency gains on mobile are lower than gains on desktop. In addition to differences between desktop and mobile environments and usage, the lower gains are explained

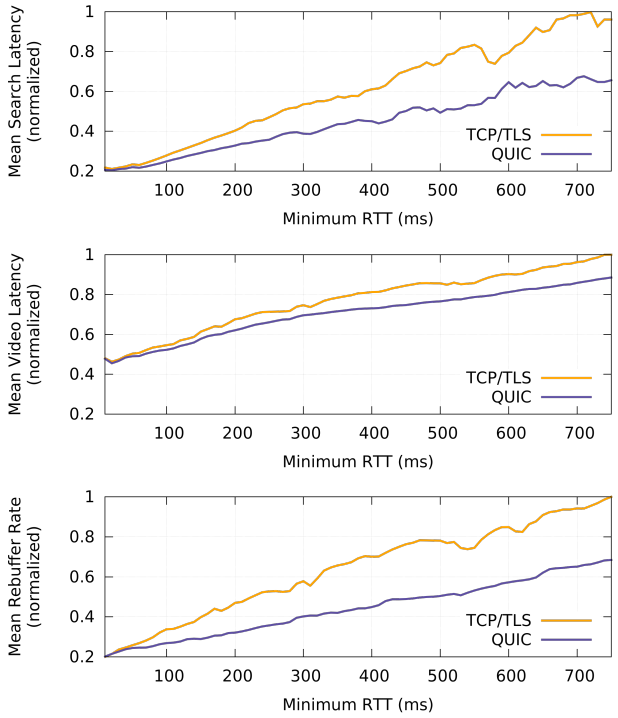

Figure 9: Comparison of $QUIC_g$ and TCP_g for various metrics, versus minimum RTT of the connection. The y-axis is normalized against the maximum value in each dataset. Presented data is for desktop, but the same trends hold for mobile as well. The x-axis shows minimum RTTs up to 750 ms, which was chosen as reasonable due to Figure 8: 750 ms encompasses over 95% of RTTs and there is no information gained by showing more data.

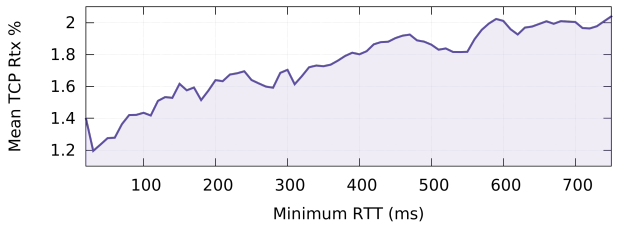

Figure 10: Average TCP retransmission rate versus minimum RTT observed by the connection. These results were gathered from video playbacks. We note that this graph shows the retransmission rate averaged within 10 ms RTT buckets, and the actual rate experienced by a connection can be much higher [22]. Across all RTTs, retransmission rates are 0%, 2%, 8% and 18% at the 50th, 80th, 90th and 95th percentiles.

in part by the fact that QUIC connections established by the mobile app only achieve a 68% 0-RTT handshake rate on average—a 20% reduction in successful 0-RTT handshake rate as compared to desktop—which we believe is due to two factors. Recall from Section 3.1 that a successful 0-RTT handshake requires both a valid server config and a valid source address token in a client's handshake message, both of which are cached at the client from a previous successful handshake. The source-address token is a server-encrypted blob containing the client's validated IP address, and the server config contains the server's credentials. First, when mobile users switch networks, their IP address changes, which invalidates the source-address token cached at the client. Second, different server configurations and keys are served and used across different data centers. When mobile users switch networks, they may hit a different data center where the servers have a different server config than that cached at the client. Analysis of server logs shows that each of these two factors contributes to about half of the reduction in successful 0-RTT handshakes.

Finally, we attribute the latency increase in $QUIC_g$ at the 1st and 5th percentiles to additional small costs in QUIC, including OS process scheduler costs due to being in user-space, which are a higher proportion of the total latency at low overall latencies. We discuss QUIC's limitations further in Section 6.8.

6.4 Video Latency

Video Latency for a video playback is measured as the time between when a user hits "play" on a video to when the video starts playing. To ensure smooth playbacks, video players typically buffer a couple seconds of video before playing the first frame. The amount of data the player loads depends on the bitrate of the playback. Table 1 shows that users in $QUIC_g$ experience decreased overall Video Latency for both desktop and mobile YouTube playbacks.

Figure 9 shows that Video Latency gains increase with client RTT, similar to Search Latency. An average of 85% of QUIC connections for video playback on desktop receive the benefit of a 0-RTT handshake, and the rest benefit from a 1-RTT handshake. As with Search Latency, QUIC loss recovery improvements may help Video Latency as client RTT increases.

QUIC benefits mobile playbacks less than desktop. The YouTube app achieves a 0-RTT handshake for only 65% of QUIC connections. Additionally, the app tries to hide handshake costs, by establishing connections to the video server in the background while users are browsing and searching for videos. This optimization reduces the benefit of QUIC's 0-RTT handshake, further reducing gains for mobile video in $QUIC_g$.

6.5 Video Rebuffer Rate

To ensure smooth playback over variable network connections, video players typically maintain a small playback buffer of video data. The amount of data in the buffer varies over time. If the player reaches the end of the buffer during playback, the video pauses until the player can rebuffer data. *Video Rebuffer Rate*, or simply *Rebuffer Rate* is the percentage of time that a video pauses during a playback to rebuffer data normalized by video watch time, where video watch time includes time spent rebuffering. In other words, Rebuffer Rate is computed as (Rebuffer Time) / (Rebuffer Time + Video Play Time).

Table 2 indicates that users in $QUIC_g$ experience reduced Rebuffer Rate on average and substantial reductions at higher percentiles. These results are qualitatively different from Search Latency and Video Latency since the contributing factors are different: Rebuffer Rate is largely insensitive to handshake latency. It is instead influenced by loss-recovery latency, since missing data on an audio or video stream can stall video playback. It is also influenced by the connection's overall throughput, which determines the rate at which video is delivered to the client.

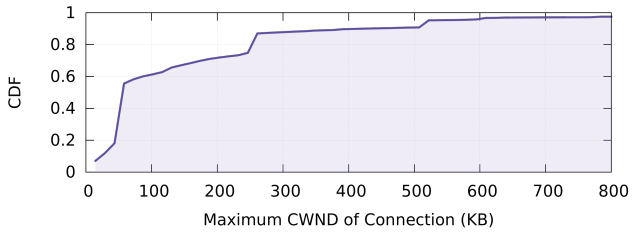

Figure 11: CDF of TCP connections where the server's maximum congestion window was limited by the client's maximum receive window. Data presented is for video playbacks from one week in March 2016. Data begins at about 16KB, which represents the smallest observed receive window advertisement.

Loss-Recovery Latency: Figure 9 shows Rebuffer Rate for video playbacks as a function of the client's minimum RTT to the video server. Benefits with QUIC increase with client RTT, which, as shown in Figure 10, also correspond to increases in network loss. The video player uses two TCP connections to the server for every playback fragment. Use of two connections causes TCP's loss detection to be slower: data and ACKs on one connection cannot assist in loss detection on the other connection, and there are two recovery tails, which increases the probability of incurring tail-recovery latency [7, 21]. Additionally, QUIC's loss-recovery improvements described in Section 3.4 appear to increase QUIC's resiliency to higher loss rates. As Figure 9 shows, both QUIC_g's and TCP_g's rebuffer rates increase at higher RTTs. However, as this figure also shows, QUIC_g's rebuffer rate increases more slowly than TCP_g's, implying that QUIC's loss-recovery mechanisms are more resilient to greater losses than TCP.

Connection Throughput: Connection throughput is dictated by a connection's *congestion window*, as estimated by the sender's congestion controller, and by its *receive window*, as computed by the receiver's flow controller. For a given RTT, the maximum send rate of a connection is directly limited by the connection's maximum achievable congestion and receive windows.

The default initial connection-level flow control limit advertised by a QUIC client is 15MB, which is large enough to avoid any bottlenecks due to flow control. Investigation into client-advertised TCP receive window however paints a different picture: TCP connections carrying video data can be limited by the client's receive window. We investigated all video connections over TCP for a week in March 2016, specifically looking into connections that were receive-window-limited—where the congestion window matched the advertised receive window—and the results are shown in Figure 11. These connections accounted for 4.6% of the connections we examined. The majority of these constrained connections were limited by a maximum receive window advertisement of 64 KB, or roughly 45 MTU-sized packets. This window size limits the maximum possible send rate, constraining the sender when the path has a large RTT and during loss recovery. We believe that the low advertised maximum receive window for TCP is likely due to the absence of window scaling [37], which in turn may be caused by legacy clients that lack support for it and/or middlebox interference.

Rebuffer rates can be decreased by reducing video quality, but QUIC playbacks show improved video quality as well as a decrease in rebuffers. As a measure of video quality, we consider the fraction of videos that were played at their *optimal rate*: the format best suited for the video viewport size and user intent. Among video playbacks that experienced no rebuffering, this fraction is the same for users in QUIC_g as those in TCP_g. Among playbacks that experienced non-zero rebuffering, QUIC increased the number of videos played at their optimal rates by 2.9% for desktop and by 4.6% for mobile playbacks.

QUIC's benefits are higher whenever congestion, loss, and RTTs are higher. As a result, we would expect QUIC to benefit users most in parts of the world where congestion, loss, and RTTs are highest; we look into this thesis next.

6.6 Performance By Region

Differences in access-network quality and distance from Google servers result in RTT and retransmission rate variations for different geographical regions. We now look at QUIC's impact on Search Latency[10] and on Video Rebuffer Rate in select countries, chosen to span a wide range of network conditions.

Table 3 show how QUIC's performance impact varies by country. In South Korea, which has the lowest average RTT and the lowest network loss, QUIC_g's performance is closer to that of TCP_g. Network conditions in the United States are more typical of the global average, and QUIC_g shows greater improvements in the USA than in South Korea. India, which has the highest average RTT and retransmission rate, shows the highest benefits across the board.

QUIC's performance benefits over TLS/TCP are thus not uniformly distributed across geography or network quality: benefits are greater in networks and regions that have higher average RTT and higher network loss.

6.7 Server CPU Utilization

The QUIC implementation was initially written with a focus on rapid feature development and ease of debugging, not CPU efficiency. When we started measuring the cost of serving YouTube traffic over QUIC, we found that QUIC's server CPU-utilization was about 3.5 times higher than TLS/TCP. The three major sources of QUIC's CPU cost were: cryptography, sending and receiving of UDP packets, and maintaining internal QUIC state. To reduce cryptographic costs, we employed a hand-optimized version of the ChaCha20 cipher favored by mobile clients. To reduce packet receive costs, we used asynchronous packet reception from the kernel via a memory-mapped application ring buffer (Linux's PACKET_RX_RING). Finally, to reduce the cost of maintaining state, we rewrote critical paths and data-structures to be more cache-efficient. With these optimizations, we decreased the CPU cost of serving web traffic over QUIC to approximately twice that of TLS/TCP, which has allowed us to increase the levels of QUIC traffic we serve. We believe that while QUIC will remain more costly than TLS/TCP, further reductions are possible. Specifically, general kernel bypass [57] seems like a promising match for a user-space transport.

6.8 Performance Limitations

QUIC's performance can be limited in certain cases, and we describe the limitations we are aware of in this section.

[10] Video Latency trends are similar to Search Latency.

			% Reduction in Search Latency		% Reduction in Rebuffer Rate	
Country	Mean Min RTT (ms)	Mean TCP Rtx %	Desktop	Mobile	Desktop	Mobile
South Korea	38	1	1.3	1.1	0.0	10.1
USA	50	2	3.4	2.0	4.1	12.9
India	188	8	13.2	5.5	22.1	20.2

Table 3: Network characteristics of selected countries and the changes to mean Search Latency and mean Video Rebuffer Rate for users in $QUIC_g$.

Pre-warmed connections: When applications hide handshake latency by performing handshakes proactively, these applications receive no measurable benefit from QUIC's 0-RTT handshake. This optimization is not uncommon in applications where the server is known *a priori*, and is used by the YouTube app. We note that some applications, such as web browsers, cannot always pre-warm connections since the server is often unknown until explicitly indicated by the user.

High bandwidth, low-delay, low-loss networks: The use of QUIC on networks with plentiful bandwidth, low delay, and low loss rate, shows little gain and occasionally negative performance impact. When used over a very high-bandwidth (over 100 Mbps) and/or very low RTT connection (a few milliseconds), QUIC may perform worse than TCP. We believe that this limitation is due to client CPU limits and/or client-side scheduler inefficiencies in the OS or application. While these ranges are outside typical Internet conditions, we are actively looking into mitigations in these cases.

Mobile devices: QUIC's gains for mobile users are generally more modest than gains for desktop users. As discussed earlier in this section, this is partially due to the fact that mobile applications are often fine-tuned for their environment. For example, when applications limit content for small mobile-screens, transport optimizations have less impact. Mobile phones are also more CPU-constrained than desktop devices, causing CPU to be the bottleneck when network bandwidth is plentiful. We are actively working on improving QUIC's performance on mobile devices.

7 EXPERIMENTS AND EXPERIENCES

We now share lessons we learned during QUIC's deployment. Some of these involved experiments at scale, such as determining QUIC's maximum packet size. Others required deploying at scale, such as detecting the extent and nature of UDP blocking and throttling on the Internet. A few lessons were learned through failures, exemplified by our attempt to design and use FEC in QUIC. We also describe a surprising ecosystem response to QUIC's deployment: its rapid ossification by a middlebox vendor.

7.1 Packet Size Considerations

Early in the project, we performed a simple experiment to choose an appropriate maximum packet size for QUIC. We performed a wide-scale reachability experiment using Chrome's experimentation framework described in Section 4. We tested a range of possible UDP payload sizes, from 1200 bytes up to 1500 bytes, in 5 byte increments. For each packet size, approximately 25,000 instances of Chrome would attempt to send UDP packets of that size to an echo server on our network and wait for a response. If at least one response was received, this trial counted as a reachability success, otherwise it was considered to be a failure.

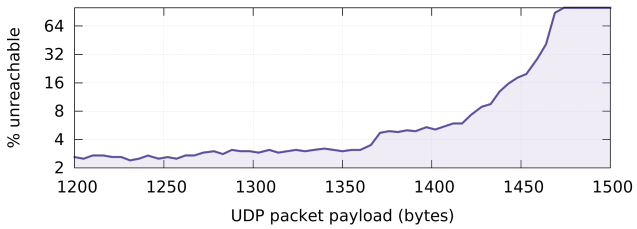

Figure 12: Unreachability with various UDP payload sizes. Data collected over 28 days in January 2014.

Figure 12 shows the percentage of clients unable to reach our servers with packets of each tested size. The rapid increase in unreachability after 1450 bytes is a result of the total packet size—QUIC payload combined with UDP and IP headers—exceeding the 1500 byte Ethernet MTU. Based on this data, we chose 1350 bytes as the default payload size for QUIC. Future work will consider path MTU discovery for QUIC [45].

7.2 UDP Blockage and Throttling

We used video playback metrics gathered in November 2016 to measure UDP blocking and throttling in the network. QUIC is successfully used for 95.3% of video clients attempting to use QUIC. 4.4% of clients are unable to use QUIC, meaning that QUIC or UDP is blocked or the path's MTU is too small. Manual inspection showed that these users are commonly found in corporate networks, and are likely behind enterprise firewalls. We have not seen an entire ISP blocking QUIC or UDP.

The remaining 0.3% of users are in networks that seem to rate limit QUIC and/or UDP traffic. We detect rate limiting as substantially elevated packet loss rate and decreased bandwidth at peak times of day, when traffic is high. We manually disable QUIC at our servers for entire Autonomous Systems (AS) where such throttling is detected and reach out to the operators running the network, asking them to either remove or at least raise their limits. Reaching out to operators has been effective—we saw a reduction in AS-level throttling from 1% in June 2015 to 0.3% in November 2016—and we re-enabled QUIC for ASes that removed their throttlers.

7.3 Forward Error Correction

Forward Error Correction (FEC) uses redundancy in the sent data stream to allow a receiver to recover lost packets without an explicit retransmission. Based on [21], which showed that single losses are common, we experimented with XOR-based FEC (simple parity) to

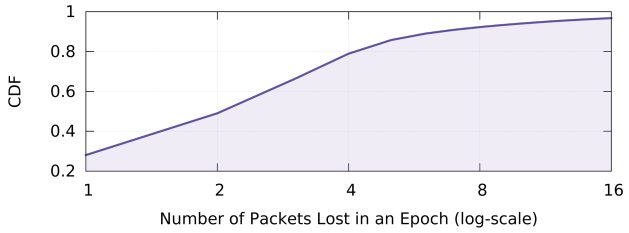

Figure 13: CDF of fraction of QUIC loss epochs vs. number of packet losses in the epoch. Data collected over one week in March 2016.

enable recovery from a single packet loss within a group. We used this simple scheme because it has low computational overhead, it is relatively simple to implement, and it avoids latency associated with schemes that require multiple packets to arrive before any can be processed.

We experimented with various packet protection policies—protecting only HTTP headers, protecting all data, sending an FEC packet only on quiescence—and found similar outcomes. While retransmission rates decreased measurably, FEC had statistically insignificant impact on Search Latency and increased both Video Latency and Video Rebuffer Rate for video playbacks. Video playback is commonly bandwidth limited, particularly at startup; sending additional FEC packets simply adds to the bandwidth pressure. Where FEC reduced tail latency, we found that aggressively retransmitting at the tail [17] provided similar benefits.

We also measured the number of packets lost during RTT-long loss epochs in QUIC to see if and how FEC might help. Our goal was to determine whether the latency benefits of FEC outweighed the added bandwidth costs. The resulting Figure 13 shows that the benefit of using an FEC scheme that recovers from a single packet loss is limited to under 30% of loss episodes.

In addition to benefits that were not compelling, implementing FEC introduced a fair amount of code complexity. Consequently, we removed support for XOR-based FEC from QUIC in early 2016.

7.4 User-space Development

Development practices directly influence robustness of deployed code. In keeping with modern software development practices, we relied heavily on extensive unit and end-to-end testing. We used a network simulator built into the QUIC code to perform fine-grained congestion control testing. Such facilities, which are often limited in kernel development environments, frequently caught significant bugs prior to deployment and live experimentation.

An added benefit of user-space development is that a user-space application is not as memory-constrained as the kernel, is not limited by the kernel API, and can freely interact with other systems in the server infrastructure. This allows for extensive logging and integration with a rich server logging infrastructure, which is invaluable for debugging. As an example, recording detailed connection state at every packet event at the server led us to uncover a decade-old Cubic quiescence bug [18]. Fixing this bug reduced QUIC's retransmission rates by about 30%, QUIC's CPU utilization by about 17%, and TCP's retransmission rates by about 20% [30].

Due to these safeguards and monitoring capabilities, we were able to iterate rapidly on deployment of QUIC modifications. Figure 14

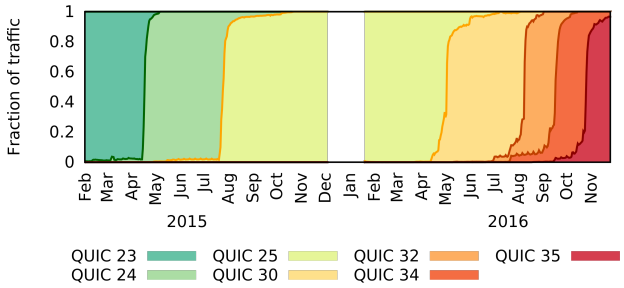

Figure 14: Incoming QUIC requests to our servers, by QUIC version.

shows versions used by all QUIC clients over the past two years. As discussed in Section 5, our ability to deploy security fixes to clients was and remains critically important, perhaps even more so because QUIC is a secure transport. High deployment velocity allowed us to experiment with various aspects of QUIC, and if found to not be useful, deprecate them.

7.5 Experiences with Middleboxes

As explained in Section 3.3, QUIC encrypts most of its packet header to avoid protocol entrenchment. However a few fields are left unencrypted, to allow a receiver to look up local connection state and decrypt incoming packets. In October 2016, we introduced a 1-bit change to the public flags field of the QUIC packet header. This change resulted in pathological packet loss for users behind one brand of firewall that supported explicit blocking of QUIC traffic. In previous versions of QUIC, this firewall correctly blocked all QUIC packets, causing clients to fall back to TCP. The firewall used the QUIC flags field to identify QUIC packets, and the 1-bit change in the flags field confounded this detection logic, causing the firewall to allow initial packets through but blocking subsequent packets. The characteristics of this packet loss defeated the TCP fallback logic described in Section 3.8. As a result, clients that were previously using TCP (since QUIC was previously successfully blocked) were now starting to use QUIC and then hitting a packet black-hole. The problem was fixed by reverting the flags change across our clients.

We identified the middlebox and reached out to the vendor. The vendor addressed the issue by updating their classifier to allow the variations seen in the flags. This fix was rolled out to their customers over the following month.

While we were able to isolate the problem to one vendor in this instance, our process of reaching out to them does not scale to all middleboxes and vendors. We do not know if other bits exposed by QUIC have been ossified by this or other middleboxes, and we do not have a method to answer this question at Internet scale. We did learn that middlebox vendors are reactive. When traffic patterns change, they build responses to these observed changes. This pattern of behavior exposes a "deployment impossibility cycle" however, since deploying a protocol change widely requires it to work through a huge range of middleboxes, but middleboxes only change behavior in response to wide deployment of the change. This experience reinforces the premise on which QUIC was designed: when deploying end-to-end changes, encryption is the only means available to ensure that bits that ought not be used by a middlebox are in fact not used by one.

8 RELATED WORK

SPDY [3] seeks to reduce web latency and has been subsumed by the HTTP/2 standard [8]. QUIC is a natural extension of this work, seeking to reduce latency further down the stack.

QUIC's design is closest to that of Structured Stream Transport (SST) [23]. Among other similarities, SST uses a channel identifier, uses monotonically increasing packet numbers, encrypts the transport header, and employs lightweight streams. QUIC builds on these design ideas. QUIC differs from SST in several, sometimes subtle, ways. For instance, while SST avoids handshake latency for subsequent streams, the first stream incurs it. QUIC avoids handshake latency on repeat connections to the same server, and includes version negotiation in the handshake. Stream multiplexing to avoid head-of-line blocking is not a new idea; it is present in SCTP [62], SST [23], and as message chaining in Minion [32]. QUIC borrows this design idea. QUIC also uses shared congestion management among multiple application streams, similar to SCTP, SST, and the Congestion Manager [7]. MinimaLT [51] was developed contemporaneously. It has a similar 0-RTT handshake, multiplexes application "connections" within an encrypted MinimaLT "tunnel", and performs congestion control and loss detection on the aggregate. MinimaLT additionally prevents linkability of a connection as it migrates from one IP address to another. This privacy-preserving feature is currently under consideration as QUIC evolves at the IETF.

Modifications to TCP and TLS have been proposed to address their handshake latencies. TCP Fast Open (TFO) [11, 53] addresses the handshake latency of TCP by allowing data in the TCP SYN segment to be delivered to a receiver on a repeat connection to the same server. TFO and QUIC differ in two key ways. First, TFO limits client data in the first RTT to the amount that fits within the TCP SYN segment. This limit is absent in QUIC, which allows a client to send as much data as allowed by the congestion and flow controllers in this initial RTT. Second, TFO is useful on repeat connections to a destination with the same IP address as the first connection. A common load balancing method employed by servers is to use multiple IP addresses for the same hostname, and repeat TCP connections to the same domain may end up at different server IP addresses. Since QUIC combines the cryptographic layer with transport, it uses 0-RTT handshakes with repeat connections to the same *origin*. Finally, while TFO is now implemented in major OSes (Windows, Linux, MacOS/iOS), its deployment is limited due to middlebox interference [50] and due to slow client OS upgrades.

QUIC's cryptographic handshake protocol was a homegrown protocol, but has been formally analyzed by various groups [20, 38, 44]. Facebook's Zero Protocol was directly derived from QUIC's cryptographic protocol [35]. TLS 1.3 [55], inspired in part by QUIC's handshake protocol [42], addresses the handshake latency of TLS 1.2, the currently deployed TLS version. Since TLS 1.3 now provides the latency benefits of QUIC's cryptographic handshake, IETF standardization work will replace QUIC's cryptographic handshake with TLS 1.3 [63].

9 CONCLUSION

QUIC was designed and launched as an experiment, and it has now become a core part of our serving infrastructure. We knew that wide deployment of a new UDP-based encrypted transport for HTTP was an audacious goal; there were many unknowns, including whether UDP blocking or throttling would be show-stoppers. Our experimentation infrastructure was critical in QUIC's deploy-measure-revise cycles, and it allowed us to build and tune a protocol suited for today's Internet.

We expect to continue working on reducing QUIC's CPU cost at both the server and the client and in improving QUIC performance on mobile devices. One of QUIC's most important features is its ability to be used as a platform for wide-scale experimentation with transport mechanisms, both at the server and at the client. Ongoing experimentation and work is continuing on several fronts. First, we are experimenting with connection migration to reduce latency and failures with various mobile applications. Later work may include implementation of general-purpose multipath [31, 54]. Second, we are experimenting with modern congestion controllers such as BBR [10] and PCC [16]. Third, we are working on using QUIC for WebRTC [4] and intend to explore avenues for better supporting real-time payloads.

The lessons we learned and described in this paper are transferable to future work on Internet protocols. Of the lessons, we'll reiterate a few important ones. First, developing and deploying networking protocols in user space brings substantial benefits, and it makes development, testing, and iteration cycles faster and easier. Second, layering enables modularity but often at the cost of performance, and re-designing and rewriting critical paths in the protocol stack is a useful exercise. Squashing the layers of HTTPS in QUIC allowed us to weed out inefficiencies in the HTTPS stack.

Finally, while a tussle between the endpoints and the network is expected and inevitable, it can only be resolved when all interested parties come to the table [13]. Previous attempts to deploy protocols that require any modification to network devices—ECN [41], SCTP [62], TCP Fast Open [11], and MPTCP [47, 54], to name a few—have unequivocally exposed the difficulties of incentivizing and achieving consensus on proposed network changes. As noted in [13], "the ultimate defense of the end to end mode is end to end encryption." Encryption forces the conversation among various parties and remains the sole guardian of the end-to-end principle.

ACKNOWLEDGMENTS

A project of this magnitude is not possible without a lot of hands. We thank Amin Vahdat, Assar Westerlund, Biren Roy, Chris Bentzel, Danner Stodolsky, Jeff Callow, Leonidas Kontothanassis, Mihai Dumitrescu, Mike Warres, Misha Efimov, Roberto Peon, Siddharth Vijayakrishnan, Tarun Bansal, Ted Hardie, and Yu-Ting Tseng for their work and support over the years.

We thank all groups at Google that helped deploy QUIC, especially Traffic Team.

We thank folks at Akamai, at Caddy (quic-go), and Christian Huitema for implementing and/or adopting QUIC early and providing incredibly valuable feedback. Without all of their help, QUIC would not be where it is today.

We thank Jeff Mogul, Matt Welsh, Stuart Cheshire, our shepherd Ankit Singla, and the anonymous SIGCOMM reviewers for reviewing previous versions of this paper. Without their help, this paper would have been an unreadable mess.

REFERENCES

[1] *Chromium QUIC Implementation*. https://cs.chromium.org/chromium/src/net/quic/.

[2] *IETF QUIC working group*. https://datatracker.ietf.org/wg/quic/.

[3] *SPDY: An experimental protocol for a faster web*. https://www.chromium.org/spdy/spdy-whitepaper.

[4] *The WebRTC Project*. https://webrtc.org.

[5] A. Barth. 2015. RFC 6454: The Web Origin Concept. *Internet Engineering Task Force (IETF)* (Dec. 2015).

[6] I. Arapakis, X. Bai, and B. Cambazoglu. 2014. Impact of Response Latency on User Behavior in Web Search. In *ACM SIGIR*.

[7] H. Balakrishnan, H. Rahul, and S. Seshan. 1999. An integrated congestion management architecture for Internet hosts. *ACM SIGCOMM* (1999).

[8] M. Belshe, R. Peon, and M. Thomson. 2015. RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2). *Internet Engineering Task Force (IETF)* (2015).

[9] M. Bishop. 2017. Hypertext Transfer Protocol (HTTP) over QUIC. *IETF Internet Draft, draft-ietf-quic-http* (2017).

[10] N. Cardwell, Y. Cheng, C. S. Gunn, V. Jacobson, and S. Yeganeh. 2016. BBR: Congestion-Based Congestion Control. In *ACM Queue*.

[11] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. 2016. RFC 7413: TCP Fast Open. *Internet Engineering Task Force (IETF)* (2016).

[12] D. Clark and D. Tennenhouse. 1990. Architectural Considerations For a New Generation of Protocols. In *ACM SIGCOMM*.

[13] D. Clark, J. Wroclawski, K. Sollins, and R. Braden. 2005. Tussle in cyberspace: defining tomorrow's internet. *IEEE/ACM Transactions on Networking (ToN)* (2005).

[14] J. Crowcroft and P. Oechslin. 1998. Differentiated End-to-end Internet Services Using a Weighted Proportional Fair Sharing TCP. *SIGCOMM Computer Communication Review* (1998).

[15] J. Dean and L. Barroso. 2013. The Tail at Scale. *Commun. ACM* (2013).

[16] M. Dong, Q. Li, D. Zarchy, B. Godfrey, and M. Schapira. 2015. PCC: Re-architecting Congestion Control for Consistent High Performance. In *USENIX NSDI*.

[17] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. 2013. Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses. (2013).

[18] E. Dumazet. 2015. tcp_cubic better follow cubic curve after idle period. (2015). https://github.com/torvalds/linux/commit/30927520dbae297182990bb21d08762bcc35ce1d.

[19] R. Fielding and J. Reschke. 2014. RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. *Internet Engineering Task Force (IETF)* (2014).

[20] M. Fischlin and F. Günther. 2014. Multi-Stage Key Exchange and the Case of Google's QUIC Protocol. In *ACM Conference on Computer and Communications Security (CCS)*.

[21] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. 2013. Reducing Web Latency: The Virtue of Gentle Aggression. *ACM SIGCOMM* (2013).

[22] T. Flach, P. Papageorge, A. Terzis, L. Pedrosa, Y. Cheng, T. Karim, E. Katz-Bassett, and R. Govindan. 2016. An Internet-wide Analysis of Traffic Policing. In *ACM SIGCOMM*.

[23] B. Ford. 2007. Structured Streams: A New Transport Abstraction. In *ACM SIGCOMM*.

[24] G. Linden. 2006. *Make Data Useful*. http://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt.

[25] I. Grigorik. 2013. Speed, Performance, and Human Perception. (2013). https://hpbn.co/primer-on-web-performance.

[26] S. Ha, I. Rhee, and L. Xu. 2008. CUBIC: A New TCP-friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review* (2008).

[27] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo. 2010. An Experimental Study of Home Gateway Characteristics. In *ACM IMC*.

[28] M. Honda, F. Huici, C. Raiciu, J. Araújo, and L. Rizzo. 2014. Rekindling network protocol innovation with user-level stacks. *ACM Computer Communication Review* (2014).

[29] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. 2011. Is It Still Possible to Extend TCP?. In *ACM IMC*.

[30] J. Iyengar. 2015. Cubic Quiescence: Not So Inactive. (2015). Presentation at IETF94, https://www.ietf.org/proceedings/94/slides/slides-94-tcpm-8.pdf.

[31] J. Iyengar, P. Amer, and R. Stewart. 2006. Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths. *IEEE/ACM Transactions on Networking (ToN)* (2006).

[32] J. Iyengar, S. Cheshire, and J. Greassley. 2013. Minion: Service Model and Conceptual API. *IETF Internet Draft, draft-iyengar-minion-concept-02* (2013).

[33] J. Iyengar and I. Swett. 2016. QUIC Loss Detection and Congestion Control. *IETF Internet Draft, draft-ietf-quic-recovery* (2016).

[34] J. Iyengar and M. Thomson. 2016. QUIC: A UDP-Based Multiplexed and Secure Transport. *IETF Internet Draft, draft-ietf-quic-transport* (2016).

[35] S. Iyengar and K. Nekritz. 2017. Building Zero protocol for fast, secure mobile connections. (2017). Facebook Post, https://code.facebook.com/posts/608854979307125/building-zero-protocol-for-fast-secure-mobile-connections/.

[36] J. Brutlag. 2009. *Speed Matters*. https://research.googleblog.com/2009/06/speed-matters.html.

[37] V. Jacobson, R. Braden, and D. Borman. 1992. RFC 1323: TCP extensions for high performance. *Internet Engineering Task Force (IETF)* (1992).

[38] T. Jager, J. Schwenk, and J. Somorovsky. 2015. On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 V1.5 Encryption. In *ACM Conference on Computer and Communications Security (CCS)*.

[39] P. Karn and C. Partridge. 1987. Improving Round-Trip Time Estimates in Reliable Transport Protocols. In *ACM SIGCOMM*.

[40] E. Kohler, M. Handley, and S. Floyd. 2006. Designing DCCP: Congestion Control Without Reliability. In *ACM SIGCOMM*.

[41] M. Kühlewind, S. Neuner, and B. Trammell. 2013. On the State of ECN and TCP Options on the Internet. In *Passive and Active Measurement Conference (PAM)*.

[42] A. Langley. 2015. QUIC and TLS. (2015). Presentation at IETF92, https://www.ietf.org/proceedings/92/slides/slides-92-saag-5.pdf.

[43] A. Langley and W. Chang. *QUIC Crypto*. http://goo.gl/OuVSxa.

[44] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru. 2015. How Secure and Quick is QUIC? Provable Security and Performance Analyses. In *IEEE Symposium on Security and Privacy*.

[45] M. Mathis and J. Heffner. 2007. RFC 4821: Packetization layer path MTU discovery. *Internet Engineering Task Force (IETF)* (2007).

[46] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. 1996. RFC 2018: TCP Selective Acknowledgment Options. *Internet Engineering Task Force (IETF)* (1996).

[47] O. Mehani, R. Holz, S. Ferlin, and R. Boreli. 2015. An early look at multipath TCP deployment in the wild. In *ACM HotPlanet*.

[48] M. Nottingham, P. McManus, and J. Reschke. 2016. RFC 7838: HTTP Alternative Services. *Internet Engineering Task Force (IETF)* (2016).

[49] M. Nowlan, N. Tiwari, J. Iyengar, S. Amin, and B. Ford. 2012. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS. In *USENIX NSDI*.

[50] C. Paasch. 2016. Network Support for TCP Fast Open. (2016). Presentation at NANOG 67, https://www.nanog.org/sites/default/files/Paasch_Network_Support.pdf.

[51] W Michael Petullo, Xu Zhang, Jon A Solworth, Daniel J Bernstein, and Tanja Lange. 2013. MinimaLT: Minimal-Latency Networking Through Better Security. In *ACM CCS*.

[52] L. Popa, A. Ghodsi, and I. Stoica. 2010. HTTP as the Narrow Waist of the Future Internet. In *ACM HotNets*.

[53] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. 2011. TCP Fast Open. In *ACM CoNEXT*.

[54] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. 2012. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *USENIX NSDI*.

[55] E. Rescorla. 2017. The Transport Layer Security (TLS) Protocol Version 1.3. *IETF Internet Draft, draft-ietf-tls-tls13* (2017).

[56] E. Rescorla and N. Modadugu. 2012. RFC 6347: Datagram Transport Layer Security Version 1.2. *Internet Engineering Task Force (IETF)* (2012).

[57] L. Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*.

[58] J. Rosenberg. 2008. UDP and TCP as the New Waist of the Internet Hourglass. *IETF Internet Draft, draft-rosenberg-internet-waist-hourglass-00* (2008).

[59] J. Roskind. 2012. QUIC: Design Document and Specification Rationale. (2012). https://goo.gl/eCYF1a.

[60] J. Salowey, H. Zhou, P. Eronen, and H. Tschofenig. 2008. RFC 5077: Transport Layer Security (TLS) Session Resumption without Server-Side State. *Internet Engineering Task Force (IETF)* (2008).

[61] Sandvine. 2016. Global Internet Phenomena Report. (2016).

[62] R. Stewart. 2007. RFC 4960: Stream Control Transmission Protocol (SCTP). *Internet Engineering Task Force (IETF)* (2007).

[63] M. Thomson and S. Turner. 2017. Using Transport Layer Security (TLS) to Secure QUIC. *IETF Internet Draft, draft-ietf-quic-tls* (2017).

[64] L. Zhang. 1986. Why TCP Timers Don't Work Well. In *ACM SIGCOMM*.