# QUIC

# Making Web Browsing Even Faster and Improving over SPDY

Version 1.0

**Guenter I. Klas**
**Aug 02, 2014**

## The 1 minute takeaway

Google has been developing QUIC as an experimental protocol to solve a number of challenges and achieve certain goals. Amongst them are: eliminating some of the performance issues SPDY has when using it over mobile radio access networks (related to use of the TCP protocol) and improving the responsiveness for web browsing through minimising latency caused by the protocols TLS/SSL and TCP. SPDY is run over both TLS/SSL and TCP. HTTP/2.0 is run over TCP and one expects it to make use of TLS in many cases as well. QUIC essentially replaces mechanisms built into TCP and TLS and aims at improving over those. For practical reasons, it needs to be transported using UDP, thus its name Quick UDP Internet Connections. The paper provides an introduction and overview of QUIC.

**Tags:** QUIC, Quick UDP Internet Connections, Streams, Frames, Chromium, Chrome, SPDY, HTTP/2.0, TLS, SSL, TCP, UDP, multiplexing, congestion avoidance, flow control, forward error correction, packet pacing, bandwidth estimation, white paper

## What is QUIC

QUIC stands for Quick UDP Internet Connections. It is an *experimental* communications protocol from Google and has been created by Google engineers using a similar approach as with SPDY. Its goal is to “speed up the Web” even more than SPDY or HTTP/2.0 is capable of doing. Technically, QUIC enables multiplexed, encrypted, connection-oriented, reliable transport over the User Datagram Protocol (UDP).

## What problem shall QUIC solve

There are various reasons why Google are developing this new protocol.

**First**, Google and others like AT&T [13] found that the SPDY protocol which multiplexes multiple HTTP sessions over a single Transmission Control Protocol (TCP) connection sometimes suffers from performance issues (most likely HTTP/2.0 will do so too). Those issues are e.g. related to how the TCP protocol behaves under packet loss (as it occurs in cellular environments).  That’s a prime reason why the new QUIC protocol runs over the connectionless UDP protocol. This is explained in the next section. The issue is summarised as SPDY Head of Line Blocking for streams [3].

**Second**, the designers aimed at reducing latency in connection set-up and transport. Their argument is correct, that even if unlimited bandwidth were available in the future, latency is impacted by the number of round-trip-times (from browser client to server and back) which is associated with certain Internet protocols per design [4]. In particular during connection set-up, the combination of TLS over TCP requires multiple round-trip times before actual user application data can be shipped. QUIC aims at increasing the probability of being able to send user application data from browser to web server completely avoiding any special “signalling delay” due to connection and security management say in steady state of a browsing session. The buzzword for this is zero-round-trip time or in short 0-RTT. Latency is further reduced by reducing the number of packet re-transmissions in case of packet loss [4].

**Third**, the designers want to provide security and privacy comparable to TLS, which is used together with SPDY and will be more frequently used by browsers together with HTTP/2.0 [3].

**Fourth**, the Google engineers search for better congestion avoidance schemes than available in today's TCP [3].

**Fifth**, they aim at persistence/survival of Internet connections in situations when devices change access networks [3]. When a mobile client changes network and gets a new IP address assigned, the TCP connection breaks as the source (=client) IP address changes. The QUIC connection instead can persist as it has its own connection identifier [3].

**Sixth**, a stated motivation is to "further coalesce" traffic [4]. This might mean collapsing different protocols on different layers into a single one. This is driven by the recognition that having capabilities distributed over two different protocols, "one atop the other, already instigates an overall connection latency that is the sum of the two connection latencies". This is discussed in detail in [4] using the example of SCTP over DTLS. QUIC *combines* features of TLS and TCP. Google call it an "integrated protocol" [4] as it integrates two layers.

**Seventh**, the designers aim at deploying speed improvements "today" rather than "tomorrow". They realise TLS and TCP are slow to evolve, thus a separate route to market (via QUIC) might get traction faster [3].

A number of additional design goals are listed in [4], including e.g. reduced bandwidth consumption.

## Why do SPDY or HTTP/2.0 over TPC have performance issues

Recall, that on a high level, SPDY and HTTP/2.0 are celebrated for speeding up Internet browsing. How is that achieved? One reason is the multiplexing feature. Instead of setting up a sequence of (often up to 6 parallel) TCP connections between browser and web server, SPDY and HTTP/2.0 multiplex many HTTP transactions into a single TCP connection. For a complex website and lengthy browsing session, this eliminates a large amount of round-trip-times for repeatedly setting up TCP connections.

However, one may argue, unknowingly at the time when the engineers decided to multiplex several data streams over a single TCP connection, they literally put all eggs into one single basket (namely a single TCP connection). If one TCP packet gets lost, TCP retransmits the packet and essentially slows down. This unfortunately stalls ***all*** multiplexed (HTTP) data streams with their TCP packets queuing up behind the packet to be retransmitted. The same happens if a single TCP packet gets delayed. This of course is not the case with 'old-fashioned' Internet browsing where multiple overlapping HTTP transactions each use their own dedicated TCP connection in parallel to the same web server. If packet n of TCP connection X gets lost, only packets n+1, n+2, … of the same TCP connection X supporting HTTP transactions get stalled, with no impact on the other, simultaneous HTTP over TCP connections.

Thus, one may argue, by introducing multiplexing of data streams into SPDY (and also HTTP/2.0), one gains speed through avoiding TLS/TCP-level round-trip times, but loses speed in case of TCP packet loss. This is also addressed in a paper from AT&T [13].

In QUIC, a lost UDP packet is impacting only a single stream, and ***not all*** multiplexed streams. This assumes that ideally no chunks from different data streams are placed into the same UDP packet.

A related issue with SPDY and HTTP/2.0 over TCP is that one and the same TCP congestion avoidance mechanism (e.g. the congestion window) affects all multiplexed data streams. In comparison to the base case of running parallel HTTP over TCP connections between browser and server, then if we have say 6 parallel TCP connections, each connection gets controlled by its own, dedicated dynamic execution of TCP congestion avoidance mechanism. Thus, even if one TCP connection decides to slow down, the other TCP connections may still progress at full speed [5].

## Google engineers' top level design challenges

Google engineers rapidly experimenting with QUIC [2] is about the same as saying, they are experimenting with improvements over both TLS/SSL and TCP. In this regard, they are not the only ones. Deficiencies of TCP are well known, and various TCP improvements have been proposed over the last years. Similarly, it is well known, that TLS/SSL has not been designed with the goal of minimising latency. The options Google engineers had can be summarised in below table.

| Problem | Solution A | Solution B |
|---|---|---|
| Related to security: Latency issues with TLS | Change TLS.<br>*Judged by Google as not a fast route to market [5].* | Create a new protocol that includes features of TLS and improves over TLS → *QUIC* |
| Related to reliable message transport: Issues with TCP | Change TCP.<br>*Judged by Google as not a fast route to market [5].* | Create a completely new substitute for TCP, a new, better, reliable transport layer.<br>*Judged by Google as completely infeasible as middleboxes block traffic unless it is TCP or UDP [5]. → Emulate reliable transport in QUIC as well.* |

As a conclusion from above table, it follows that as QUIC itself cannot be usefully transported over the Internet Protocol, the only remaining choice of transport protocol was UDP. The fact that UDP permits out-of-order delivery of packets is used to avoid stalling of multiplexed QUIC streams [5].

To summarise, why the designers of QUIC are not simply changing TLS and TCP:

First, if Google changed some TCP features today, they estimate it would take 5-15 years until a significant adoption would be noticeable in the market. The prime reason is that TCP is built directly into the kernel of operating systems, and deployed operating system versions tend to have a long life-cycle in the market [5]. One may argue this argument is correct at least for desktops (Windows, Linux), though it may be less applicable for the faster pace in smartphones. Based on this argument, modifying TCP was ruled out.

Second, modifying TLS is not considered a fast route to market either because deployment and iteration is slow [2].

## Is the result, namely QUIC, unique or similar to other approaches taken elsewhere

As explained above, the QUIC designers ended up with using UDP and creating mechanisms on top of UDP which allow: (1) secure transfer - in the end over UDP, (2) multiplexing of application data streams. Similar goals are achieved by the protocol stack SCTP over DTLS over UDP. Datagram Transport Layer Security (DTLS) is the TLS variant used over unreliable UDP connections [15]. Stream Control Transmission Protocol (SCTP) provides multiplexing of multiple application data streams [14]. The reason why the choice was not to re-use or modify SCTP/DTLS boils down to the argument that it would be difficult to incorporate the latency-reducing innovations as used in QUIC into existing standards in any timely fashion [4], [5].

## How is Google positioning QUIC

Robbie Shade from Google says that "SPDY/HTTP2 is already a significant step forward, QUIC is a natural extension of this research" [2]. He argues there are two ways to get the innovation into the market: (1) QUIC makes headway on its own (as SPDY has done) versus (2) TCP and TLS over time

leverage the lessons learned and insight gained from QUIC. Jim Roskind sees "wide deployment of this protocol as being plausible within 1-2 years" [4]. That could mean 2015+.

## What's the status of QUIC and its performance

QUIC was first added to the Chrome Canary channel in June 2013 (where the most bleeding-edge, though sometimes buggy, official version of the Chrome browser is available). Google has also developed a prototype QUIC server, able to speak to a QUIC-enabled Chrome browsers. As of Aug 2014, QUIC is not 'commercially' deployed yet, but available in both the Chrome Canary and Developer channel. Its status is an "experimental implementation", which is accessible to interested parties to try out and use through some versions of the Google Chrome browser. The more people try out QUIC, the more Google can learn how QUIC performs in real-world conditions. Neither the documentation nor the Google implementation of QUIC is stable yet. QUIC has also been introduced in the Opera browser. We are not aware of other companies having implemented QUIC yet.

As per [5], Google is still lacking evidence about QUIC's performance and how much it will actually reduce latency in different environments (e.g. 3G, 4G, 5G, Wi-Fi). The fact that QUIC is an experimental protocol is also underpinned by the following statement in its QUIC Design document [4]: "[Hard Problem: We need to nail down a plan.] There is a fundamental problem that….".

## How do the protocol stacks compare for SPDY, HTTP/2.0 and QUIC

Google's Jim Roskind sums this up nicely in [3]: QUIC collapses and reuses protocol layers.

An illustration of how the protocol stacks may be compared is shown in Figure 1.

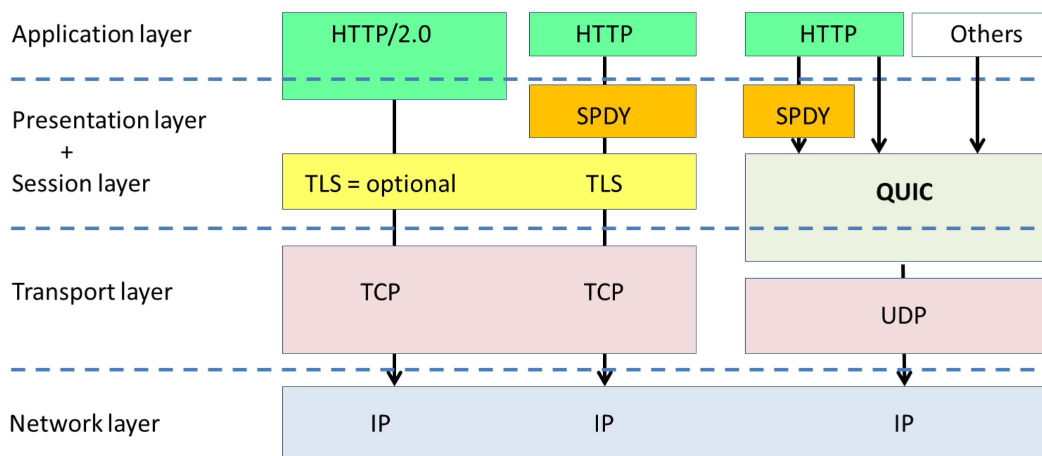


**Figure 1:** Simplified interpretation of protocol stacks for HTTP/2.0 over TLS, SPDY and QUIC.

QUIC is shown to overlap with the scope of TCP and TLS, as it includes features that are similar to TLS and simulates the in-order reliable packet transmission and congestion avoidance mechanisms of TCP.

## Is it fair to compare QUIC directly with SPDY and HTTP/2.0

A look at the protocol stacks tells us that QUIC is not an alternative or substitute of SPDY or HTTP/2.0 per se. It is better to think of QUIC combining some features or mechanisms of

- SPDY, TLS, and TCP

in a single protocol layer.

## How does QUIC work in a nutshell

QUIC can be summarised as follows: It is supposed to be a better replacement for a combination of features of TLS and TCP and includes a number of innovations which in particular reduce latency and improve the performance in situations of unavoidable packet loss. QUIC "effectively replaces TLS and TCP out from under SPDY" [3].

It is basically a tunnelling protocol which runs on top of the connectionless UDP [4]. The data streams multiplexed together in QUIC can be seen as "nearly equivalent to independent TCP connections".

A bi-directional *QUIC connection* is identified by a unique Connection Identifier. Several QUIC connections that are established to the same server (same IP address) and same UDP port (e.g. 80, the UDP port for HTTP) are kept apart by the server based on the Connection Identifier. Although the source UDP port and source IP address may change over time (due to NAT), the Connection Identifier remains constant over the lifetime of the connection [4].

Each QUIC connection can carry a number of multiplexed logical data *streams*.

A first-ever QUIC connection between a client and a server is established by performing a cryptographic handshake, after which encrypted data packets are sent. A repeat-connection will only send an unencrypted cryptographic hello message to the server and then immediately start to send encrypted data packets belonging to different data streams without waiting for any server response.

A *QUIC session* corresponds to a sequence of UDP packets sent over a single QUIC connection [6]. A *stream* corresponds to a bi-directional flow of bytes over a virtual channel within a QUIC session [6].

Each bidirectional data stream is partitioned into *stream frames* by the sender, and one or more stream frames are placed into a UDP packet, which is routed through the Internet. A UDP packet should hold data only from a single stream to avoid head-of-line blocking with other streams of the same QUIC connection [4]. Each stream frame's header includes information about the associated stream number and the starting offset within the stream for the contained data [4]. Steams are open and closed using flags in the stream frame header.

Apart from *stream frames*, a number of *other frame types* are supported, e.g. an ack frame for the receiver to acknowledge received QUIC packets. Stream frames can be of variable length [6].

We conclude that each UDP packet therefore contains 1 or more frames (either frames from streams or other kinds of frames), and that frames bundled into a QUIC packet are mapped into a UDP packet.[1]

When an end point receives a UDP packet, it removes the UDP header and decrypts the payload using the cryptographic context associated with that QUIC connection's Connection Identifier [4, Section Steady State]. The frames contained in various packets are re-assembled to re-create the streams for application level protocols or frames are used by QUIC itself (e.g. for control purposes).

Streams can be instigated by either the client or the server. Stream 1 is created by the client and reserved for cryptographic negotiation.

As QUIC packets are shipped in UDP packets and UDP packets can be delivered out-of-order to the receiver, QUIC marks each QUIC packet with a sequence number, so receivers can watch for duplicate packets and communicate to the sender which QUIC packets got lost.

[1] The relation between QUIC packet and UDP packet lacks some clarity. Note that the QUIC packet has encrypted payload, where the QUIC packet sequence number is used as basis for the Initialization Vector to decrypt the payload. Also alignment of encryption blocks with UDP packet boundaries is sought. Together this points towards a QUIC packet being mapped into exactly one UDP packet.

An example use would be that a user visits the bbc.co.uk web site. The browser opens a QUIC connection to the server which remains in place until the user navigates completely away from this web site.

Figure 2 provides an illustration of how data from two different streams could be mapped into UDP packets.

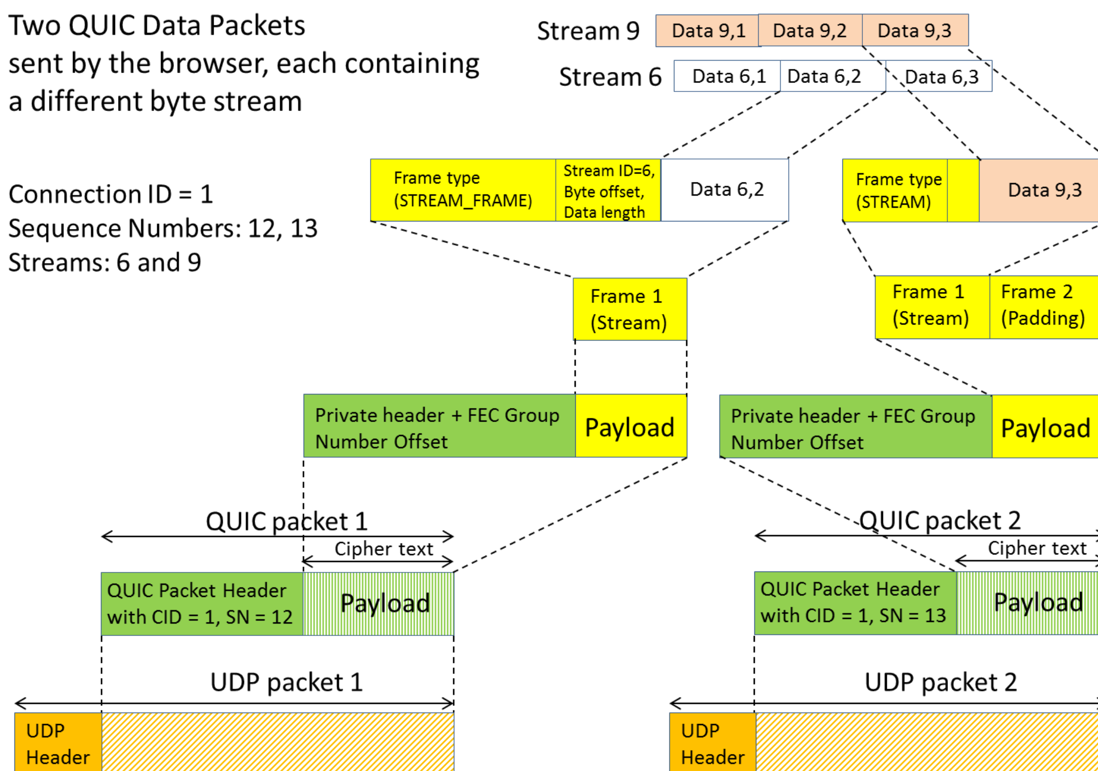


**Figure 2:** A simplified example of how chunks of two application data byte streams could be mapped into QUIC and UDP packets.

## How can QUIC be used – relationship to SPDY

According to the QUIC Design document, it should be possible to run different application protocols over QUIC. For example, a QUIC API is considered which mimics a TCP socket. That implies, any application protocol that makes use of TCP could make use of QUIC as well [4, section API Concepts].

When SPDY is run over QUIC, then SPDY streams are mapped to QUIC streams. See the section SPDY Layering over QUIC in [4]. Data sent over the QUIC stream then consists of SPDY headers and body.

If a server is supporting QUIC, it can announce itself to the client through sending an Alternate-Protocol header which advertises its capability.

## How do characteristics of SPDY/HTTP/2.0 over TLS over TCP compare to SPDY over QUIC

The following table provides some non-exhaustive comparison.

| Area | Characteristic | SPDY or HTTP/2.0 over TLS over TCP | SPDY over QUIC over UDP |
|---|---|---|---|
| Security | End-to-end encryption, | Yes (via TLS) | Yes (via QUIC crypto). |

| | | | |
|---|---|---|---|
| | authentication | | Argued to be comparable to TLS [2] and DTLS [4]. |
| Latency | Latency when establishing connectivity | High<br>3 RTT (from TCP + TLS) until HTTP request is sent [2] | Low (often 0-RTT) |
| | Avoidance of packet loss | TCP congestion window mechanism | Packet pacing in QUIC |
| | Reduction in transmission latency | No | Use of packet error correction (FEC) to avoid packet re-transmissions |
| | Head of line blocking | Present due to TCP HOL blocking, impacts all multiplexed SPDY or HTTP/2.0 streams [2] | Less head-of-line blocking, maximum 1 out of n streams impacted, due to UDP [2] |
| Persistent connections | Persistent connections when changing access technology | No | Yes, through QUIC Connection Identifier, even when source IP and source UDP port change. |
| Congestion control, flow control | Congestion control mechanisms | Yes, built into TCP.<br>Flow control is done for all data streams together. | Added to QUIC, whereby different mechanisms can be plugged in or switched on. TCP Cubic as default algorithm. Other alternative: bandwidth estimation to drive packet pacing [3]. Individual stream flow control is provided. |
| Multiplexing | Multiplexing of application data streams (like HTTP transactions) | Yes, done in SPDY layer | Yes, done in QUIC layer |
| Reliable transport | In-order reliable packet delivery | Yes, achieved by TCP | Yes, achieved by QUIC which uses packet sequence numbers [3]. |

## What are key design features and innovations in QUIC (and not in SPDY)

QUIC operates on the OSI session and presentation layers. Given that QUIC shall be used e.g. for secure Internet browsing between browser clients and web servers, QUIC has to take care of all necessary features for end-to-end security and reliable, in-order message transfer, because the underlying UDP protocol caters neither for security (e.g. encryption) nor for reliable, in-order message transfer. Some innovative ideas baked into QUIC include:

- **Bandwidth estimation** in each direction to feed advanced congestion avoidance algorithms. Mind, in the case of SPDY or HTTP/2.0 over TCP, TCP takes care of congestion avoidance [5].

- **Pacing UDP packet transmissions** evenly to help avoid packet loss due to congestion in routers [5]. Google have gathered experimental results which show that the combination of packet pacing and use of forward error correction packets dramatically reduces the likelihood of required packet retransmissions in case of UDP packet loss [3].

- **Use of forward error correction codes on packet level** in order to reduce the likelihood of needing to retransmit packets in case of UDP packet loss [5]. Note, that in the case of SPDY

and HTTP/2.0, the loss of a TCP packet can have a dramatic impact on the overall speed and performance of SPDY and HTTP/2.0 due to the fact, that TCP is a reliable protocol that uses retransmission of lost TCP packets. In the case of QUIC, the Google designers bet on lost packet recovery to work more often than not by means of forward error correction FEC, thereby reducing the amount of packet retransmissions and thus end-to-end latency. Use of FEC requires extra FEC packets and thus more bandwidth. Therefore, it's a trade-off between bandwidth need and latency [3]. Whether error correction works depends on how bursty UDP packet loss is in practice. Google gather experimental data to prove FEC is beneficial.

- **Alignment of cryptographic block boundaries with UDP packet boundaries** [5]. In contrast, TLS/SSL encrypted data blocks don't match IP packet boundaries [3]. This means, that in case UDP packet 1 gets lost under way, UDP packet 2 which arrives at the receiver can still be properly decrypted without any cryptographic dependency on a previously sent packet, thus no waiting for that packet to arrive. Again, this reduces latency. Though this in-order dependency of packets appears resolved in TLS 1.1 and DTLS, it comes at a cost of more bytes per packet, which QUIC wants to avoid [4].

- **Probabilistic approaches and speculative client requests and server responses**. For example, the speculation is made that the server's public key is unchanged since last contact by the client. This is important to achieve a reduction to zero round-trip-time at connection establishment through a client [3], [4]. Only if the server's key has changed, the client will have to re-transmit what it optimistically has previously sent.

- **Proactive speculative retransmission/redundant transmission**: A packet is retransmitted even in the absence of evidence for any UDP packet loss in order to reduce the likelihood of the receiver not receiving it at all, again for the sake of reducing latency [4].

- **QUIC APIs to higher layer application protocols** shall allow the application to tell QUIC the desired characteristics of each stream (e.g. reliability, priority level). Further, an application shall be able to query the connection status (e.g. to learn about an estimate of round-trip-time, estimate of bandwidth, byte queue size at the sender) and to receive event notifications from QUIC.

## Message formats in QUIC

The format of messages used by QUIC including the exact definition of header fields is defined in the still evolving Wire Specification [6]. This document in particular appears to be far from final and is certainly not stable yet. An illustration of some message formats is provided in [8].

## Encryption with QUIC

As it stands today, QUIC requires end-to-end encryption. The official reason stated by Google engineers is that unless they encrypt traffic, middle boxes, either deliberately or unwittingly would corrupt the transmission when those boxes "try to 'helpfully' filter or 'improve' the traffic" [5].

As a consequence HTTP and HTTPS over QUIC will both be encrypted [3], though the designers say "we may (TBD) have reduced certificate validation on UDP port 80", i.e. for HTTP [4].

## How does QUIC compare to HTTP, SPDY, HTTP/2.0 over encrypted connections

QUIC can transport various application layer protocols including SPDY, HTTP and HTTP/2.0. In all cases, it establishes a secure QUIC connection between client and server.

There are similarities between the cryptographic part of QUIC and TLS. For example, QUIC initiates a first-ever connection from a client to a server using a ClientHello message. This is similar to a TLS ClientHello message. The server responds with a ServerHello, similar as in TLS, with a server certificate.

All QUIC packets are authenticated and encrypted apart from the initial Hello message which is not encrypted [4].

## Implications of QUIC for operator middle boxes

Generally, the content of a QUIC connection won't be visible to a middle box, as the connection is protected via encryption.

An encrypted connection via TLS for HTTP, SPDY or HTTP/2.0 can be inspected using SSL inspection. One would assume the same should be possible for a QUIC connection by intercepting the QUIC Hello message, though details are for further study.

Whether it's more difficult to do any traffic analysis for a QUIC connection compared to a SPDY or encrypted HTTP or HTTP/2.0 connection is difficult to judge. A statement from [4] is worth quoting here: "We should provide support to pad packets to reduce vulnerability to traffic analysis", though the author then concedes that anti-traffic-analysis measures are not yet sufficiently well understood to bake them into QUIC.

## QUIC features

In the following table a number of important features implemented in QUIC are listed.

| Feature | Comment | Reference |
|---|---|---|
| Adaptive congestion control | QUIC needs to take care of this as UDP doesn't | [5] |
| Automatic retransmission of lost UDP packets | QUIC needs to somehow emulate the retransmission mechanisms of TCP | [5] |
| End-to-end encryption for both HTTP and HTTPS | Similar to TLS, but different!<br>UDP port 80 (HTTP) and UDP port 443 (HTTPS) seem reserved for QUIC [3], [12]. | [2] |
| Forward error correction on packet level, dynamically adjustable | Not used in SPDY | [2], [4] |
| Monitoring of inter-packet spacing | One-way packet transit times are measured. This is used to estimate bandwidth and feed into a packet pacing algorithm | [3] |
| Packet pacing | Not required for a protocol that makes use of TCP. Needed here to avoid congestion with UDP | [2] |
| 0-RTT connections | At least after a browser has first contacted a server, further connection setup can happen with no extra round-trip times. | [2] |
| Implementable outside operating system in the application space | QUIC is, in contrast to TCP, not part of the operating system kernel | [3] |
| Header compression | As SPDY does | [3] |

## Challenges remaining for Google with QUIC

At least the following challenges remain before QUIC will enjoy massively wider adoption:

- According to Google, they need to show that QUIC is compliant to standards from the PCI Security Standards Council in particular to be able to handle credit card transactions over QUIC [3].

- Algorithms for congestion control need to be proven, including bandwidth estimation and pacing of UDP packets. In this regard, the statement in [4] that “we may need some better hooks deep into the operating system to better facilitate accurate pacing in OS level buffering” might be in contradiction with the stated goal to deploy fast without need to touch client operating systems.
- Effects of network address translation (NAT) need to be further studied, e.g. premature unbinding of UDP port mappings and the optimised use of UDP keep-alive packets to revitalise NAT bindings. See [4, section on Overcoming NATs).

## References

[1] Home page for QUIC in the Google Chromium project: http://www.chromium.org/quic

[2] Robbie Shade, Google: Presentation about QUIC at Google Developers Live, Feb 2014: https://docs.google.com/presentation/d/13LSNCCvBijabnn1S4-Bb6wRIm79gN6hnPFHByEXXptk/edit?pli=1#slide=id.g176a9a2e9_0143

[3] Jim Roskind, Google presentation about QUIC at IETF-88, 7 Nov 2013: http://www.ietf.org/proceedings/88/slides/slides-88-tsvarea-10.pdf

[4] QUIC design document. Provides the design rationale on currently 50 pages and is an excellent piece of work: https://docs.google.com/a/google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/preview?sle=true

[5] QUIC FAQ: http://www.chromium.org/quic/quic-faq

[6] QUIC Wire Layout Specification. This document appears to be much work-in-progress: https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U/edit?pli=1

[7] QUIC Crypto specification: https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit?pli=1

[8] ixia blog, article covering QUIC message formats: http://blogs.ixiacom.com/ixia-blog/quic-or-youll-miss-it/

[9] QUIC on GitHub: https://github.com/gburd/quic

[10] QUIC Prototype Protocol Discussion group: https://groups.google.com/a/chromium.org/forum/#!forum/proto-quic

[11] QUIC source code on Google’s Chromium project: http://src.chromium.org/viewvc/chrome/trunk/src/net/quic/

[12] List of TCP and UDP port numbers: http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

[13] AT&T paper raising issues with SPDY: http://conferences.sigcomm.org/co-next/2013/program/p303.pdf

[14] Stream Control Transmission Protocol SCTP: http://tools.ietf.org/html/rfc4960

[15] Datagram Transport Layer Security DTLS: http://tools.ietf.org/html/rfc6347