The ISP Column
A monthly column on things Internet

March 2019
Geoff Huston

A quick look at QUIC

Quick UDP Internet Connection (QUIC) is a network protocol initially developed and deployed by Google, and now being standardized in the Internet Engineering Task Force. In this article we'll take a quick tour of QUIC, looking at what goals influenced its design, and what implications QUIC might have on the overall architecture of the Internet Protocol.

QUIC is not exactly a recent protocol, as the protocol appears to have been developed by Google in 2012, and initial public releases of this protocol were included in Chromium version 29, released in August 2013. QUIC is one of many transport layer network protocols that attempt to refine the basic operation of IP's *Transmission Control Protocol* (TCP).

Why are we even thinking about refining TCP?

TCP is now used in billions of devices and is perhaps the most widely adopted network transport protocol that we've witnessed so far. Presumably, if this protocol wasn't fit for general use, then we would have moved on and adopted some other protocol or protocols instead. But the fact is that TCP is not only good enough for a broad diversity of use cases, but in many cases, TCP is incredibly good at its job. Part of the reason for TCP's longevity and broad adoption is TCP's incredibly flexibility. The protocol can support a diverse variety of uses, from micro-exchanges to gigabyte data movement, transmission speeds that vary from hundreds of bits per second to tens and possibly hundreds of gigabits per second. TCP is undoubtedly the workhorse of the Internet. But even so, there is always room for refinement. TCP is put to many different uses and the design of TCP represents a set of trade-offs that attempt to be a reasonable fit for many purposes but not necessarily a truly ideal fit for any particular purpose.

One of the aspects of the original design of the Internet Protocol suite was that of elegant brevity and simplicity. The specification of TCP [1] is not a single profile of behavior that has been cast into a fixed form that was chiseled into the granite slab of a rigid standard. TCP is malleable in a number of important ways. Numerous efforts over the years have shown that it is possible to stay within the standard definition of TCP, in that all the packets in a session use the standard TCP header fields in mostly conventional ways, but also to create TCP implementations that behave radically differently from each other. TCP is an example of a conventional sliding window positive acknowledgement data transfer protocol. But while this is what the standard defines, there are some undefined aspects of the protocol. Critically, the TCP standard does not strictly define how the sender can control the amount of data in flight across the network to strike a fair balance between this data flow across the network and all other flows that coincide across common network path elements. There is a general convention these days in TCP to adopt an approach of slowly increasing the amount of data in flight while there are no visible errors in the data transfer (as shown by the stream of received acknowledgement packets) and quickly responding to signals of network congestion (interpreted as network packet drop, as shown by duplicate acknowledgements received by the sender) by rapidly decreasing the sending rate. Variants of TCP use different controls to manage this "slow increase" and "rapid drop" behavior [2] and may also use different signals to control this data flow, including measurements of end-to-end delay, or inter-packet jitter (such as the recently published BBR protocol [3]). All of these variants still manage to fit with the broad parameters of what is conventionally called TCP.

It is also useful to understand that most of these rate control variants of TCP need only be implemented on the data sender (the "server" in a client/server environment). The common assumption of all TCP implementations is that clients will send a TCP ACK packet on both successful receipt of in-sequence data and on receipt of out-of-sequence data. It is left to the server's TCP engine to determine how the received ACK stream will be applied to refine the sending TCP's internal model of network capability and how it will modify its subsequent sending rate accordingly. This implies that deployment of new variants of TCP flow control are essentially based on deployment within service delivery platforms and does not necessarily imply changing the TCP implementations in all the billions of clients. This factor of server-side control of TCP behavior also contributes to the flexibility of TCP.

But despite this considerable flexibility, TCP has its problems, particularly with web-based services. These days most web pages are not simple monolithic objects. They typically contain many separate components, including images, scripts, customized frames, style sheets and similar. Each of these is a separate web "object" and if you are using a browser that is equipped the original implementation of HTTP each object will be loaded in a new TCP session, even if they are served from the same IP address. The overheads of setting up both a new TCP session and a new *Transport Layer Security* (TLS) [4] session for each distinct web object within a compound web resource can become quite significant, and the temptations to re-use an already established TLS session for multiple fetches from the same server are close to overwhelming. But this approach of multiplexing a number of data streams within a single TCP session also has its attendant issues. Multiplexing multiple logical data flows across a single session can generate unwanted inter-dependencies between the flow processors and may lead to *head of line blocking* situations, where a stall in the transfer of the currently active stream blocks all queued fetch streams. It appears that while it makes some logical sense to share a single end-to-end security association and a single rate-controlled data flow state across a network across multiple logical data flows, TCP represents a rather poor way of achieving this outcome. The conclusion is that if we want to improve the efficiency of such compound transactions by introducing parallel behaviors into the protocol we need to look beyond TCP.

Why not just start afresh and define a new transport protocol that addresses these shortcomings of TCP? The answer is simple: NATs!

Network Address Translation and Transport Protocols

The original design of IP allowed for a clear separation between the network element that allowed the network to accept an IP packet and forward it onto its intended destination (the "Internet" part of the IP protocol suite) and the end-to-end transport protocol that enabled two applications to communication via some form of "session". The transport protocol field in the IPv4 packet header and the Next Header field of the IPv6 packet header uses an 8-bit field to identify the end-to-end protocol. This clear delineation between network and host parts of an IP packet header design assumes that the network has no intrinsic need to "understand" what end-to-end protocol was being used within a packet. At the network level the protocol architecture asserts that these packets are all stateless datagrams and should be treated identically by each network switching element. Ideally an IP packet switch will not differentiate in its treatment of packets depending on the inner end-to-end protocol. (It all this sounds somewhat dated these days it's because it is a somewhat dated view of the network, and these days many network elements reach into the supposed host part of a packet header. As a simple example, think of flow-aware traffic load balancing where in order to preserve packet order within a TCP stream the load balancer will use parts of the TCP header to identify packets that belong to the same logical flow.)

There are some 140 protocols listed in the IP protocol field registry [5]. TCP and UDP are just two of these protocols (protocol values 6 and 17 respectively) and, in theory at any rate, there is room for a least 100 more. In the public Internet the story is somewhat different. TCP and UDP are widely accepted protocols, and ICMP (protocol 2) is generally accepted, but little else. How did this happen?

NATs changed the assumption about network devices not looking inside the packet (well, to be precise, *port-translating NATs* changed that assumption). NATs are network devices that look inside the IP packet and re-write the port addresses used by TCP and UDP [6]. What if an IP packet contains an end-to-end transport protocol identifier value that is neither TCP or UDP? Most NATs will simply drop the packet, on the basis of a security paradigm that "what you don't recognize is likely to be harmful." The pragmatic result is that NATs have limited an application's choice of transport protocols in the public Internet to just two: TCP and UDP.

If the aim is to deploy a new transport protocol but not confuse active network elements that are expecting to see a conventional TCP or UDP header, then how can this be achieved? This was the challenge faced by the developers of QUIC.

QUIC over UDP

The solution chosen by QUIC was a UDP-based approach.

UDP is a minimal framing protocol that allows an application to access the basic datagram services offered by IP. Apart from the source and destination port numbers, the UDP header adds a length header and a checksum that covers the UDP header and UDP payload. It essentially an abstraction of the underlying datagram IP model with just enough additional information to allow an IP protocol stack to direct an incoming packet to an application that has bound itself to a nominated UDP port address. If TCP is an overlay across the underlying IP datagram service, then it's a small step to think about positioning TCP as a overlay in an underlying UDP datagram service.

Using our standard Internet model QUIC is, strictly speaking, a datagram transport application. An application that uses the QUIC protocol sends and receives packets using UDP port 443.

Technically, this is a very small change to an IP packet, adding just 8 bytes to the IP packet by placing a UDP header between the IP and TCP packet headers (Figure 1). The implications of this change are far more significant than these 8 bytes would suggest. However, before we consider these implications, let's look at some QUIC services.

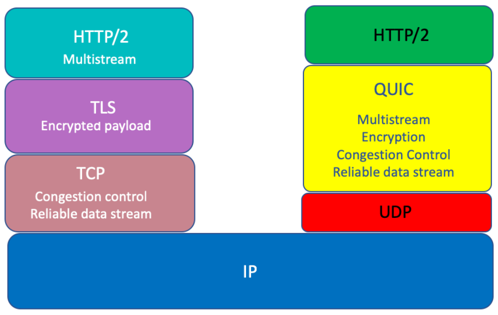

Figure 1 – The QUIC Protocol Architecture

Not only does the use of a UDP "shim" provide an essential protection of QUIC against NATs' enforcement of TCP and UDP as the only viable transport protocols, there is another aspect of QUIC which is equally important in today's Internet. The internal architecture of host systems has changed little from the 1970's. An operating system provides a consistent abstraction of a number of functions for an application. The operation system not only looks after scheduling the processor and managing memory, but also manages the local file store, and critically for QUIC, provides the networking protocol stack. The operating system contains the device drivers that implementation a consistent view of i/o devices, and also contains the implementations of the network protocol stack up to and including the transport protocol. The abstracted interface provided to the application may not be completely identical for all operating systems, but its sufficiently similar that with a few cosmetic changes an application can be readily ported to any platform with an expectation that not only will the application drive the network as expected, but do so in a standard manner such that it will seamlessly interoperate with any other application instance that is a standard-compliant implementation of the same function. Operating System network libraries not only relieve applications of the need to re-implement the network protocol stack, but assist in the overall task of ensuring seamless interoperation within a diverse environment.

However, such consistent functionality comes at a cost, and in this case the cost is resistance to change. Adding a new transport protocol to all operating systems, and to all package variants of all operating systems is an incredibly daunting task these days. As we have learned from the massive efforts to clean up various security vulnerabilities in operating systems, getting the installed base of systems to implement change suffers from an incredibly static and resistant tail of laggards!

Applications may not completely solve this issue, but they appear to have a far greater level of agility to self-apply upgrades. This means that if an application chooses to use its own implementation of networking protocols it has a greater level of control over the implementation, and need not await the upgrade cycle of third party operating system upgrades to apply changes to the code base. Web browser clients are a good example of this, where many functions are folded into the application in order to provide the desired behavior.

In the case of QUIC the transport protocol code is lifted into the application and executed in user space rather than an operating system kernel function. This may not be as efficient as a kernel implementation of the function, but the gain lies in greater flexibility and control by the application.

QUIC and the Connection ID

If the choice of UDP as the visible end-to-end protocol for QUIC was a choice dictated by the inflexibility of the base of deployed NAT devices in the public Internet, and their collective inability to accommodate new protocols, the handling of UDP packets by NATs have further implications for QUIC.

NATs maintain a *translation table*. In the most general model, a NAT takes the 5-tuple of incoming packets, using the destination and source IP addresses, the destination and source port addresses and the protocol field and perform a lookup into the table, and finds the associated translated fields. The packet's address headers are rewritten to these new values, checksums are recomputed, and the packet is passed onward. Certain NAT implementations may use variants of this model. For example, some NATs only use the source IP address and port address on *outbound* packets as the lookup key, and the corresponding destination IP address and port address in *incoming* packets.

Typically, the NAT will generate a new translation table entry when a triggering packet is passed from the *inside* to the *outside* and will subsequently remove the table entry when the NAT assumes that the translation is no longer needed. For TCP sessions it is possible to maintain this translation table quite accurately. New translation table entries are created in response to *outbound* TCP SYN connection establishment packets and removed either when the NAT sees the TCP FIN exchange or in response to a TCP RST packet or when the session is idle for an extended period.

UDP packets do not have these clear packet exchanges to start and stop sessions, so NATs need to make some assumptions. Most NATs will create a new translation table entry when it sees an outbound UDP packet that has not matched any existing translation table. The entry will be maintained for some period of time (as determined by the NAT) and will then be removed if there are no further packets that match the session signature. Even when there are further matching UDP packets the NAT may use an overall UDP session timer and remove the NAT entry after some pre-determined time interval.

For QUIC and NATs this is a potential problem. The QUIC session is established between a QUIC server on UDP port 443 and the NAT-generated source address and port. However, at some point in the session lifetime the NAT may drop the translation table entry, and the next outbound client packet will generate a new translation table entry and that entry may use a different source address and port. How can the QUIC server recognize that this next received packet, with its new source address and source port number is actually part of an existing QUIC session?

QUIC uses the concept of *connection identifiers* (connection IDs). Each endpoint generates connection IDs that will allow received packets with that connection ID to be routed to the process that is using that connection ID. During QUIC version negotiation these connection IDs are exchanged, and thereafter each sent QUIC packet includes the current connection ID of the remote party.

This form of semantic distinction between the identity of a connection to an endpoint and the current IP address and port number that is used by QUIC is similar to the Host Identity Protocol (HIP) [7]. This protocol also used a constant endpoint identifier that allowed a session to survive changes in the endpoint IP addresses and ports.

QUIC Streams

TCP provided the abstraction of a reliable order byte stream to applications. QUIC provides a similar abstraction to the application, termed within QUIC as *streams*. The essential difference here is that TCP implements a single behavior, while a single QUIC session can support multiple streams profiles.

Bidirectional streams place the client and server transactions into a matched context, as is required for the conventional request/response transactions of HTTP/1. A client would be expected to open a bidirectional stream with a server and then issue a request in a stream which would generate a matching response from the server. It is possible for a server to initiate a bidirectional *push stream* to a client, which contains a response without an initial request. Control information is supported using unidirectional *control streams*, where one side can pass a message to the other as soon as they are able. An underlying *unidirectional stream* interface, used to support control streams, is also exposed to the application.

Not only can QUIC support a number of different stream profiles, but QUIC can support different stream profiles within a single end-to-end QUIC session. This is not a novel concept of course, and the HTTP/2 protocol is a good example of an application-level protocol adding multiplexing and stream framing in order to carry multiple data flows across a single transport data stream. However, a single TCP transport stream as used by HTTP/2 may encounter *head of line blocking* where all overlay data streams fate-share across a single TCP session. If one of the streams stalls, then it's possible that all overlay data streams will be affected and may stall as well.

QUIC allows for a slightly different form of multiplexing where each overlay data stream can use its own end-to-end flow state, and a pause in one overlay stream does not imply that any other simultaneous stream is affected.

Part of the reason to multiplex multiple data flows between the same two endpoints in HTTP/2 was to reduce the overhead of setting up a TLS security association for each TCP session. This can be a major issue when the individual streams are each sending a small object, and it's possible to encounter a situation where the TCP and TLS handshake component of a compound web object fetch dominates both the total download time and the data volume.

QUIC pushes the security association to the end-to-end state that is implementation as a UDP data flow, so that streams can be started in a very lightweight manner because they essentially reuse the established secure session state.

QUIC Encryption

As is probably clear from the references to TLS already, QUIC uses end-to-end encryption. This encryption is performed on the UDP payload, so once the TLS handshake is complete very little of the subsequent QUIC packet exchange is in the clear (Figure 2).

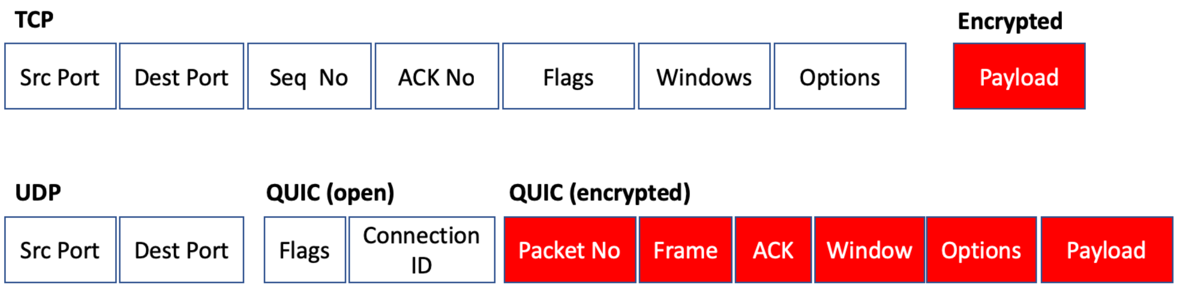

Figure 2 – Comparison of TCP and TLS with QUIC

What is exposed in QUIC are the *public flags*. This initial part of a QUIC packet consists of the connection ID, allowing the receiver to associate the packet with an endpoint without decrypting the entire packet. The QUIC version is also part of the public flag set. This is used in the initial QUIC session establishment and can be omitted thereafter.

The remainder of the QUIC packet are *private flags* and the payload. These are encrypted and are not directly visible to an eavesdropper. This private section includes the packet sequence number. This field used to detect duplicate and missing packets. It also includes all the flow control parameters, including window advertisements.

This is one of the critical differences between TCP and QUIC. With TCP the control parts of the protocol are in the clear, so that a network element would be able to inspect the port addresses (and infer the application type), as well as the flow state of the connection. Connection of a sequence of such TCP packets, even if only looking at the packets flowing in one direction within the connection would allow the network element to infer the round-trip time and the data transmission rate. And, like a NAT, manipulation of the receive window in the ACK stream would allow a network element to apply a throttle to a connection and reduce the transfer rate in a manner that would be invisible to both endpoints. By placing all of this control information inside the encrypted part of the QUIC packet ensures that no network element has direct visibility to this information, and no network element can manipulate the connection flow.

One could take the view that QUIC enforces a perspective that was assumed in the 1980's. This is that the end-to-end transport protocol is not shared with the network. All the network 'sees' are stateless datagrams, and the endpoints can safely assume that the information contained in the end-to-end transport control fields is carried over the network in a manner that protects it from third party inspection and alteration.

QUIC and IP Fragmentation

The short answer is "no!" QUIC packets cannot be fragmented.

The way this is achieved is by having the QUIC HELLO packet be padded out to the maximal packet size, and not completing the initial HELLO exchange if the maximally-sized packet is fragemented.

For IPv4 the QUIC maximum QUIC packet is 1,350 bytes. Adding 8 bytes for the UDP header, 20 bytes for IPv4, and 14 bytes for the Ethernet frame this means that QUIC packet on Ethernet are 1,392 packets in size. There is no particular rationale for this choice of 1,350 other than the results of empirical testing on the public Internet.

For IPv6 the QUIC maximum packet size is reduced by 20 bytes to 1,330. The resultant ethernet packet is still 1,392 bytes because of the larger IPv6 IP packet header.

What happens if the network path has a smaller MTU than this value? The answer is in the next section.

QUIC and TCP

QUIC is not intended as a replacement for TCP. Indeed, QUIC relies on the continued availability of TCP.

Whenever QUIC encounters a fatal error, such as fragmentation of the QUIC HELLO packet, the intended response from QUIC is to shut down the connection. As QUIC itself lies in the application space not the kernel space, the client-side application can be directly informed of this closure of the QUIC connection and it can re-open a connection to the server using a conventional TCP transport protocol.

The implication is that QUIC does not necessarily have to have a robust response for all forms of behavior, and when QUIC encounters a state where QUIC has no clear definition of the desired behavior it is always an option to signal a QUIC failure to the application. The failure need not be fatal to the application, as such a signal can trigger the application to repeat the transaction using a conventional TCP session.

I can QUIC, do you?

Unlike all other TCP services that use a dedicated TCP port address to distinguish itself from all other services, QUIC does not advertise itself in such a manner. That leaves a number of ways in which a server could potentially advertise itself as being accessible over QUIC.

One such possible path is the use of DNS service records (SRV) [7]. The SRV record can indicate the connection point for a named service using the name of the transport protocol and the protocol-specific service address. This may be an option for the future, but no such DNS service record has been defined for QUIC.

Instead, in keeping with QUIC's overall approach of loading up most of the service functionality into the application itself, a server that supports QUIC can signal its capability within HTTP itself. The way to do this is defined in an Internet standard for "Alternative Services" [8], which is a means to list alternative ways to access the same resources.

For example. the Google homepage, www.google.com, includes the HTTP header:

alt-svc: quic=":443"; ma=2592000; v="44,43,39"

This indicates that the same material is accessible using QUIC over port 443. The "ma" field is the time to keep this information on the local client, which in this case is 30 days, and the "v" field indicates that the server will negotiate QUIC versions 39, 43 and 44.

QUIC Lessons

QUIC is a rather forceful assertion that the Internet infrastructure is now heavily ossified and more highly constrained than ever. There is no room left for new transport protocols in today's network. If what you want to do can't be achieved within TCP, then all that's left is UDP.

The IP approach to packet size adaptation through fragmentation was a powerful concept once upon a time. A sender did not need to be aware of the constraints that may apply on a path. Any network-level packet fragmentation and reassembly was invisible to the end-to-end packet transfer. This is no longer wise. Senders need to ensure that their packets can reach their intended destinations without any additional requirement for fragmentation handling.

Mutual trust is over in today's Internet. Applications no longer trust other applications. They don't trust the platform that host the application or the shared libraries that implemented essential functions. They are no longer prepared to wait for the platform to support novel features in transport protocols. Applications no longer have any trust in a network to keep their secrets. More and more functions and services are being pulled back into the application and what is passed out from an application is, as much as possible, cloaked in a privacy shroud.

There is a tension between speed, security and paranoia. An ideal outcome is one that is faster, private and secure. Where this is not obvious and the inevitable trade-offs emerge, it seems that we have some minimum security and privacy requirements that simply must be achieved. But once we have achieved this minimum, we are then happy to trade off incremental improvements in privacy and security for better session performance.

The traditional protocol stack model was a convenient abstraction, not a design rule. Applications do not necessarily need to bind to transport-layer sockets provided by the underlying platform. Applications can implement their own end-to-end transport if necessary.

The Internet's infrastructure might be heavily ossified, but the application space is seeing a new set of possibilities open up. Applications need not wait for the platform to include support for a particular transport protocol or await the deployment of a support library to support a particular name resolution function. Applications can solve these issues for themselves directly. The gain in flexibility and agility is considerable.

There is a price to pay for this new-found agility, and that price is broad interoperability. Browsers that support QUIC can open up UDP connections to certain servers and run QUIC, but browsers cannot assume, as they do with TCP, that QUIC is a universal and interoperable lingua franca of the Internet. While QUIC is a fascinating adaptation, with some very novel concepts, it is still an optional adaptation. For those clients and servers who do not support QUIC, or for network paths where UDP port 443 is not supports the common fallback is TCP. The expansion of the Internet is inevitably accompanied by inertial bloat, and as we've seen with the extended saga of IPv6 deployment, it is a formidable expectation to think that the entire Internet will embrace a new technical innovation in a timeframe of months, years or possibly even decades! That does not mean that we can't think new thoughts, and that we can't realize these new ideas into new services on the Internet. We certainly can, and QUIC is an eloquent demonstration of exactly how to craft innovation into a rather stolid and resistant underlying space.

Further Reading

QUIC has excited considerable interest over the past couple of years and there are many posts to be found on the net. Here's a small sample of this online material that you may find to be of interest.

A useful consideration of positive and negative aspects of QUIC are in Robin Marx's post "QUIC and HTTP/3: Too big to fail?" https://calendar.perfplanet.com/2018/quic-and-http-3-too-big-to-fail/

A slightly older (2014) but useful technical overview of QUIC can be found in Shigeki Ohtsu's presentation to the HTTP/2 Conference Japan https://www.slideshare.net/shigeki_ohtsu/quic-overview

A commentary on Cloudflare's investigations with QUIC can be found in a recent blog post: "The Road to QUIC": https://blog.cloudflare.com/the-road-to-quic/

References

[1] Jon Postel, "Transmission Control Protocol," RFC 793, September 1981.

[2] Geoff Huston, "Faster," The ISP Column, June 2005. **https://www.potaroo.net/ispcol/2005-06/faster.html**

[3] Neal Cardwell, Yuchuing Cheng, C. Stephen Gunn, Soheil Hasses Yeganeh and Van Jacobsen, "BBR: congestion-based congestion control," Communications of the ACM, Vol. 60, Issue 2, pp 58-66, February 2017.

[4] Eric Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.34," RFC 8446, August 2018.

[5] IANA Protocol Numbers Registry. **https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml**

[6] Geoff Huston, "Anatomy: A look Inside Network Address Translators," The Internet Protocol Journal, Vol. 7, No. 3, September 2004.

[7] Arnt Gulbrandsen, Paul Vixie and Levon Esibov, "A DNS RR for specifying the location of servicesd (DNS SRV)," RFC 2782, February 2000.

[8] Mark Nottingham, Patrick McManus and Julian Reschke, "HTTP Alternative Services," RFC7838, April 2016.

Disclaimer

Author

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

www.potaroo.net