November 2022
Geoff Huston

Comparing TCP and QUIC

There is a common view out there that the QUIC transport protocol (RFC 9000) is just another refinement to the original TCP transport protocol [1, 2]. I find it hard to agree with this sentiment, and for me QUIC represents a significant shift in the set of transport capabilities available to applications in terms of communication privacy, session control integrity and flexibility. QUIC embodies a different communications model that makes intrinsically useful to many more forms of application behaviours. Oh, yes. It's also faster than TCP! In my opinion It's likely that over time QUIC will replace TCP in the public Internet. So, for me QUIC is a lot more than just a few tweaks to TCP. Here we will describe both TCP and QUIC and look at the changes that QUIC has bought to the transport table.

However, we should first do a brief recap of TCP.

What is TCP?

TCP is the embodiment of the end-to-end principle in the overall Internet architecture. All the functionality required to take a simple base of datagram delivery and impose upon this model an end-to-end signalling regime that implements reliability, sequencing, adaptive flow control, and streaming is embedded within the TCP protocol.

TCP is a bilateral full duplex protocol. That means that TCP is a two-party communications protocol that supports both parties simultaneously sending and receiving data within the context of a single TCP connection. Rather than impose a state within the network to support the connection, TCP uses synchronized state between the two end points, and much of the protocol design ensures that each local state transition is communicated to, and acknowledged by, the remote party without any mediation by the network whatsoever.

TCP is a *stream protocol*. The stream of data generated by the sender will be seen by the receiver in the precisely the same order as generated by the sender. TCP is a true streaming protocol, and application-level network operations are not transparent. Other transport protocols have explicitly encapsulated each application transaction; for every sender's *write*, there must be a matching receiver's *read*. In this manner, the application-derived segmentation of the data stream into a logical record structure is preserved across the communication. TCP explicitly does not preserve such an implicit structure of the data, so that there is no explicit pairing between *write* and *read* operations within the network protocol. For example, a TCP application may write three data blocks in sequence into the network connection, which may be collected by the remote reader in a single read operation. It is left to the application to mark the stream with its own record boundaries, if such boundaries exist in the data.

There is a rudimentary level of stream formatting permitted within TCP through the concept of *urgent data* in which the sender can mark the end of a data segment that the application wants to bring to the attention of the receiver. The TCP data segment that carries the final byte of the urgent data segment can mark this data point, and the TCP receiving process has the responsibility to pass this mark to the receiving application.

The TCP connection is identified by the hosts at both ends by a 5-tuple of protocol identifier, source IP address, source port, destination IP address, and destination port.

The setup of a TCP connection requires a three-way handshake, ensuring that both sides of the connection have an unambiguous understanding of the remote side's byte sequence values. The operation of the connection setup is as follows: The local system sends the remote end an initial sequence number to the remote port using a SYN packet. The remote system responds with an ACK of the initial sequence number and the remote end's initial sequence number in a response SYN packet. The local end responds with an ACK of this remote sequence number. These handshake packets are conventionally TCP packets without any data payload. At this point TCP shifts into a reliable data flow control mode of operation. (Figure 1)

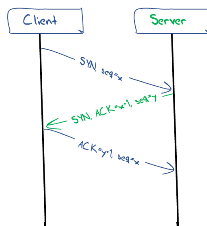

Figure 1 – TCP 3-way handshake

TCP is a *sliding window* protocol. The data stream is a sequence of numbered bytes. The sender retains a copy of all sent but as yet unacknowledged data in a local send buffer. When a receiver receives a data segment whose starting sequence number is the next expected data segment, then it will send an acknowledgement (ACK) back to the sender with the sequence number of the end of the received data segment. This allows the sender to discard the data in the local send buffer all data whose sequence number is less than this received ACK sequence and advance the send window. When the received data is out of order, it will send an acknowledgement (ACK) back to the sender with the sequence number of the last in-order received data. In addition, the ACK message includes the size of the receiver's available buffer size (receive window). The volume of unacknowledged data must be no larger than this receiver window size. The overall constraint is that at all points in time the sender should ensure that the volume of unacknowledged data in flight in the network is the smaller of the advertised receive window size and the total capacity of the local send buffer.

TCP is an *ACK-clocked* flow control protocol, in that within a static lossless mode of operation each received ACK packet indicates that a certain volume of data has been received at the receiver end (and hence has been removed from the network), and this is accompanied by an advertised receive window that then permits the sender to inject the same volume of data into the network as has been removed by the receiver. Hence, the sending rate is governed by the received ACK rate.

However, TCP is not necessarily aware of the available path capacity of the network and must do so by a control algorithm implemented at the sending end that attempts to establish a dynamic equilibrium between the flow volume of the TCP session and all other concurrent TCP sessions that have path segments in common with this session. The mode of operation of this flow control is not fixed in the TCP specification, and a number of flow control algorithms are in use. Many of these control algorithms use an induced instability in TCP through an approach of slow inflation of the sending window for each received ACK, and a rapid drop of the sending window in response to an indication of packet drop (3 duplicate ACKs). This process of sending rate inflation will stop when the send buffer is full, indicating that the sender cannot store any more sent data and must await ACKs before sending more data (send buffer rate limited). It will also stop sending rate inflation when the network cannot accept any further data in flight as the network's buffers are already full, so further sent data will cause packet loss, which will be signalled back to the sender by duplicate ACKs.

This process has a number of outcomes relevant to service quality. Firstly, TCP behaves adaptively, rather than predictively. The flow-control algorithms are intended to increase the data-flow rate to fill all

available network path capacity but also quickly back off if the available capacity changes due to network congestion or if a dynamic change occurs in the end-to-end network path that reduces this available capacity. Secondly, a single TCP flow across an otherwise idle network attempts to fill the network path with data, optimizing the flow rate (as long as the send buffer is larger than the network flow capacity). If a second TCP flow opens up across the same path, the two flow-control algorithms will interact so that both flows will stabilize to use approximately half of the available capacity per flow. More generally, TCP attempts to behave fairly, in that when multiple TCP flows are present the TCP algorithm is intended to share the network resource evenly across all active flows. A design tension always exists between the efficiency of network use and enforcing predictable session performance. With TCP, you do not necessarily have predictable throughput but gain a highly utilized and efficient network.

TCP and TLS

Transport layer security (TLS) [3] is handled as a further layer of indirection. Once the TCP 3-way handshake is complete, the parties then enter a TLS negotiation phase to allow authentication of the remote end of the connection, and to establish a session key that can be used to manage the encryption of the session data.

TLS commences with an exchange of credentials. In version 1.3 of TLS (the latest version of this protocol) the client sends a *client hello* message which includes which TLS version the client supports, the cipher suites supported, the name of the service, and a string of random bytes known as the *client random*. The server responds with a *server hello* message that contains the server's public key certificate, the *server random*, the chosen cipher suite and a digital signature of the hello messages. Both ends now know each other's random values and the chosen cipher suite, so both can generate a master secret for session encryption. The client sends a *finished* message to indicate that the secure symmetric session key is ready for use (Figure 2).

Earlier versions of TLS used additional packets in the hello exchange that increased the time to complete the TLS handshake.

QUIC

We can now move on to QUIC. The QUIC transport protocol [4] was apparently designed to address several issues with TCP and TLS, and in particular to improve the transport performance for encrypted traffic with faster session setup, and to allow for further evolution of transport mechanisms and explicitly avoid the emerging TCP ossification within the network.

It is a grossly inaccurate simplification, but at its simplest level QUIC is simply TCP encapsulated and encrypted in a UDP payload. To the external network QUIC has the appearance of a bi-directional UDP packet sequence where the UDP payload is concealed. To the endpoints QUIC can be used as a reliable full duplex data flow protocol. Even at this level, QUIC has a number of advantages over TCP. The first lies in the deliberate concealing of the transport flow control parameters from the network. The practice of deploying network middleware that rewrites TCP flow control values to impair the application's behaviour is not one that has enjoyed widespread support from the application layer, and hiding these flow control parameters from the network certainly prevents this practice. Secondly, it can allow the shift of responsibility for providing the transport protocol from the platform to the application. There has been a longstanding tension between the application and the platform. Changes to kernel-level TCP are performed via updates to the platform software, and often applications have to wait for the platform to make changes before they can take advantage of the change. For example, if an application wanted to use the TCP BBR flow control algorithm, then it would need to wait for a platform to integrate an implementation of the algorithm. By using a basic UDP interface to the platform's transport services, the entire flow control and encryption service can be lifted into the application itself, if so desired. There may be some performance penalty of shifting the transport code from the kernel to user space, but in return the application regains complete control of the transport service and allows it to operate in a mode which is not only independent of the platform but also hidden from the platform. This gives the application environment greater levels of control and agility.

However, QUIC is a whole lot more than just wrapping up TCP in UDP, so let's look at the QUIC protocol in a little more detail.

QUIC Connections

A QUIC connection is a shared state between a single client and a single server. QUIC uses the combination of two numbers, one selected by each end, to form a pair of connection IDs. This acts as a persistent identity for the QUIC session, which is used to ensure that changes in addressing at lower protocol layers (addresses or ports) will not cause packets to be delivered to a wrong recipient on the end host.

The primary function of a connection ID is to ensure that changes in addressing at lower protocol layers (IP source address and UDP source port numbers) do not cause packets for a QUIC connection to be dropped when there is a change in the external IP address of an endpoint. Each endpoint selects a connection ID using an implementation-specific (and perhaps deployment-specific) method that will allow received packets with that connection ID to be to be identified by the endpoint upon receipt to the appropriate QUIC connection instance.

After an endpoint receives a packet with the same connection ID and a different IP address or UDP port, it will perform verify the peer's ownership of the new address by sending a challenge frame containing random data to this new address and waiting for an echoed response with the same data. This challenge and response exchange is performed within the established crypto state, so it is intentionally challenging for an eavesdropper to hijack a session in this way. The two endpoints can continue to exchange data after the verification of the new address.

This is particularly useful in terms of negotiating various forms of Network Address Translation (NAT) behaviour. NATs are intentionally transport-aware and for TCP NATs will attempt to maintain a translation state until it observes the closing FIN exchange. UDP offers no such externally visible clues as to the ending of a session and NATs are prone to interpreting a silent period as a signal to tear down the NAT state. In such a case the next outbound packet might be assigned a new source address and/or UDP source port number by the NAT. It is also useful in terms of session resumption where the connection may have been idle for an extended period, and the NAT binding may have timed out. With TCP any change in any of the four address and port fields of the connection 5-tuple will cause the packet to be rejected as part of the TCP session. QUIC's use of a persistent connection ID permits the receiver to associate the new sender's address details with an existing connection.

This QUIC functionality of address agility can also be used in the context of network-level changes, such as a device switching between WiFi and cellular services while maintaining an active QUIC transport session.

QUIC Connection Handshake

A QUIC connection starts with a handshake which establishes a shared communications state and a shared secret using the QUIC-TLS protocol cryptographic handshake protocol in a single exchange. This protocol merges the TCP 3-way handshake and the TLS 1.3 3-way handshake into a single 3 packet exchange (Figure 2). This eliminates a full Round Trip Time (RTT) in the QUIC startup phase, which for short sessions is a very significant improvement in session performance.

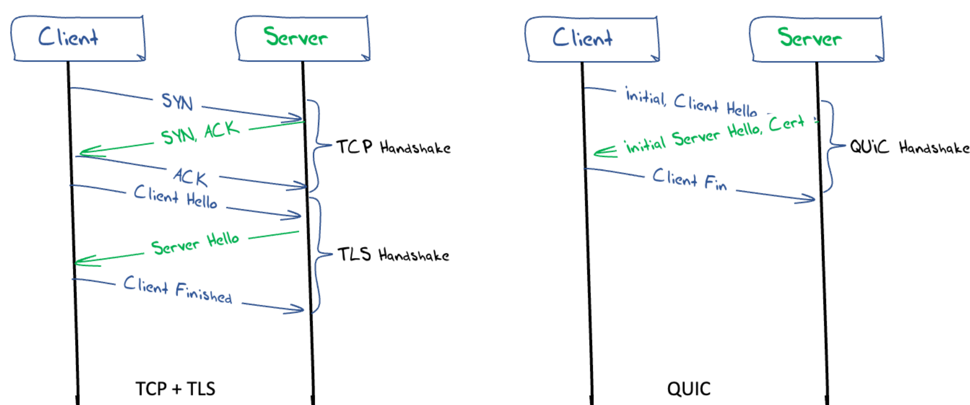

Figure 2 – TCP/TLS and QUIC Handshakes

QUIC also allows a client to send 0-RTT encrypted application data in its first packet to the server by reusing the negotiated parameters from a previous connection and a TLS 1.3 pre-shared key (PSK) identity issued by the server, although these 0-RTT data exchanges are not protected against replay attack.

Packets and Frames

The QUIC protocol sends *packets* along the connection. Packets are individually numbered in a 62-bit number space. There is no allowance for retransmission of a numbered packet. If data is to be retransmitted, it is done so in a new packet with the next packet number in sequence. That way there is a clear distinction between the reception of an original packet and a retransmission of the data payload.

Multiple QUIC packets can be loaded into a single UDP datagram. QUIC UDP datagrams must not be fragmented, and unless the end performs path MTU discovery, then QUIC assumes that the path can support a 1,200-byte UDP payload.

A QUIC client expands the payload of all UDP datagrams carrying Initial packets to at least the smallest allowed maximum datagram size of 1,200 bytes by adding padding frames to the Initial packet or by coalescing a set of Initial packets. The payload of all UDP datagrams carrying ack-eliciting Initial packets is padded to at least the smallest allowed maximum datagram size of 1,200 bytes. Sending UDP datagrams of this size ensures that the network path supports a reasonable Path Maximum Transmission Unit (PMTU), in both directions. Additionally, a client that expands Initial packets helps reduce the order of amplitude gain of amplification attacks caused by server responses toward an unverified client address.

QUIC packets are encrypted individually, so that the decryption process does not result in data decryption waiting for partially delivered packets. This is not generally possible under TCP, where the encryption records are in a byte stream and the protocol stack is unaware of higher-layer boundaries within this stream. The additional inference from this per-packet encryption is that it's a requirement that QUIC IP packets are not fragmented. QUIC implementations typically use a conservative choice in the maximum packet size so that IP packet fragmentation does not occur.

A QUIC receiver ACKs the highest packet number received so far, together with a listing of all received contiguous packet number blocks of lower-numbered packets if there are gaps in the received packet sequence. Because QUIC uses purpose-defined ACK frames, QUIC can code up to 256 such number ranges in a single frame, whereas TCP SACK has a limit of 3 such sequence number ranges. This allows QUIC to provide a more detailed view of packet loss and reordering, leading to higher resiliency against packet losses and more efficient recovery. Lost packets are not retransmitted. Data recovery is performed in the context of each QUIC stream.

QUIC Streams

A QUIC connection is further broken into *streams*. Each QUIC stream provide an ordered byte-stream abstraction to an application similar in nature to a TCP byte-stream. QUIC allows for an arbitrary number

of concurrent streams to operate over a connection. Applications may indicate the relative priority of streams.

Because the connection has already performed the end-to-end association and established the encryption context, streams can be established with minimal overhead. A single stream frame can open, pass data, and close down within a single packet, or a stream can exist for the entire lifetime of the connection.

By comparison, it is possible to multiplex a TCP session into streams, but all such multiplexed TCP streams share a single flow control state. If the TCP receiver advertises a zero-sized window to the sender, then all multiplexed streams are blocked in a TCP scenario.

Each QUIC stream is identified by a unique stream ID, where its two least significant bits are used to identify which endpoint initiated the stream and whether the stream is bidirectional or unidirectional. The byte stream is segmented to data frames, and the stream frame offset is equivalent to the TCP sequence number, used for data frame delivery ordering and loss detection and retransmission for reliable data delivery.

QUIC endpoints can decide how to allocate bandwidth between different streams, how to prioritize transmission of different stream frames based on information from the application. This ensures effective loss recovery, congestion control, flow control operations, which can significantly impact application performance.

QUIC Datagrams

In addition to reliable streams, QUIC also supports an unreliable but secured data delivery service with DATAGRAM frames, which will not be retransmitted upon loss detection [5]. When an application sends a datagram over a QUIC connection, QUIC will generate a DATAGRAM frame and send it in the first available packet. When a QUIC endpoint receives a valid DATAGRAM frame, it is expected that it would deliver the data to the application immediately. These DATAGRAM frames are not associated with any stream.

If a received packet contains only DATAGRAM frames, then the ACK frame can be delayed, as the sender will not retransmit a frame when there is an ACK failure in any case. This is not a reliable datagram service. If a sender detects that a packet containing a specific DATAGRAM frame might have been lost, the implementation may notify the application that it believes the datagram was lost. Similarly, if a packet containing a DATAGRAM frame is acknowledged, the implementation may notify the sender application that the datagram was successfully transmitted and received.

QUIC Frames

Each packet contains a sequence of *frames*. Frames have a frame type field and type-dependant data. The QUIC standard [4] defines 20 different frame types. They serve an analogous purpose to the TCP flags, carrying a control signal about the state of streams and the connection itself.

Frame types include padding, ping (or keepalive), ack frames for received packet numbers, which themselves contain ECN counts as well as ack ranges, as well as stream data frames, and datagram frames.

The larger organisation of QUIC connections, streams and frames is shown in Figure 3.

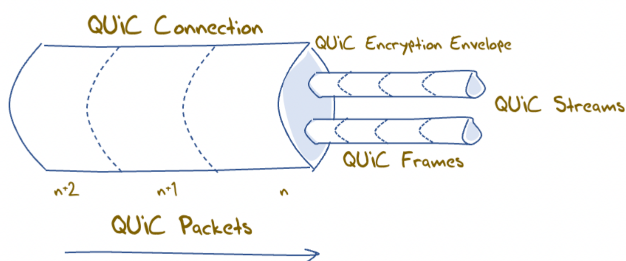

Figure 3 – QUIC logical organisation

QUIC Recovery and Flow Control

QUIC packets contain one or more frames. QUIC performs loss detection based on these packets, not on individual frames. For each packet that is acknowledged by the received, all frames carried in that packet are considered received. The packet is considered lost if that packet is unacknowledged when a later sent packet has been acknowledged, and when a loss threshold is met.

QUIC uses two thresholds to determine loss. The first is a *packet reordering threshold* ***t***. When packet ***x*** is acknowledged, then all unacknowledged packets with a number less than ***x – t*** are considered lost. The second is related to the QUIC-measured RTT interval, the *waiting time* ***w*** which is determined as a weight factor applied to the current estimated RTT interval. If the time of the most recent acknowledgement is ***t***, then all unacknowledged packets sent before time ***t – w*** will be considered lost.

For recovery, all frames in lost packets where the associated stream requires retransmission will be placed into new packets for retransmission. The lost packet itself is not retransmitted.

As with TCP's advertised receiver window, QUIC contains a mechanism to enable a QUIC receiver to control the maximum amount of data that a sender can send on an individual stream, and the maximum amount on all streams at any time. Also, as with TCP, the flow control algorithm to be used by reliable streams is not specified by QUIC, although one such sender-side congestion controller is defined in [6]. This is an algorithm similar to TCP's New Reno [7].

QUIC Issues

RPC support

IP hosts commonly support just two transport services, UDP and TCP. UDP is a simple datagram delivery service. Data encapsulated using UDP have no assured delivery. TCP, as we have seen, is a reliable streaming service. Any packet loss, or changes to the delivered packet sequence is repaired by the TCP protocol.

There is another model, namely the Remote Procedure Call (RPC) model. This model emulates the functionality of procedure calls, and rather than the byte stream model of TCP or the datagram model of UDP, the RPC model is a reliable request/reply model, where the reply is uniquely associated with the request. Perhaps the most well-known example today of an RPC framework is gRPC [8]. gRPC is based on an HTTP/2 platform, which implies that the framework is susceptible to head of line blocking as with any other TCP-based substrate.

The issue here is that a reliable byte stream is not the right abstraction for RPC, as the core of RPC is a request/reply paradigm, which is more aligned to a reliable messaging paradigm, with all that such a paradigm entails. A capable RPC framework needs to handle lost, mis-ordered, and duplicated messages, with an identifier space that can match requests and responses. The underlying message transport needs to handle messages of arbitrary size, which entails packetization adaptation within the transport.

The bidirectional stream framework is a reasonable match to the RPC communications model where each RPC can be is matched against an individual stream. The stream is reliable and sequenced. The data framing is not contained in QUIC, and it is still an application task to add a record structure to an RPC stream, if that is what is required. The invocation overhead is low in that the encrypted end-to-end connection is already established.

It certainly appears that HTTPS behaves much more like RPC than a reliable byte stream. That can benefit applications that run over HTTP(S), such as gRPC, and a set of RESTful APIs.

Load Balancing QUIC

In today's world of managing scale its very common to place a front-end load balancer across a number of servers. The load balancer in the TCP world typically categorizes packets as being in the same TCP

session because of a common 5-tuple value of protocol, IP addresses and port numbers, with the confident assurance that this value is stable for the life of the TCP session.

With QUIC there are no such assurances. The 5-tuple load balancing approach can work, but if the client is behind a NAT that performs what could be called "aggressive" rebinding then any such load balancing approach will be thrown. The reason why is that UDP does not provide session signalling to a NAT, so the is no a priori assurance that the NAT bindings (and the presented source address and port) will remain constant for the entire QUIC session. Now in theory IPv6 could invoke the Flow-ID to provide a proxy persistent field that remains constant for a flow, but the Flow-ID is of limited size and has no assurances of uniqueness, as well as evidence of highly variable treatment by IPv6 network infrastructure and end hosts.

This topic touches upon a major assumption in today's high-capacity server infrastructure on the public Internet. Data streams use TCP and the DNS uses UDP. Using UDP to carry sustained high-volume streams may not match the internal optimisations used in server content delivery networks.

DDOS Defence

The next issue here is exposure to DOS attacks. An attacker can send a large volume of packets to the server and cause the server to perform work to attempt to decrypt the packet. For this to be successful in TLS over TCP the attacker must make a reasonable guess of the TCP sequence number and window size for the packet to be accepted and passed to the TLS decoder. In QUIC there is no lightweight packet filter before the decoder is invoked.

On the other hand, the session encryption uses symmetric crypto algorithms, which are less of a load on the receiver to decode than asymmetric encryption. Is this enough of a difference to allow large scale QUIC platforms that are DOS resistant to be constructed? I'm unsure if there are clear answers here, but it seems that its part of the cost of having a more complete encryption framework, which in itself appears to be sorely needed on the public Internet.

Private QUIC

For private contexts can QUIC negotiate a "null" TLS encryption algorithm?

There is a bigger world out there beyond the public Internet and in many private data centre environments the overheads of encrypting and decrypting packets are overheads that may appear to be unnecessary. While QUIC can present some clear advantages in terms of suitability to complex application behaviours in the data centre that can leverage QUIC's multi-stream capability, the cost of encryption may be too high.

Of course, there is nothing stopping an implementation using a null encryption algorithm, but such an implementation could only talk to other implementations of itself. Strictly speaking, if you remove encryption, then it's no longer QUIC and won't interoperate with anything else that is QUIC.

QUIC and OpenSSL

It useful to ask that is QUIC has such clear advantages over TCP then why hasn't the adoption of QUIC been rapid. Metrics of QUIC use tend to point to a use rate of some 30% of web sessions (such as Cloudflare's Radar report [9]). However, if you alter the measurement to measure the extent to which browsers on end systems are capable of supporting a QUIC session, then the measurement jumps to 60% [10].

There are a couple of reasons why QUIC use is far lower than QUIC capability. The first is that the Chrome browser still relies on the content-level switch to QUIC, so that the client has to visit the site for the first time using HTTP/2 (TCP/TLS) and thereby receive an indication if the server can support QUIC, and then on the second visit the client may use QUIC. It's not quite as simple as this, as HTTP/2 uses persistent connection, so if the second visit is sufficiently close in time to the first, then the HTTP/2 session will remain open and still be use. The Safari browser is capable of using QUIC on first use because

it is triggered by the SVCB record in the DNS, but the market share of Safari is relatively small in comparison to QUIC.

The second reason lies in the web server environment. Many servers rely on the OpenSSL TLS library [11], and so far, (November 2022) OpenSSL does not include support for QUIC. QUIC is supported in BoringSSL [12], but as the notes for BoringSSL states, BoringSSL is a fork of OpenSSL that is designed to meet Google's needs, and while this works for Google, it may not work for everyone else. Google does not recommend that third parties depend on BoringSSL. There is also QuicTLS, a fork of OpenSSL supported by Akamai and Google [13]. This fragmentation of OpenSSL is not exactly helpful and the result is that many server environments are waiting for OpenSSL to incorporate a QUIC library. This effort was delayed by the work on OpenSSL release 3.0.0, and then the OpenSSL folk announced their intention to provide a fully functional QUIC implementation, and this development of a new QUIC protocol stack may further delay QUIC support in OpenSSL by months, if not years. This may well be the major impediment behind the very large scale deployment of QUIC in the guise of HTTP/3 across the Internet.

Conclusions

There are a few conclusions we can draw from this effort with QUIC.

Any useful public communications medium needs to safeguard the privacy and integrity of the communications that it carries. The time when open protocols represented an acceptable compromise between efficiency, speed and privacy are over and these days all network transactions in the public Internet need to be protected by adequate encryption. QUIC's model of wrapping a set of transactions between a client and a server in a single encryption state represents a sensible design decision.

Encryption is no longer an expensive luxury, but a required component for all transactions over the public Internet. The added imposition is that adding encryption into a network transaction should impose no additional performance penalty in terms of speed and responsiveness.

Network transactions come in many forms, and TCP and UDP tend to represent two ends of a relatively broad spectrum. UDP is just too susceptible to abuse so we've heaped everything onto TCP. The issue is TCP was designed as an efficient single streaming protocol and retro-fitting multiple sessions, short transactions, shared congestion state and shared encryption state have proved to be extremely challenging.

Applications are now dominant in the Internet ecosystem, while platforms and networks are being commoditised. We are seeing losing patience with platforms providing common transport services for the application that they host, and a new model where the application comes with its own transport service. This is not just the HTTP client/server model but has been extended into application-specific DNS name resolution with DNS over HTTPS. It's highly likely that this trend will continue for the moment.

Taking an even broader perspective, the context of the Internet's success lay in shifting the responsibility for providing service from the network to the end system. This allowed us to make more efficient use of the common network substrate and push the cost of this packetization of network transactions over to end systems. It shifted the innovation role from the large and lumbering telco operators into the more nimble world of platform software. The success of Microsoft with its Windows product was not an accident by any means. QUIC takes this one step further, and pushes the innovation role from platforms to applications, just at the time when platforms are declining in relative importance within the ecosystem. From such a perspective the emergence of an application-centric transport model that provides faster services, a larger repertoire of transport models and encompassing comprehensive encryption was an inevitable development.

References

[1] Eddy, W., Ed.,, “Transmission Control Protocol (TCP)”, RFC 9293, August 2022, <https://www.rfc-editor.org/info/rfc9293>.

[2] Postel, J., Ed., “Transmission Control Protocol”, RFC 793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[3] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

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

[5] Pauly, T., Kinnear, E., and D. Schinazi, "An Unreliable Datagram Extension to QUIC", RFC 9221, March 2022, <https://www.rfc-editor.org/info/rfc9221>.

[6] Iyengar, J., Ed., and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, May 2021, <https://www.rfc-editor.org/info/rfc9002>.

[7] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, April 2012, <https://www.rfc-editor.org/info/rfc6582>.

[8] gPRC – A cross-platform open source RPC framework <https://grpc.io/>.

[9] Cloudflare Rader, retrieved 1 November 2022, <https://radar.cloudflare.com/>.

[10] APNIC Labs, Quic Usage Report, retrieved 1 November 2022, <https://stats.labs.apnic.net/quic>

[11] OpenSSL a library for secure communication, retrieved 1 November 2022, <https://www.openssl.org>.

[12] BoringSSL, an open source fork of the OpenSSL library operated by Google for internal use, retrieved 1 November 2022, <https://boringssl.googlesource.com/boringssl/>.

[13] QuicTLS, an open source fork of the OpenSSL library developed by Akamai and Microsoft as an interim QUIC API, retrieved 1 November 2022, <https://github.com/quictls/openssl>.

Disclaimer

The above views do not necessarily represent the views or positions of the Asia Pacific Network Information Centre.

Author

Geoff Huston AM, B.Sc., M.Sc., is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region.

www.potaroo.net