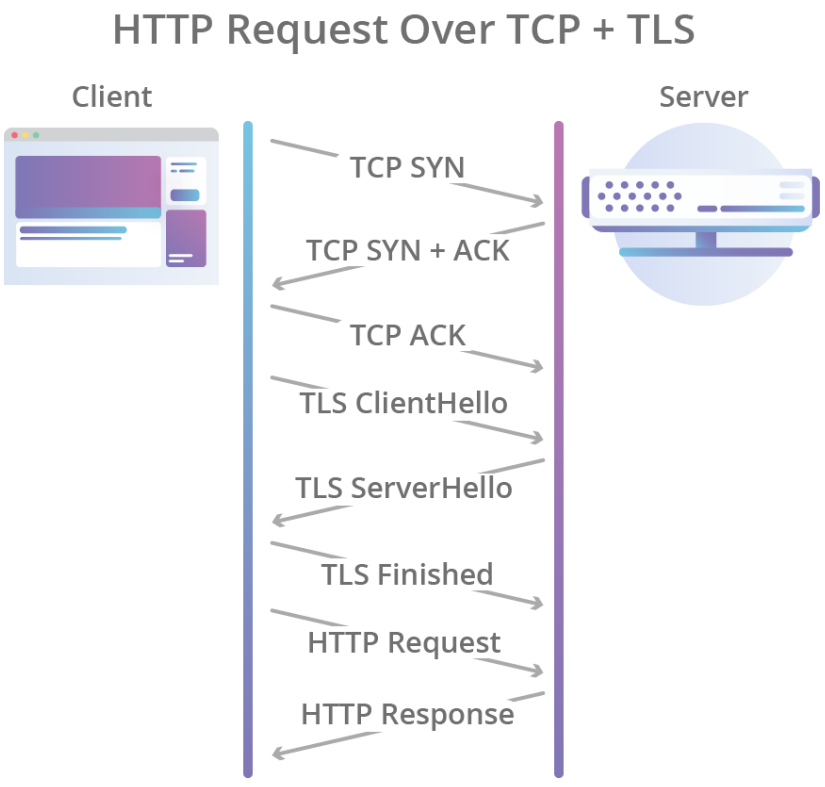
# HTTP/3: the past, the present, and the future

Before we dive deeper into HTTP/3, let's have a quick look at the evolution of HTTP over the years in order to better understand why HTTP/3 is needed.

It all started back in 1996 with the publication of the HTTP/1.0 specification which defined the basic HTTP textual wire format as we know it today (for the purposes of this post I'm pretending HTTP/0.9 never existed). In HTTP/1.0 a new TCP connection is created for each request/response exchange between clients and servers, meaning that all requests incur a latency penalty as the TCP and TLS handshakes are completed before each request.

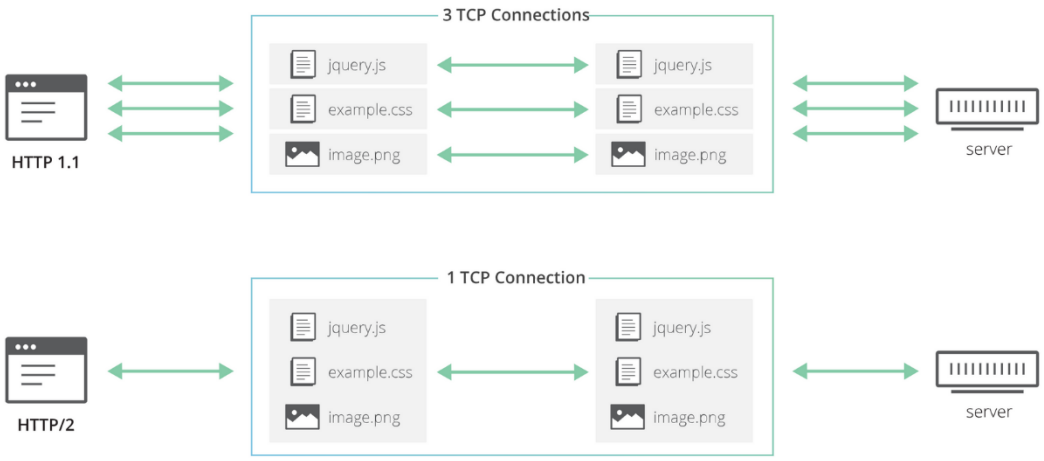


Worse still, rather than sending all outstanding data as fast as possible once the connection is established, TCP enforces a warm-up period called “slow start”, which allows the TCP congestion control algorithm to determine the amount of data that can be in flight at any given moment before congestion on the network path occurs, and avoid flooding the network with packets it can’t handle. But because new connections have to go through the slow start process, they can’t use all of the network bandwidth available immediately.

The HTTP/1.1 revision of the HTTP specification tried to solve these problems a few years later by introducing the concept of “keep-alive” connections, that allow clients to reuse TCP connections, and thus amortize the cost of the initial connection establishment and slow start across multiple requests. But this was no silver bullet: while multiple requests could share the same connection, they still had to be serialized one after the other, so a client and server could only execute a single request/response exchange at any given time for each connection.

As the web evolved, browsers found themselves needing more and more concurrency when fetching and rendering web pages as the number of resources (CSS, JavaScript, images, …) required by each web site increased over the years. But since HTTP/1.1 only allowed clients to do one HTTP request/response exchange at a time, the only way to gain concurrency at the network layer was to use multiple TCP connections to the same origin in parallel, thus losing most of the benefits of keep-alive connections. While connections would still be reused to a certain (but lesser) extent, we were back at square one.

Finally, more than a decade later, came SPDY and then HTTP/2, which, among other things, introduced the concept of HTTP “streams”: an abstraction that allows HTTP implementations to concurrently multiplex different HTTP exchanges onto the same TCP connection, allowing browsers to more efficiently reuse TCP connections.




But, yet again, this was no silver bullet! HTTP/2 solves the original problem — inefficient use of a single TCP connection — since multiple requests/responses can now be transmitted over the same connection at the same time. However, all requests and responses are equally affected by packet loss (e.g. due to network congestion), even if the data that is lost only concerns a single request. This is because while the HTTP/2 layer can segregate different HTTP exchanges on separate streams, TCP has no knowledge of this abstraction, and all it sees is a stream of bytes with no particular meaning.

The role of TCP is to deliver the entire stream of bytes, in the correct order, from one endpoint to the other. When a TCP packet carrying some of those bytes is lost on the network path, it creates a gap in the stream and TCP needs to fill it by resending the affected packet when the loss is detected. While doing so, none of the successfully delivered bytes that follow the lost ones can be delivered to the application, even if they were not themselves lost and belong to a completely independent HTTP request. So they end up getting unnecessarily delayed as TCP cannot know whether the application would be able to process them without the missing bits. This problem is known as “head-of-line blocking”.

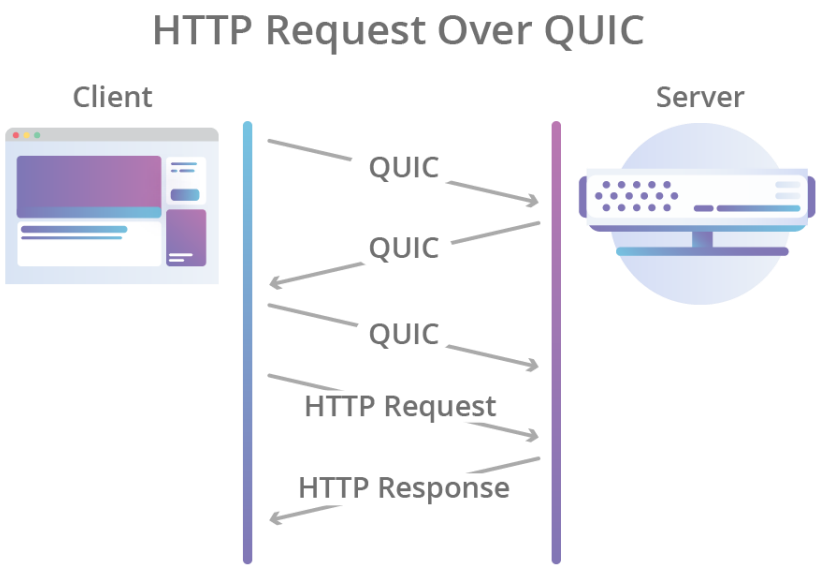
# Enter HTTP/3

This is where HTTP/3 comes into play: instead of using TCP as the transport layer for the session, it uses QUIC, a new Internet transport

protocol, which, among other things, introduces streams as first-class citizens at the transport layer. QUIC streams share the same QUIC connection, so no additional handshakes and slow starts are required to create new ones, but QUIC streams are delivered independently such that in most cases packet loss affecting one stream doesn't affect others. This is possible because QUIC packets are encapsulated on top of UDP datagrams.

Using UDP allows much more flexibility compared to TCP, and enables QUIC implementations to live fully in user-space — updates to the protocol's implementations are not tied to operating systems updates as is the case with TCP. With QUIC, HTTP-level streams can be simply mapped on top of QUIC streams to get all the benefits of HTTP/2 without the head-of-line blocking.

QUIC also combines the typical 3-way TCP handshake with TLS 1.3's handshake. Combining these steps means that encryption and authentication are provided by default, and also enables faster connection establishment. In other words, even when a new QUIC connection is required for the initial request in an HTTP session, the latency incurred before data starts flowing is lower than that of TCP with TLS.



But why not just use HTTP/2 on top of QUIC, instead of creating a whole new HTTP revision? After all, HTTP/2 also offers the stream multiplexing feature. As it turns out, it's somewhat more complicated than that.

While it's true that some of the HTTP/2 features can be mapped on top of QUIC very easily, that's not true for all of them. One in particular, HTTP/2's header compression scheme called HPACK, heavily depends on the order in which different HTTP requests and responses are delivered to the endpoints. QUIC enforces delivery order of bytes within single streams, but does not guarantee ordering among different streams.

This behavior required the creation of a new HTTP header compression scheme, called QPACK, which fixes the problem but requires changes to the HTTP mapping. In addition, some of the features offered by HTTP/2 (like per-stream flow control) are already offered by QUIC itself, so they were dropped from HTTP/3 in order to remove unnecessary complexity from the protocol.

# HTTP/3, powered by a delicious quiche

QUIC and HTTP/3 are very exciting standards, promising to address many of the shortcomings of previous standards and ushering in a new era of performance on the web. So how do we go from exciting standards documents to working implementation?

Cloudflare's QUIC and HTTP/3 support is powered by quiche, our own open-source implementation written in Rust.

You can find it on GitHub at github.com/cloudflare/quiche.

We announced quiche a few months ago and since then have added support for the HTTP/3 protocol, on top of the existing QUIC support. We have designed quiche in such a way that it can now be used to implement HTTP/3 clients and servers or just plain QUIC ones.


You can find it on GitHub at github.com/cloudflare/quiche.

We announced quiche a few months ago and since then have added support for the HTTP/3 protocol, on top of the existing QUIC support. We have designed quiche in such a way that it can now be used to implement HTTP/3 clients and servers or just plain QUIC ones.