HTTP/3 Explained

by Daniel Stenberg

Table of Contents

HTTP/3 explained

This book effort was started in March 2018. The plan is to document HTTP/3 and its underlying protocol: QUIC. Why, how they work, protocol details, the implementations and more.

The book is entirely free and is meant to be a collaborative effort involving anyone and everyone who wants to help out.

Prerequisites

A reader of this book is presumed to have a basic understanding of TCP/IP networking, the fundamentals of HTTP and the web. For further insights and specifics about HTTP/2 we recommend first reading up the details in http2 explained.

Author

This book is created and the work is started by Daniel Stenberg. Daniel is the founder and lead developer of curl, the world's most widely used HTTP client software. Daniel has worked with and on HTTP and internet protocols for over two decades and is the author of http2 explained.

Home

The home page for this book is found at daniel.haxx.se/http3-explained.

Help out

If you find mistakes, omissions, errors or blatant lies in this document, please send us a refreshed version of the affected paragraph and we will make amended versions. We will give proper credits to everyone who helps out. I hope to make this document better over time.

Preferably, you submit errors or pull requests on the book's github page.

License

Why QUIC

QUIC is a name, not an acronym. It is pronounced exactly like the English word "quick".

QUIC is in many ways what could be seen as a way of doing a new reliable and secure transport protocol that is suitable for a protocol like HTTP and that can address some of the known shortcomings of doing HTTP/2 over TCP and TLS. The logical next step in the web transport evolution.

QUIC is not limited to just transporting HTTP. The desire to make the web and data in general delivered faster to end users is probably the largest reason and push that initially triggered the creation of this new transport protocol.

So why create a new transport protocol and why do it on top of UDP?

Remember HTTP/2?

The HTTP/2 specification RFC 7540 was published in May 2015 and the protocol has since then been implemented and deployed widely across the Internet and the World Wide Web.

In early 2018, almost 40% of the top-1000 web sites run HTTP/2, around 70% of all HTTPS requests Firefox issues get HTTP/2 responses back and all major browsers, servers and proxies support it.

HTTP/2 addresses a whole slew of shortcomings in HTTP/1 and with the introduction of the second version of HTTP users can stop using a bunch of work-arounds. Some of which are pretty burdensome on web developers.

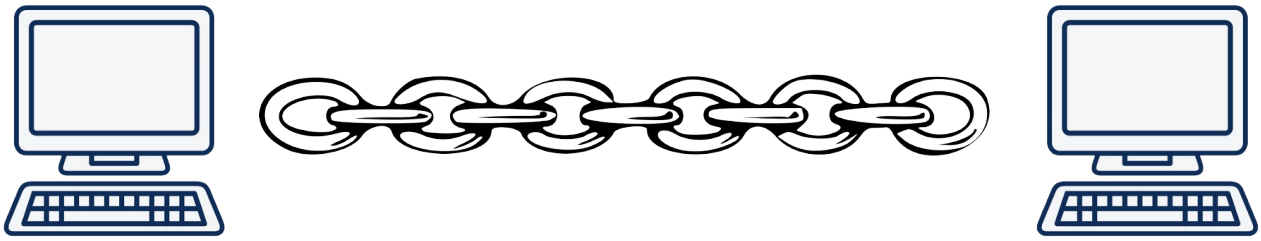
One of the primary features of HTTP/2 is that it makes use of multiplexing, so that many logical streams are sent over the same physical TCP connection. This makes a lot of things better and faster. It makes congestion control work much better, it lets users use TCP much better and thus properly saturate the bandwidth, makes the TCP connections more long-lived - which is good so that they get up to full speed more frequently than before. Header compression makes it use less bandwidth.

With HTTP/2, browsers typically use *one* TCP connection to each host instead of the previous *six*. In fact, connection coalescing and "desharding" techniques used with HTTP/2 may actually even reduce the number of connections much more than so.

HTTP/2 fixed the HTTP head of line blocking problem, where clients had to wait for the first request in line to finish before the next one could go out.

TCP head of line blocking

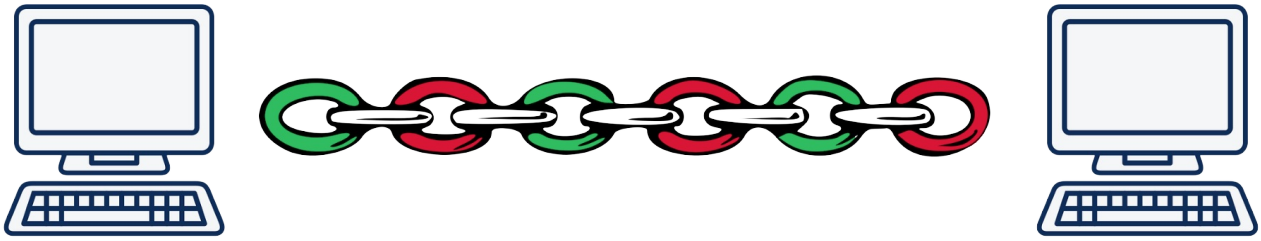
HTTP/2 is done over TCP and with much fewer TCP connections than when using earlier HTTP versions. TCP is a protocol for reliable transfers and you can basically think of it as an imaginary chain between two machines. What is being put out on the network in one end will end up in the other end, in the same order - eventually. (Or the connection breaks.)



With HTTP/2, typical browsers do tens or hundreds of parallel transfers over a single TCP connection.

If a single packet is dropped, or lost in the network somewhere between two endpoints that speak HTTP/2, it means the entire TCP connection is brought to a halt while the lost packet is re-transmitted and finds its way to the destination. Since TCP is this "chain", it means that if one link is suddenly missing, everything that would come after the lost link needs to wait.

An illustration using the chain metaphor when sending two streams over this connection. A red stream and a green stream:



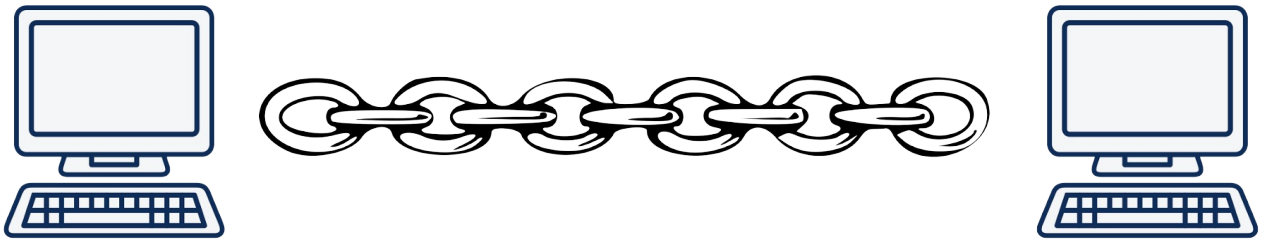
It becomes a TCP-based head of line block!

As the packet loss rate increases, HTTP/2 performs less and less well. At 2% packet loss (which is a terrible network quality, mind you), tests have proven that HTTP/1 users are usually better off - because they typically have up to six TCP connections to distribute lost packets over. This means for every lost packet the other connections can still continue.

Fixing this issue is not easy, if at all possible, with TCP.

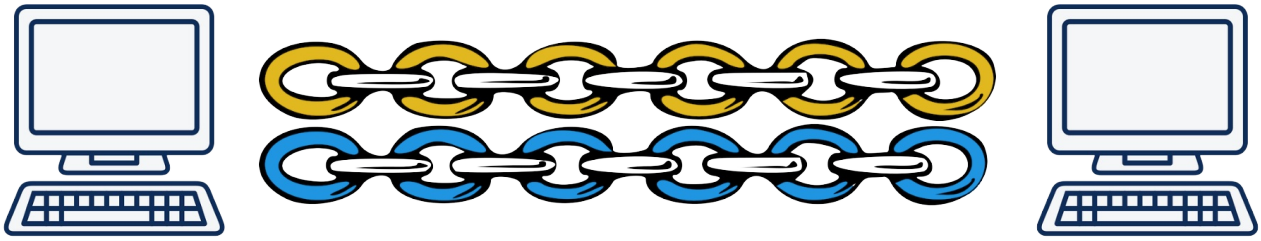
Independent streams avoids the block

With QUIC there is still a connection setup between the two end-points that makes the connection secure and the data delivery reliable.



When setting up two different streams over this connection, they are treated independently so that if any link goes missing for one of the streams, only that stream, that particular chain, has to pause and wait for the missing link to get retransmitted.

Illustrated here with one yellow and one blue stream sent between two end-points.



TCP or UDP

If we can't fix head-of-line blocking within TCP, then in theory we should be able to make a new transport protocol next to UDP and TCP in the network stack. Or perhaps even use SCTP which is a transport protocol standardized by the IETF in RFC 4960 with several of the desired characteristics.

However, in recent years efforts to create new transport protocols have almost entirely been halted because of the difficulties in deploying them on the Internet. Deployment of new protocols is hampered by many firewalls, NATs, routers and other middle-boxes that only allow TCP or UDP are deployed between users and the servers they need to reach. Introducing another transport protocol makes N% of the connections fail because they are being blocked by boxes that see it not being UDP or TCP and thus evil or wrong somehow. The N% failure rate is often deemed too high to be worth the effort.

Additionally, changing things in the transport protocol layer of the network stack typically means protocols implemented by operating system kernels. Updating and deploying new operating system kernels is a slow process that requires significant effort. Many TCP improvements standardized by the IETF are not widely deployed or used because they are not broadly supported.

Why not SCTP-over-UDP

SCTP is a reliable transport protocol with streams, and for WebRTC there are even existing implementations using it over UDP.

This was not deemed good enough as a QUIC alternative due to several reasons, including:

- SCTP does not fix the head-of-line-blocking problem for streams
- SCTP requires the number of streams to be decided at connection setup
- SCTP does not have a solid TLS/security story
- SCTP has a 4-way handshake, QUIC offers 0-RTT
- QUIC is a bytestream like TCP, SCTP is message-based
- QUIC connections can migrate between IP addresses but SCTP cannot

For more details on the differences, see A Comparison between SCTP and QUIC.

Ossification

The internet is a network of networks. There is equipment set up on the Internet in many different places along the way to make sure this network of networks works as it is supposed to. These devices, the boxes that are distributed out in the network, are what we sometimes refer to as middle-boxes. Boxes that sit somewhere between the two end-points that are the primary parties involved in a traditional network data transfer.

These boxes serve many different specific purposes but I think we can say that universally they are put there because someone thinks they must be there to make things work.

Middle-boxes route IP packets between networks, they block malicious traffic, they do NAT (Network Address Translation), they improve performance, some try to spy on the passing traffic and more.

In order to perform their duties these boxes must know about networking and the protocols that are monitored or modified by them. They run software for this purpose. Software that is not always upgraded frequently.

While they are glue components that keep the Internet together they are also often not keeping up with the latest technology. The middle of the network typically does not move as fast as the edges, as the clients and the servers of the world.

The network protocols that these boxes might want to inspect, and have ideas about what is okay and what is not then have this problem: these boxes were deployed some time ago when the protocols had a feature set of that time. Introducing new features or changes in behavior that were not known before risks ending up considered bad or illegal by such boxes. Such traffic may well just be dropped or delayed to the degree that users really do not want to use those features.

This is called "protocol ossification".

Changes to TCP also suffer from ossification: some boxes between a client and the remote server will spot unknown new TCP options and block such connections since they do not know what the options are. If allowed to detect protocol details, systems learn how protocols typically behave and over time it becomes impossible to change them.

The only truly effective way to "combat" ossification is to encrypt as much of the communication as possible in order to prevent middle-boxes from seeing much of the protocol passing through.

Secure

QUIC is always secure. There is no clear-text version of the protocol so to negotiate a QUIC connection means doing cryptography and security with TLS 1.3. As mentioned above, this prevents ossification as well as other sorts of blocks and special treatments, as well as making sure QUIC has all the secure properties of HTTPS that web users have come to expect and want.

There are only a few initial handshake packets that are sent in the clear before the encryption protocols have been negotiated.

Earlier data

QUIC offers both 0-RTT and 1-RTT handshakes that reduce the time it takes to negotiate and setup a new connection. Compare with the 3-way handshake of TCP.

In addition to that, QUIC offers "early data" support from the get go which is done to allow more data and it is used more easily than TCP Fast Open.

With the stream concept, another logical connection to the same host can be done at once without having to wait for the existing one to end first.

TCP Fast Open is problematic

TCP Fast Open was published as RFC 7413 in December 2014 and that specification describes how applications can pass data to the server to be delivered already in the first TCP SYN packet.

Actual support for this feature in the wild has taken time and is riddled with problems even today in 2018. The TCP stack implementors have had issues and so have applications trying to take advantage of this feature - both in knowing in which OS version to try to activate it but also in figuring out how to gracefully back down and deal when problems arise. Several networks have been identified to interfere with TFO traffic and they have thus actively ruined such TCP handshakes.

Process

The initial QUIC protocol was designed by Jim Roskind at Google and was initially implemented in 2012, announced publicly to the world in 2013 when Google's experimentation broadened.

Back then, QUIC was still claimed to be an acronym for "Quick UDP Internet Connections", but that has been dropped since then.

Google implemented the protocol and subsequently deployed it both in their widely used browser (Chrome) and in their widely used server-side services (Google search, gmail, youtube and more). They iterated protocol versions fairly quickly and over time they proved the concept to work reliably for a vast portion of users.

In June 2015, the first internet draft for QUIC was sent to the IETF for standardization, but it took until late 2016 for a QUIC working group to get approved and started. But then it took off immediately with a high degree of interest from many parties.

In 2017, numbers quoted by QUIC engineers at Google mentioned that around 7% of *all* Internet traffic were already using this protocol. The Google version of the protocol.

IETF

The QUIC working group that was established to standardize the protocol within the IETF quickly decided that the QUIC protocol should be able to transfer other protocols than "just" HTTP. Google-QUIC only ever transported HTTP - in practice it transported what was effectively HTTP/2 frames, using the HTTP/2 frame syntax.

It was also stated that IETF-QUIC should base its encryption and security on TLS 1.3 instead of the "custom" approach used by Google-QUIC.

In order to satisfy the send-more-than-HTTP demand, the IETF QUIC protocol architecture was split in two separate layers: the transport QUIC and the "HTTP over QUIC" layer (the latter sometimes referred to as "hq").

This layer split, while it may sound innocuous, has caused the IETF-QUIC to differ quite a lot from the original Google-QUIC.

The working group did however soon decide that in order to get the proper focus and ability to deliver QUIC version 1 on time, it would focus on delivering HTTP, leaving non-HTTP transports to later work.

In March 2018 when we started working on this book, the plan was to ship the final specification for QUIC version 1 in November 2018; this was later postponed to July 2019.

While the work on IETF-QUIC has progressed, the Google team has incorporated details from the IETF version and has started to slowly progress their version of the protocol towards what the IETF version might become. Google has continued using their version of QUIC in their browser and services.

Most new implementations under development have decided to focus on the IETF version and are not compatible with the Google version.

Experience from HTTP/2

The HTTP/2 specification RFC 7540 was published in May 2015, just a month before QUIC was brought to IETF for the first time.

With HTTP/2, the foundation for changing HTTP over the wire was laid out and the working group that created HTTP/2 was already of the mindset that this would help iterating to new HTTP versions much faster than it had taken to go to version 2 from version 1 (about 16 years).

With HTTP/2, users and software stacks got used to the idea that HTTP can no longer be assumed to be done with a text-based protocol in a serial manner.

HTTP-over-QUIC was renamed to HTTP/3 in November 2018.

Status

The QUIC working group has worked fiercely since late 2016 on specifying the protocols and the plan is now to have it done by July 2019.

As of November 2018, there still has not been any larger interoperability tests with HTTP/3 - only with the existing two implementations and none of them are done by a browser or a popular open server software.

There are fifteen or so different QUIC implementations listed in the QUIC working groups' wiki pages, but far from all of them can interoperate on the latest spec draft revisions.

Implementing QUIC is not easy and the protocol has kept moving and changing even up to this date.

Servers

NGINX support for QUIC and HTTP/3 is under development. It is planned to be released during NGINX 1.17 development cycle.

There have been no public statement in terms of support for QUIC from Apache.

Clients

None of the larger browser vendors have yet shipped any version, at any state, that can run the IETF version of QUIC or HTTP/3.

Google Chrome has shipped with a working implementation of Google's own QUIC version since many years, but that does not interoperate with the IETF QUIC protocol and its HTTP implementation is different than HTTP/3.

Mozilla is developing Neqo - a QUIC and HTTP/3 implementation written in Rust. Neqo is planned to be integrated in Necko (which is a network library used in many Mozilla-based client applications - including Firefox).

curl shipped the first experimental HTTP/3 support (draft-22) in the 7.66.0 release on September 11, 2019. curl uses either the Quiche library from Cloudflare or the ngtcp2 family of libraries to get the work done.

Implementation Obstacles

QUIC decided to use TLS 1.3 as the foundation for the crypto and security layer to avoid inventing something new and instead lean on a trustworthy and existing protocol. However, while doing this, the working group also decided that to really streamline the use of TLS in QUIC, it should only use "TLS messages" and not "TLS records" for the protocol.

This might sound like an innocuous change, but this has actually caused a significant hurdle for many QUIC stack implementors. Existing TLS libraries that support TLS 1.3 simply do not have APIs enough to expose this functionality and allow QUIC to access it. While several QUIC implementors come from larger organizations who work on their own TLS stack in parallel, this is not true for everyone.

The dominant open source heavyweight OpenSSL for example, does not have any API for this. The plan to address this seems to happn in their PR 8797 that aims to introduce an API that is very similar to the one of BoringSSL.

This will eventually also lead to deployment obstacles since QUIC stacks will need to either base themselves on other TLS libraries, use a separate patched OpenSSL build or require an update to a future OpenSSL version.

Kernels and CPU load

Both Google and Facebook have mentioned that their wide scale deployments of QUIC require roughly twice the amount of CPU than the same traffic load does when serving HTTP/2 over TLS.

Some explanations for this include

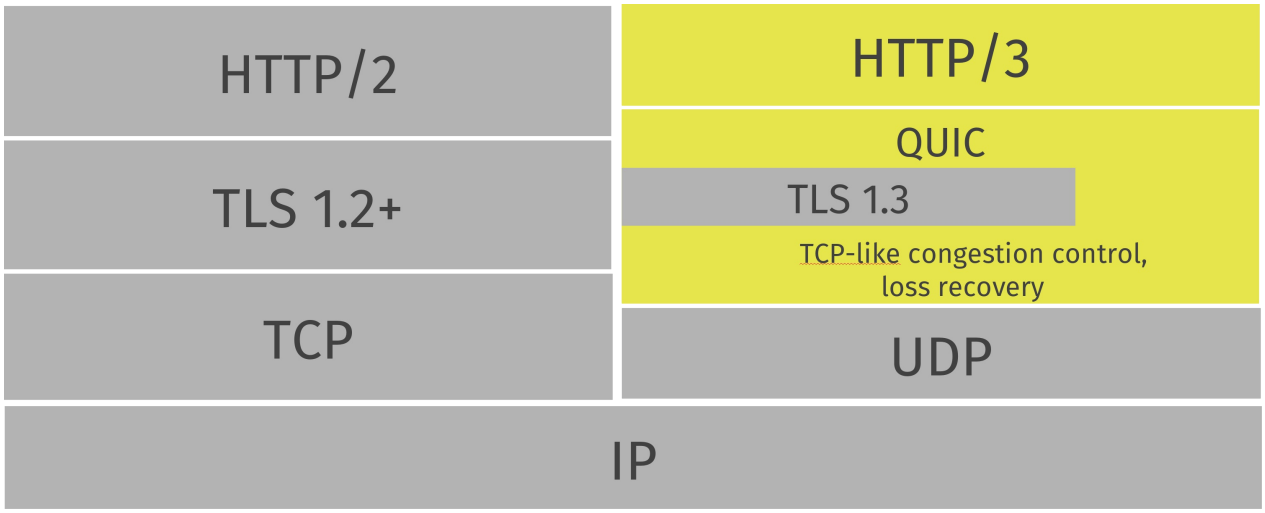
- the UDP parts in primarily Linux is not at all as optimized as the TCP stack is, since it has not traditionally been used for high speed transfers like this.
- TCP and TLS offloading to hardware exist, but that is much rarer for UDP and basically non-existing for QUIC.

There are reasons to believe that performance and CPU requirements will improve over time.

Protocol features

The QUIC protocol from a high level.

Illustrated below is the HTTP/2 network stack on the left and the QUIC network stack on the right, when used as HTTP transport.



Transfer protocol over UDP

QUIC is a transfer protocol implemented on top of UDP. If you watch your network traffic casually, you will see QUIC appear as UDP packets.

Based on UDP it also then uses UDP port numbers to identify specific network services on a given IP address.

All known QUIC implementations are currently in user-space, which allows for more rapid evolution than kernel-space implementations typically allow.

Will it work?

There are enterprises and other network setups that block UDP traffic on other ports than 53 (used for DNS). Others throttle such data in ways that makes QUIC perform worse than TCP based protocols. There is no end to what some operators may do.

For the foreseeable future, all use of QUIC-based transports will probably have to be able to gracefully fall-back to another (TCP-based) alternative. Google engineers have previously mentioned measured failure rates in the low single-digit percentages.

Will it improve?

Chances are that if QUIC proves to be a valuable addition to the Internet world, people will want to use it and they will want it to function in their networks and then companies may start to reconsider their obstacles. During the years the development of QUIC has progressed, the success rate for establishing and using QUIC connections across the Internet has increased.

Reliable data transfers

While UDP is not a reliable transport, QUIC adds a layer on top of UDP that introduces reliability. It offers re-transmissions of packets, congestion control, pacing and the other features otherwise present in TCP.

Data sent over QUIC from one end-point will appear in the other end sooner or later, as long as the connection is maintained.

Multiple streams within connections

Similar to SCTP, SSH and HTTP/2, QUIC features separate logical streams within the physical connections. A number of parallel streams that can transfer data simultaneously over a single connection without affecting the other streams.

A connection is a negotiated setup between two end-points similar to how a TCP connection works. A QUIC connection is made to a UDP port and IP address, but once established the connection is associated by its "connection ID".

Over an established connection, either side can create streams and send data to the other end. Streams are delivered in-order and they are reliable, but different streams may be delivered out-of-order.

QUIC offers flow control on both connection and streams.

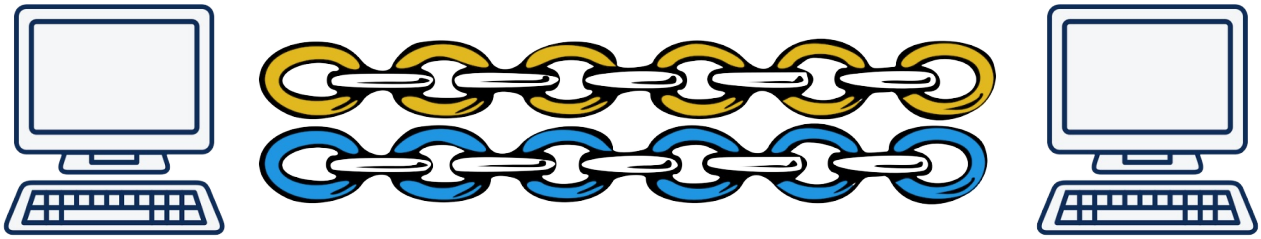
See further details in connections and streams sections

In order delivery

QUIC guarantees in-order delivery of streams, but not between streams. This means that each stream will send data and maintain data order, but each stream may reach the destination in a different order than the application sent it!

For example: stream A and B are transferred from a server to a client. Stream A is started first and then stream B. In QUIC, a lost packet only affects the stream to which the lost packet belongs. If stream A loses a packet but stream B does not, stream B may continue its transfers and complete while stream A's lost packet is re-transmitted. This was not possible with HTTP/2.

Illustrated here with one yellow and one blue stream sent between two QUIC end-points over a single connection. They are independent and may arrive in a different order, but each stream is reliably delivered to the application in order.



Fast handshakes

QUIC offers both 0-RTT and 1-RTT connection setups, meaning that at best QUIC needs no extra round-trips at all when setting up a new connection. The faster of those two, the 0-RTT handshake, only works if there has been a previous connection established to a host and a secret from that connection has been cached.

Early data

QUIC allows a client to include data already in the 0-RTT handshake. This feature allows a client to deliver data to the peer as fast as it possibly can, and that then of course allows the server to respond and send data back even sooner.

TLS 1.3

The transport security used in QUIC is using TLS 1.3 (RFC 8446) and there are never any unencrypted QUIC connections.

TLS 1.3 has several advantages compared to older TLS versions but a primary reason for using it in QUIC is that 1.3 changed the handshake to require fewer roundtrips. It reduces protocol latency.

The Google legacy version of QUIC used a custom crypto.

Transport and application level

The IETF QUIC protocol is a transport protocol, on top of which other application protocols can be used. The initial application layer protocol is HTTP/3 (h3).

The transport layer supports connections and streams.

The legacy Google version of QUIC had transport and HTTP glued together into one single do-it-all and was a more special-purpose send-http/2-frames-over-udp protocol.

HTTP/3 over QUIC

The HTTP layer, called HTTP/3, does HTTP-style transports, including HTTP header compression using QPACK - which is similar to the HTTP/2 compression named HPACK.

The HPACK algorithm depends on an *ordered* delivery of streams so it was not possible to reuse it for HTTP/3 without modifications since QUIC offers streams that can be delivered out of order. QPACK can be seen as the QUIC-adapted version of HPACK.

Non-HTTP over QUIC

The work on sending protocols other than HTTP over QUIC has been postponed until after QUIC version 1 has shipped.

How QUIC works

Without explaining the exact bits and bytes on the wire, this section describes how the fundamental building blocks of the QUIC transport protocol work. If you want to implement your own QUIC stack, this description should give you a general understanding, but for all the details, refer to the actual IETF Internet Drafts and RFCs.

1. Set up a connection
2. ... that includes TLS security
3. Then use streams

Connections

A QUIC connection is a single conversation between two QUIC endpoints. QUIC's connection establishment combines version negotiation with the cryptographic and transport handshakes to reduce connection establishment latency.

To actually send data over such a connection, one or more streams have to be created and used.

Connection ID

Each connection possesses a set of connection identifiers, or connection IDs, each of which can be used to identify the connection. Connection IDs are independently selected by endpoints; each endpoint selects the connection IDs that its peer uses.

The primary function of these connection IDs is to ensure that changes in addressing at lower protocol layers (UDP, IP, and below) do not cause packets for a QUIC connection to be delivered to the wrong endpoint.

By taking advantage of the connection ID, connections can thus migrate between IP addresses and network interfaces in ways TCP never could. For instance, migration allows an in-progress download to move from a cellular network connection to a faster wifi connection when the user moves their device into a location offering wifi. Similarly, the download can proceed over the cellular connection if wifi becomes unavailable.

Port numbers

QUIC is built atop UDP, so a 16 bit port number field is used to differentiate incoming connections.

Version negotiation

An QUIC connection request originating from a client will tell the server which QUIC protocol version it wants to speak, and the server will respond with a list of supported versions for the client to select from.

Connections use TLS

Immediately after the initial packet setting up a connection, the initiator sends a crypto frame that starts setting up the secure layer handshake. The security layer uses TLS 1.3 security.

There is no way to opt-out or avoid using TLS for a QUIC connection. The protocol is designed to be hard for middle-boxes to tamper with, in order to help prevent ossification of the protocol.

Streams

Streams in QUIC provide a lightweight, ordered byte-stream abstraction.

There are two basic types of stream in QUIC:

- Unidirectional streams carry data in one direction: from the initiator of the stream to its peer.
- Bidirectional streams allow for data to be sent in both directions.

Either type of stream can be created by either endpoint, can concurrently send data interleaved with other streams, and can be canceled.

To send data over a QUIC connection, one or more streams are used.

Flow control

Streams are individually flow controlled, allowing an endpoint to limit memory commitment and to apply back pressure. The creation of streams is also flow controlled, with each peer declaring the maximum stream ID it is willing to accept at a given time.

Stream Identifiers

Streams are identified by an unsigned 62-bit integer, referred to as the Stream ID. The least significant two bits of the Stream ID are used to identify the type of stream (unidirectional or bidirectional) and the initiator of the stream.

The least significant bit (0x1) of the Stream ID identifies the initiator of the stream. Clients initiate even-numbered streams (those with the least significant bit set to 0); servers initiate odd-numbered streams (with the bit set to 1).

The second least significant bit (0x2) of the Stream ID differentiates between unidirectional streams and bidirectional streams. Unidirectional streams always have this bit set to 1 and bidirectional streams have this bit set to 0.

Stream concurrency

QUIC allows for an arbitrary number of streams to operate concurrently. An endpoint limits the number of concurrently active incoming streams by limiting the maximum stream ID.

The maximum stream ID is specific to each endpoint and applies only to the peer that receives the setting.

Sending and Receiving Data

Endpoints use streams to send and receive data. That is after all their ultimate purpose. Streams are an ordered byte-stream abstraction. Separate streams are however not necessarily delivered in the original order.

Stream Prioritization

Stream multiplexing has a significant effect on application performance if resources allocated to streams are correctly prioritized. Experience with other multiplexed protocols, such as HTTP/2, shows that effective prioritization strategies have a significant positive impact on performance.

QUIC itself does not provide frames for exchanging prioritization information. Instead it relies on receiving priority information from the application that uses QUIC. Protocols that use QUIC are able to define any prioritization scheme that suits their application semantics.

When HTTP/3 is used over QUIC, the prioritization is done in the HTTP layer.

0-RTT

To reduce the time required to establish a new connection, a client that has previously connected to a server may cache certain parameters from that connection and subsequently set up a **0-RTT** connection with the server. This allows the client to send data immediately, without waiting for a handshake to complete.

Spin Bit

One of the perhaps longest design discussions within the QUIC working group that has been the subject of several hundred mails and hours of discussions concerns a single bit: the Spin Bit.

The proponents of this bit insist that there is a need for operators and people on the path between two QUIC endpoints to be able to measure latency.

The opponents to this feature do not like the potential information leak.

Spinning a bit

Both endpoints, the client and the server, maintain a spin value, 0 or 1, for each QUIC connection, and they set the spin bit on packets it sends for that connection to the appropriate value.

Both sides then send out packets with that spin bit set to the same value for as long as one round trip lasts and then it toggles the value. The effect is then a pulse of ones and zeroes in that bitfield that observers can monitor.

This measuring only works when the sender is neither application nor flow control limited and packet reordering over the network can also make the data noisy.

User-space

Implementing a transport protocol in user-space helps enable quick iteration of the protocol, as it is comparatively easy to evolve the protocol without necessitating that clients and servers update their operating system kernel to deploy new versions.

Nothing inherent in QUIC prevents it from being implemented and offered by operating system kernels in the future, should someone find that a good idea.

Many implementations

One obvious effect of implementing a new transport protocol in user-space is that we can expect to see many independent implementations.

Different applications are likely to include (or layer atop) different HTTP/3 and QUIC implementations for the foreseeable future.

API

One of the success factors for regular TCP and programs using that, is the standardized socket API. It has well defined functionality and using this API you can move programs between many different operating systems as TCP works the same.

QUIC is not there. There is no standard API for QUIC.

With QUIC, you need to pick one of the existing library implementations and stick with its API. It makes applications "locked in" to a single library to some extent. Changing to another library means another API and that might involve a lot of work.

Also, since QUIC is typically implemented in user-space, it can't easily just extend the socket API or appear similar to how existing TCP and UDP functionality do. Using QUIC will mean using another API than the socket API.

HTTP/3

As mentioned previously, the first and primary protocol to transport over QUIC is HTTP.

Much like HTTP/2 was once introduced to transport HTTP over the wire in a completely new way, HTTP/3 is yet again introducing a new way to send HTTP over the network.

HTTP still maintains the same paradigms and concepts like before. There are headers and a body, there is a request and a response. There are verbs, cookies and caching. What primarily changes with HTTP/3 is how the bits gets sent over to the other side of the communication.

In order to do HTTP over QUIC, changes were required and the results of this is what we now call HTTP/3. These changes were required because of the different nature that QUIC provides as opposed to TCP. These changes include:

- In QUIC the streams are provided by the transport itself, while in HTTP/2 the streams were done within the HTTP layer.
- Due to the streams being independent of each other, the header compression protocol used for HTTP/2 could not be used without it causing a head of block situation.
- QUIC streams are slightly different than HTTP/2 streams. The HTTP/3 section will detail this somewhat.

HTTPS:// URLs

HTTP/3 will be performed using `HTTPS://` URLs. The world is full of these URLs and it has been deemed impractical and downright unreasonable to introduce another URL scheme for the new protocol. Much like HTTP/2 did not need a new scheme, neither will HTTP/3.

The added complexity in the HTTP/3 situation is however that where HTTP/2 was a completely new way of transporting HTTP over the wire, it was still based on TLS and TCP like HTTP/1 was. The fact that HTTP/3 is done over QUIC changes things in a few important aspects.

Legacy, clear-text, `HTTP://` URLs will be left as-is and as we proceed further into a future with more secure transfers they will probably become less and less frequently used. Requests to such URLs will simply not be upgraded to use HTTP/3. In reality they rarely upgrade to HTTP/2 either, but for other reasons.

Initial connection

The first connection to a fresh, not previously visited host for a HTTPS:// URL probably has to be done over TCP (possibly in addition to a parallel attempt to connect via QUIC). The host might be a legacy server without QUIC support or there might be a middle box in between setting up obstacles preventing a QUIC connection from succeeding.

A modern client and server would presumably negotiate HTTP/2 in the first handshake. When the connection has been setup and the server responds to a client HTTP request, the server can tell the client about its support of and preference for HTTP/3.

Alt-svc

The alternate service (Alt-svc:) header and its corresponding `ALT-SVC` HTTP/2 frame are not specifically created for QUIC or HTTP/3. They are part of an already designed and created mechanism for a server to tell a client: *"look, I run the same service on THIS HOST using THIS PROTOCOL on THIS PORT"*. See details in RFC 7838.

A client that receives such an Alt-svc response is then advised to, if it supports and wants to, connect to that given other host in parallel in the background - using the specified protocol - and if it is successful switch its operations over to that instead of the initial connection.

If the initial connection uses HTTP/2 or even HTTP/1, the server can respond and tell the client that it can connect back and try HTTP/3. It could be to the same host or to another one that knows how to serve that origin. The information given in such an Alt-svc response has an expiry timer, making clients able to direct subsequent connections and requests directly to the alternative host using the suggested alternative protocol, for a certain period of time.

Example

An HTTP server includes an `Alt-Svc:` header in its response:

```
Alt-Svc: h3=":50781"
```

This indicates that HTTP/3 is available on UDP port 50781 at the same host name that was used to get this response.

A client can then attempt to setup a QUIC connection to that destination and if successful, continue communicating with the origin like that instead of the initial HTTP version.

QUIC streams and HTTP/3

HTTP/3 is made for QUIC so it takes full advantage of QUIC's streams, where HTTP/2 had to design its entire stream and multiplexing concept of its own on top of TCP.

HTTP requests done over HTTP/3 use a specific set of streams.

HTTP/3 frames

HTTP/3 means setting up QUIC streams and sending over a set of frames to the other end. There's but a small fixed number (actually nine on December 18th, 2018!) of known frames in HTTP/3. The most important ones are probably:

- HEADERS, that sends compressed HTTP headers
- DATA, sends binary data contents
- GOAWAY, please shutdown this connection

HTTP Request

The client sends its HTTP request on a client-initiated *bidirectional* QUIC stream.

A request consists of a single HEADERS frame and might optionally be followed by one or two other frames: a series of DATA frames and possibly a final HEADERS frame for trailers.

After sending a request, a client closes the stream for sending.

HTTP Response

The server sends back its HTTP response on the bidirectional stream. A HEADERS frame, a series of DATA frames and possibly a trailing HEADERS frame.

QPACK headers

The HEADERS frames contain HTTP headers compressed using the QPACK algorithm. QPACK is similar in style to the HTTP/2 compression called HPACK (RFC 7541), but modified to work with streams delivered out of order.

QPACK itself uses two additional unidirectional QUIC streams between the two end-points. They are used to carry dynamic table information in either direction.

HTTP/3 Prioritization

One of the HTTP/3 stream frames is called `PRIORITY`. It is used to set priority and dependency on a stream in a way similar to how it works in HTTP/2.

The frame can set a specific stream to depend on another specific stream and it can set a "weight" on a given stream.

A dependent stream should only be allocated resources if either all of the streams that it depends on are closed or it is not possible to make progress on them.

A stream weight is value between 1 and 256 and it is specified that streams with the same parent **should** be allocated resources proportionally based on their weight.

HTTP/3 Server push

HTTP/3 server push is similar to what is described in HTTP/2 (RFC 7540), but uses different mechanisms.

A server push is effectively the response to a request that the client never sent!

Server pushes are only allowed to happen if the client side has agreed to them. In HTTP/3 the client even sets a limit for how many pushes it accepts by informing the server what the max push stream ID is. Going over that limit will cause a connection error.

If the server deems it likely that the client wants an extra resource that it hasn't asked for but ought to have anyway, it can send a `PUSH_PROMISE` frame (over the request stream) showing what the request looks like that the push is a response to, and then send that actual response over a new stream.

Even when pushes have been said to be acceptable by the client before-hand, each individual pushed stream can still be canceled at any time if the client deems that suitable. It then sends a `CANCEL_PUSH` frame to the server.

Problematic

Ever since this feature was first discussed in the HTTP/2 development and later after the protocol shipped and has been deployed over the Internet, this feature has been discussed, disliked and pounded up in countless different ways in order to get it to become useful.

Pushing is never "free", since while it saves a half round-trip it still uses bandwidth. It is often hard or impossible for the server-side to actually know with a high level of certainty if a resource should be pushed or not.

HTTP/3 compared to HTTP/2

HTTP/3 is designed for QUIC, which is a transport protocol that handles streams by itself.

HTTP/2 is designed for TCP, and therefore handles streams in the HTTP layer.

Similarities

The two protocols offer clients virtually identical feature sets.

- Both protocols offer streams
- Both protocols offer server push support
- Both protocols have header compression, and QPACK and HPACK are similar in design.
- Both protocols offer multiplexing over a single connection using streams
- Both protocols do prioritization on streams

Differences

The differences are in the details and primarily there thanks to HTTP/3's use of QUIC:

- HTTP/3 has better and more likely to work early data support thanks to QUIC's 0-RTT handshakes, while TCP Fast Open and TLS usually sends less data and often faces problems.
- HTTP/3 has much faster handshakes thanks to QUIC vs TCP + TLS.
- HTTP/3 does not exist in an insecure or unencrypted version. HTTP/2 can be implemented and used without HTTPS - even if this is rare on the Internet.
- HTTP/2 can be negotiated directly in a TLS handshake with the ALPN extension, while HTTP/3 is over QUIC so it needs an `Alt-Svc:` header response first to inform the client about this fact.

Criticism

UDP will never work

A lot of enterprises, operators and organizations block or rate-limit UDP traffic outside of port 53 (used for DNS) since it has in recent days mostly been abused for attacks. In particular, some of the existing UDP protocols and popular server implementations for them have been vulnerable for amplification attacks where one attacker can make a huge amount of outgoing traffic to target innocent victims.

QUIC has built-in mitigation against amplification attacks by requiring that the initial packet must be at least 1200 bytes and by restriction in the protocol that says that a server must not send more than three times the size of the request in response without receiving a packet from the client in response.

UDP is slow in kernels

This seems to be the truth, at least today in 2018. We can of course not tell how this will develop and how much of this is simply the result of UDP transfer performance not having been in developers' focus for many years.

For most clients, this "slowness" is probably never even noticeable.

QUIC takes too much CPU

Similar to the "UDP is slow" remark above, this is partly because the TCP and TLS usage of the world has had a longer time to mature, improve and get hardware assistance.

There are reasons to expect this to improve over time. The question is how much this extra CPU usage will hurt deployers.

This is just Google

No it is not. Google brought the initial spec to the IETF after having proved, on a large Internet-wide scale, that deploying this style of protocol over UDP actually works and performs well.

Since then, individuals from a large number of companies and organizations have worked in the vendor-neutral organization IETF to put together a standard transport protocol out of it. In that work, Google employees have of course been participating, but so have employees from a large number of other companies that are interested in furthering the state of transport protocols on the Internet, including Mozilla, Fastly, Cloudflare, Akamai, Microsoft, Facebook and Apple.

This is too small of an improvement

That is not really a critique but an opinion. Maybe it is, and maybe it is too little of an improvement so close in time since HTTP/2 was shipped.

HTTP/3 is likely to perform much better in packet loss-ridden networks, it offers faster handshakes so it will improve latency both as perceived and actual. But is that enough of benefits to motivate people to deploy HTTP/3 support on their servers and for their services? Time and future performance measurements will surely tell!

The specifications

Here is a collection of the latest official drafts for the various parts and components of QUIC and HTTP/3.

Invariants

Version-Independent Properties of QUIC

Transport

QUIC: A UDP-Based Multiplexed and Secure Transport

Recovery

QUIC Loss Detection and Congestion Control

TLS

Using TLS to Secure QUIC

HTTP

Hypertext Transfer Protocol Version 3 (HTTP/3)

QPACK

QPACK: Header Compression for HTTP/3

QUIC v2

In order to get the most possibly focus on the core QUIC protocol and be able to ship it on time, several features that were originally planned to be part of the core protocol have been postponed and are now planned to instead get done in a subsequent QUIC version. QUIC version 2 or beyond.

The author of this document does however have a rather faulty crystal ball so we can not tell for sure exactly what features will or will not appear in version 2. We can however mention some of the features and things that explicitly have been removed from the v1 work to be "worked on later" and that then possibly might appear in a version 2.

Forward Error Correction

Forward error correction (FEC) is a method of obtaining error control in data transmission in which the transmitter sends redundant data and the receiver recognizes only the portion of the data that contains no apparent errors.

Google experimented with this in their original QUIC work but it was subsequently removed again since the experiments did not turn out well. This feature is subject for discussion for QUIC v2 but probably takes for someone to prove that it actually can be a useful addition without too much penalty.

Multipath

Multipath means that the transport can by itself use multiple network paths to maximize resource usage and increase redundancy.

The SCTP proponents of the world like to mention that SCTP already features multipath and so does modern TCP.

Unreliable data

It has been discussed to offer "unreliable" streams as an option, that would then allow QUIC to also replace UDP-style applications.

Non-HTTP adaptions

DNS over QUIC was one of the early mentioned non-HTTP protocols that just might get some attention once QUIC v1 and HTTP/3 ship. But with a new transport like this having been brought on to the world I cannot imagine that it will stop there.