# HTTP/3 - how it performs compared to HTTP/2?

HTTP is the primary application-level protocol used for the web. Its third iteration has already been developed for a couple of years. In June 2022, it became a [proposed standard](https://datatracker.ietf.org/doc/html/rfc9114). Therefore, it is a perfect opportunity to test how it performs, as HTTP/3 support is already built in NGINX binaries since version 1.25.0. We use NGINX at Kiwee as a proxy server for many projects, including [Shopware 6](https://kiwee.eu/blog/is-shopware-6-the-next-generation-ecommerce-platform/)-based online stores.

## How HTTP/3 is different from its predecessors

Previous versions of HTTP used TCP as the transport protocol. TCP, though has certain inefficiencies, particularly in handling packet loss, which can block an entire stream of data (known as head-of-line blocking) until the lost packet is retransmitted. HTTP/3 adopted QUIC, a new transport protocol initially designed and developed at Google. QUIC is based on UDP. It aims to eliminate the issue of head-of-line blocking by allowing individual streams to fail without impacting the entire connection.

{{ graph }}

## More improvements in HTTP/3

Another one is security. QUIC includes built-in TLS support. In previous HTTP versions, the security layer was separate. With QUIC, encryption, and data transfer can happen simultaneously, reducing latency. QUIC currently adopts the latest version of TLS - 1.3.

Multiplexing - both HTTP/2 and HTTP/3 support it. Multiple requests can be made in parallel over a single connection. However, in HTTP/2, a lost packet still impacts all the streams due to TCP’s head-of-line blocking. QUIC’s stream multiplexing uses UDP. Hence, it does not suffer from this problem, leading to better performance when packet loss occurs.

{{ graph }}

Faster initial connection - QUIC has TLS built-in, so it can handshake once, when the connection is being established. Unlike HTTP/2, where TCP and TLS each need to handshake individually.

{{ graph }}

Connection migration - QUIC supports connection migration, which means that if a user changes their network interface ( e.g., from Wi-Fi to mobile data), QUIC can continue using the same connection without interruption. Conversely, TCP ties connections to the specific IP address, so changing networks often means starting a new connection.

Zero Round Trip Time Resumption (0-RTT) - QUIC allows data to be sent by a client before a connection is formally established. It can speed up the first request to the web server.

## HTTP/3 vs HTTP/2 Benchmark

The primary test website was a static CMS page. A static page eliminates additional latency caused by executing an application, connection to a database, or other external services.

### Testing tool

The [HTTP testing script](https://github.com/KiweeEu/http3-test) incorporates Puppeteer - a Node.js library that allows running Chrome in headless mode and accessing the DevTools metrics, particularly metrics from the network section.

### Testing methodology

The test page’s complete download time was measured, including all linked assets like fonts, images, CSS, and JS files, but excluding requests initiated by a JavaScript code.

The given page was downloaded 50 times in enforced HTTP/3 mode (to ensure that even the first request used HTTP/3) and 50 times with disabled QUIC support.

To make all tests consistent, for each test iteration, always a fresh container was spun up.

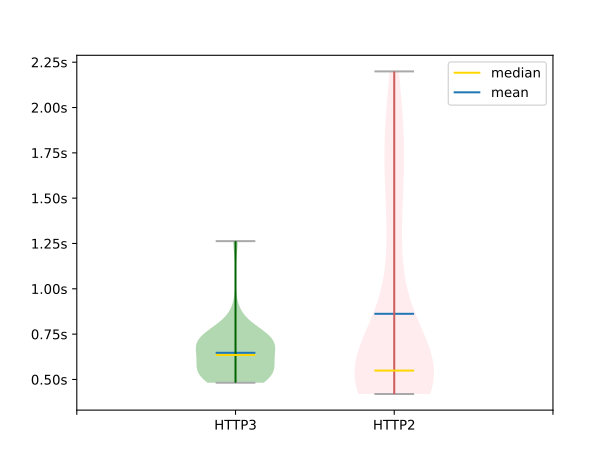
### Test locations

The target website was hosted in a data center located in Northern Germany. The tests were executed from:

* Three different data centers of Hetzner Cloud (Nürnberg, Germany; Helsinki, Finland; Ashburn, Virginia, US);
* A laptop connected to the internet via fiber 1 Gbps;
* A laptop connected via a mobile hotspot (Wrocław, Poland). Finally, one test series simulated poor-quality networking conditions by adding extra 100ms latency and about 15% packet loss.

### Test results

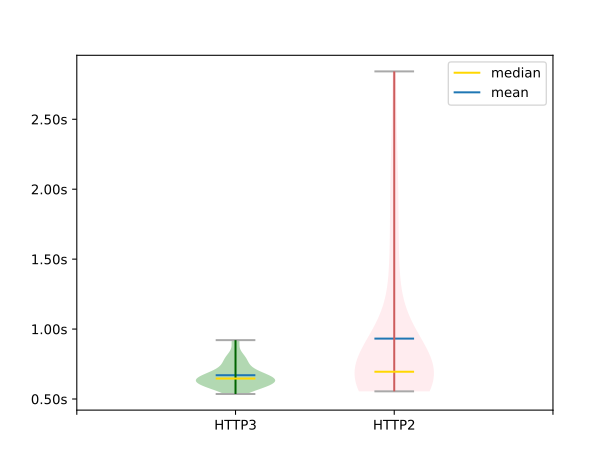
#### Page load time from Nürnberg, Germany - Hetzner Cloud



Page Load time from Nürnberg, Germany

| Nürnberg, DE | HTTP/3 | HTTP/2 |
| --- | --- | --- |
| median | 0.6350 | 0.5490 |
| mean | 0.6472 | 0.8623 |
| fastest 90th percentile median | 0.6115 | 0.5330 |
| fastest 90th percentile mean | 0.6148 | 0.7364 |
| min | 0.4820 | 0.4200 |
| max | 1.2630 | 2.1990 |

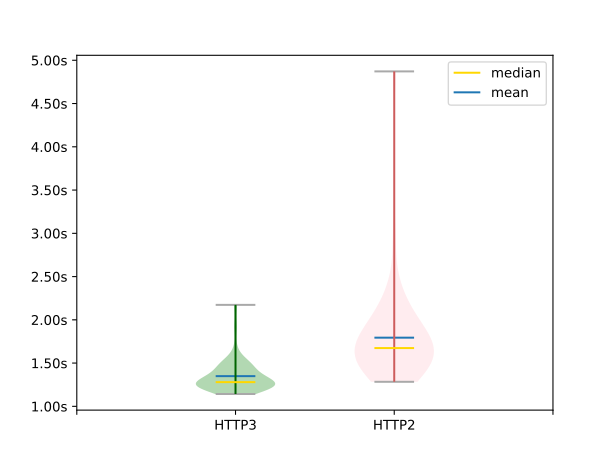
#### Page load time from Helsinki, Finland - Hetzner Cloud



Page load time from Helsinki, Finland

| Helsinki, FI | HTTP/3 | HTTP/2 |
| --- | --- | --- |
| median | 0.6465 | 0.6950 |
| mean | 0.6702 | 0.9317 |
| fastest 90th percentile median | 0.6425 | 0.6550 |
| fastest 90th percentile mean | 0.6489 | 0.7769 |
| min | 0.5360 | 0.5550 |
| max | 0.9210 | 2.8420 |

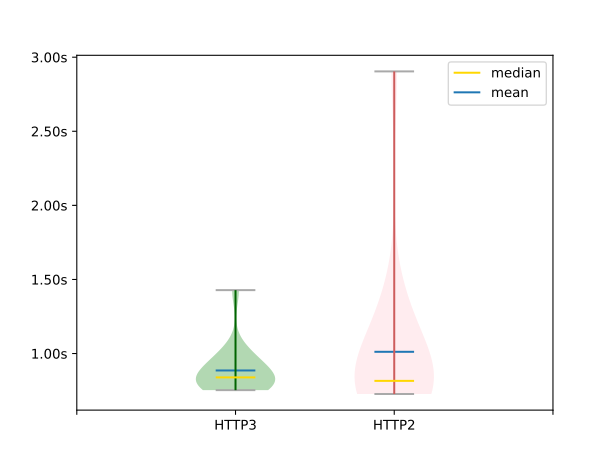
#### Page load time from Virginia, US - Hetzner Cloud



Page load time from Viginia, US - Hetzner Cloud

| Viginia, US | HTTP/3 | HTTP/2 |
| --- | --- | --- |
| median | 1.2800 | 1.6740 |
| mean | 1.3491 | 1.7942 |
| fastest 90th percentile median | 1.2715 | 1.6350 |
| fastest 90th percentile mean | 1.3022 | 1.6609 |
| min | 1.1430 | 1.2840 |
| max | 2.1730 | 4.8710 |

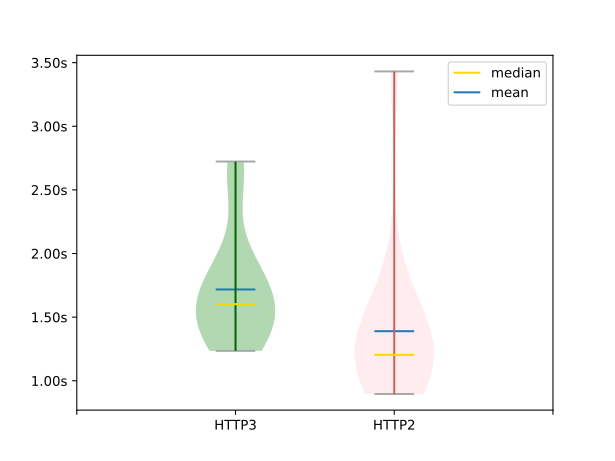
#### Page load time from Wrocław, Poland - fiber 1Gbps



Page load time from Wroclaw, Poland - fiber

| PL (Wroclaw, Fiber) | HTTP/3 | HTTP/2 |
| --- | --- | --- |
| median | 0.8390 | 0.8155 |
| mean | 0.8856 | 1.0117 |
| fastest 90th percentile median | 0.8380 | 0.8095 |
| fastest 90th percentile mean | 0.8409 | 0.8740 |
| min | 0.7530 | 0.7270 |
| max | 1.4280 | 2.9040 |

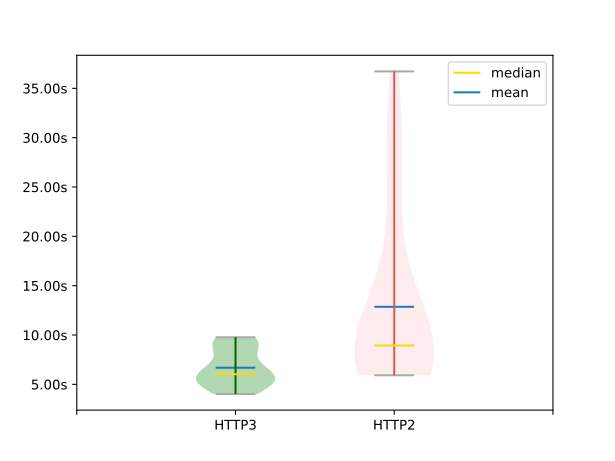
#### Page load time from Wrocław, Poland - mobile LTE



Page load time from Wroclaw, Poland - mobile LTE

| PL (Wrocław, LTE) | HTTP/3 | HTTP/2 |
| --- | --- | --- |
| median | 1.6020 | 1.2035 |
| mean | 1.7172 | 1.3895 |
| fastest 90th percentile median | 1.5850 | 1.1815 |
| fastest 90th percentile mean | 1.5624 | 1.2130 |
| min | 1.2350 | 0.8960 |
| max | 2.7230 | 3.4310 |

#### Page load time from Wrocław, Poland - mobile low quality



Page load time from Wroclaw, Poland - mobile low quality

| PL (Wroclaw, mobile ~15% packet loss, +100ms latency) | HTTP/3 | HTTP/2 |
| --- | --- | --- |
| median | 6.0280 | 8.9400 |
| mean | 6.6856 | 12.8554 |
| fastest 90th percentile median | 5.8070 | 8.2430 |
| fastest 90th percentile mean | 6.3260 | 10.2258 |
| min | 4.0230 | 5.9210 |
| max | 9.7740 | 36.7130 |

HTTP/3 wins unquestionably in cross-continental connections (US East Coast - Germany) - 23% faster download (average). It also outperforms HTTP/2 when clients use unstable mobile networks with high latency and packet loss - 54% faster download (average).

A close distance between the client and the server, stable network conditions still show a slight average advantage of HTTP/3. However, medians are lower for HTTP/2.

A Jupiter Notebook that [aggregates the results](https://github.com/KiweeEu/http3-test-results) can be found on GitHub.

## How to enable HTTP/3 for your web application

There are several solutions available depending on your infrastructure restrictions and limitations.

In the case of infrastructure that does not have the ability to replace or update the front HTTP server, and yet the simplest solution is to incorporate a third-party CDN service that supports HTTP/3. Most of them already do, including Cloudflare, Fastly, AWS Cloudfront, or Google Cloud CDN.

Use a server that supports HTTP/3 as your application HTTP server or a proxy server. Note that QUIC cannot work without TLS. Thus it requires a valid TLS certificate. The following servers support HTTP/3.

* [NGINX](https://nginx.org/en/docs/quic.html)
* [Caddy](https://caddyserver.com/docs/caddyfile/options#global-options)
* [LiteSpeed](https://docs.litespeedtech.com/lsws/cp/cpanel/quic-http3/)
* [H2O](https://github.com/h2o/h2o/blob/master/examples/h2o/h2o.conf).

HAProxy load balancer currently only supports HTTP/3 by default in the enterprise version. However, [HAProxy can be compiled](https://www.haproxy.com/blog/how-to-enable-quic-load-balancing-on-haproxy#how-to-enable-quic-in-haproxy) from the sources with QUIC enabled option.

Apache sadly still has no QUIC support in its roadmap. There is a [feature request](https://bz.apache.org/bugzilla/show_bug.cgi?id=64462) pending.

## Enable HTTP/3 with NGINX in a Docker container

The official NGINX images, like its binaries for download, have built-in QUIC support since version 1.25.0. To enable HTTP/3, follow the steps below:

1. Enable TLS 1.3. QUIC works with the latest 1.3 version only. Append TLSv1.3 to the ssl\_protocol directive, like the following:

ssl\_protocols TLSv1.2 TLSv1.3;

1. Reuse port 443 for QUIC protocol:

listen 443 quic reuseport;

Note that reuseport cannot be used more than once per host. So when having more than one virtual host listening on the same IP address, only one can have the reuseport clause.

1. Add an Alt-Svc header to tell the browser that HTTP/3 is available. The very first request always uses HTTP/2. Once getting the first response containing the Alt-Svc, all subsequent requests already use HTTP/3.

add\_header Alt-Svc 'h3=":443"; ma=86400';

1. HTTP/3 implementation in NGINX does not forward the Host header as well as the proxy-specific ones, such as x-forwarded-for, x-forwarded-host, x-forwarded-proto, x-forwarded-port, x-forwarded-prefix. They need to be explicitly set if your application requires them.
2. Expose both TCP and UDP port 443 from the NGINX container. With docker run as below:

docker run -p 80:80 -p 443:443/tcp -p 443:443/udp nginx

Alternatively, in the Dockerfile:

...  
EXPOSE 80 # without the explicit protocol name the default one is TCP  
EXPOSE 443/tcp  
EXPOSE 443/udp

In docker-compose.yml, it is:

...  
ports:  
- "80:80"  
- "443:443/tcp"  
- "443:443/udp"

1. Check your firewall rules to see whether the UDP port 443 is open.

How to better inform the browser about HTTP/3 availability? Typically, a browser makes the first connection with HTTP/2. Then, it finds out that HTTP/3 is available when it receives the Alt-Svc response header with the value matching the pattern:

Alt-Svc 'h3=":<port>"; ma=<timeout\_seconds>';

There is a solution to make the browser aware even beforehand through a DNS HTTPS record.

yourdomain.com 3600 IN HTTPS 1 . alpn=”h3,h2”

Unfortunately, this feature is only supported by Firefox so far. However, according to [Chromestatus](https://chromestatus.com/feature/5154357283651584) it is in development for Chrome and Safari.

## Conclusion

HTTP/3 is not as big a step forward as HTTP/2 was over HTTP/1.1 in terms of performance boost. HTTP/2 introduced multiplexing, allowing browsers to download multiple resources simultaneously on a single TCP connection. HTTP/3 introduces evolutionary improvements, particularly for high network latency and packet loss. In such conditions, there is a higher chance of head-of-line blocking occurring for HTTP/2. The tests verified HTTP/3 there performing much better. Even for fast and stable networks it surpassed HTTP/2 a bit.

Is HTTP/3 ready for adoption? In my opinion, yes. Big players like Google, Cloudflare, or Shopify have already used it in production. It could be specifically beneficial for mobile users. A mobile network may be unstable. Therefore, it can turn to an advantage like increasing conversion rates and overall sales for online stores.