# **HTTP /3 От А до Я: основные концепции. Quick**

*После почти пяти лет разработки новый протокол HTTP / 3 приближается к своей окончательной форме. Предыдущие итерации уже были доступны в качестве экспериментальной функции, но вы можете ожидать, что доступность и использование собственно HTTP / 3 увеличатся в 2021 году. Итак, что же такое HTTP / 3? Зачем он понадобился так скоро после HTTP / 2? Как вы можете или должны его использовать? И особенно, как он повышает производительность веб-сайта? Давайте выясним.*

Возможно, вы читали какие-то записи в блогах или слышали выступления на конференциях на эту тему и думаете, что знаете ответы. Вы, вероятно, слышали такие вещи, как: “HTTP / 3 намного быстрее, чем HTTP / 2, когда происходит потеря пакетов”, или “Соединения HTTP / 3 имеют меньшую задержку и требуют меньше времени на настройку”, и, вероятно, “HTTP / 3 может отправлять данные быстрее и может отправлять больше ресурсов параллельно”.

В этих заявлениях и статьях обычно пропускаются некоторые **важнейшие технические детали**, отсутствуют нюансы и, как правило, они верны лишь частично. Часто создается впечатление, что HTTP / 3 - это революция в производительности, в то время как на самом деле это более скромная (но все еще полезная!) **эволюция**. Это опасно, потому что новый протокол, вероятно, не сможет оправдать эти высокие ожидания на практике. Я боюсь, что это приведет к тому, что многие люди в конечном итоге разочаруются, а новички будут сбиты с толку кучей слепо распространяемой дезинформации.

Я боюсь этого, потому что мы видели, как то же самое произошло с HTTP / 2. Это было объявлено как потрясающая революция в производительности, с захватывающими новыми функциями, такими как серверная передача, параллельные потоки и расстановка приоритетов. Мы могли бы прекратить объединять ресурсы, прекратить распределять наши ресурсы по нескольким серверам и значительно упростить процесс загрузки страниц. Веб-сайты волшебным образом стали бы на 50% быстрее одним щелчком переключателя!

Пять лет спустя мы знаем, что [серверный push на самом деле не работает](https://www.ctrl.blog/entry/http2-push-chromium-deprecation.html) на практике, [потоки и расстановка приоритетов часто плохо реализованы](https://github.com/andydavies/http2-prioritization-issues), и, следовательно, (сокращенное) [объединение](https://jakearchibald.com/2021/f1-perf-part-7/#lots-of-little-resources-vs-one-big-resource) [ресурсов](https://twitter.com/yoavweiss/status/1254650804524507136) [и](https://twitter.com/zachleat/status/1055219667894259712) даже сегментированиевсе еще являются хорошей практикой в некоторых ситуациях.

Аналогичным образом, другие механизмы, изменяющие поведение протокола, такие как [подсказки предварительной загрузки](https://developer.mozilla.org/en-US/docs/Web/HTML/Preloading_content), часто содержат [скрытые](https://twitter.com/csswizardry/status/1349681832393109510) [глубины](https://twitter.com/programmingart/status/1351557858354225159) и [ошибки](https://andydavies.me/blog/2019/02/12/preloading-fonts-and-the-puzzle-of-priorities/), что затрудняет их правильное использование.

Поэтому я считаю важным предотвратить распространение подобного рода дезинформации и этих **нереалистичных ожиданий** и в отношении HTTP / 3.

В этой серии статей я расскажу о новом протоколе, особенно о его **характеристиках** производительности, с большим количеством нюансов. Я покажу, что, хотя в HTTP / 3 действительно есть несколько многообещающих новых концепций, к сожалению, их влияние, вероятно, будет относительно ограниченным для большинства веб-страниц и пользователей (но потенциально решающим для небольшого подмножества). HTTP / 3 также довольно сложен в настройке и правильном использовании, поэтому будьте осторожны при настройке нового протокола.

* **Часть 1: История HTTP / 3 и основные концепции**  
  **Эта статья предназначена для людей, не знакомых с HTTP / 3 и протоколами в целом, и в ней в основном обсуждаются основы.**
* [Часть 2: Особенности производительности HTTP / 3](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/)Эта более глубокая и техническая. Люди, которые уже знают основы, могут начать здесь.
* [Часть 3: Практические варианты развертывания HTTP / 3](https://www.smashingmagazine.com/2021/09/http3-practical-deployment-options-part3/)В этой третьей статье из серии объясняются проблемы, связанные с самостоятельным развертыванием и тестированием HTTP / 3. В ней подробно рассказывается, как и следует ли вам изменять свои веб-страницы и ресурсы.

**Примечание**: *Эта серия предназначена в основном для веб-разработчиков, которые не обязательно обладают глубокими знаниями протоколов и хотели бы узнать больше. Тем не менее, он содержит достаточно технических деталей и множество ссылок на внешние источники, чтобы заинтересовать и более продвинутых читателей.*

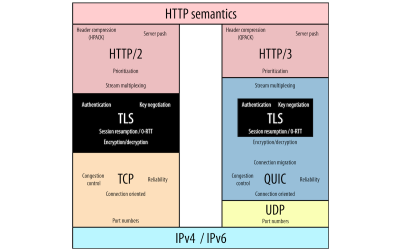
## Зачем Нам Нужен HTTP /3?

Один из вопросов, с которым я часто сталкиваюсь, звучит так: “Зачем нам нужен HTTP / 3 так скоро после HTTP / 2, который был стандартизирован только в 2015 году?” Это действительно странно, пока вы не поймете, что на самом деле нам изначально нужна была не новая версия HTTP, а обновление базового *протокола управления передачей* (TCP).

TCP - это основной протокол, который предоставляет критически важные услуги, такие как **надежность и своевременная доставка**, другим протоколам, таким как HTTP. Это также одна из причин, по которой мы можем продолжать пользоваться Интернетом с большим количеством одновременных пользователей, потому что это разумно ограничивает использование полосы пропускания каждым пользователем в соответствии с его справедливой долей.

***Знаете ли вы?***  
  
***При использовании HTTP (ов) вы на самом деле используете несколько протоколов, помимо HTTP, одновременно. Каждый из протоколов в этом “стеке” имеет свои особенности и обязанности (см. Изображение ниже). Например, в то время как HTTP занимается URL-адресами и интерпретацией данных, Transport Layer Security (TLS) обеспечивает безопасность с помощью шифрования, TCP обеспечивает надежную передачу данных путем повторной передачи потерянных пакетов, а Internet Protocol (IP) направляет пакеты от одной конечной точки к другой через разные промежуточные устройства (промежуточные ящики).***

Это “наложение” протоколов друг на друга сделано для того, чтобы обеспечить легкое повторное использование их функций. Протоколам более высокого уровня (таким как HTTP) не нужно переопределять сложные функции (такие как шифрование), потому что протоколы более низкого уровня (такие как TLS) уже делают это за них. В качестве другого примера, большинство приложений в Интернете используют TCP внутри компании для обеспечения полной передачи всех своих данных. По этой причине TCP является одним из наиболее широко используемых и развернутых протоколов в Интернете.

*Сравнение стека протоколов HTTP / 2 и HTTP / 3 (*[*Большой предварительный просмотр*](https://archive.smashing.media/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/6ef36d1e-d91e-43e0-8732-f3e66ba9ea64/protocol-stack-h2-h3.png)*)*

TCP был краеугольным камнем Интернета на протяжении десятилетий, но в конце 2000-х годов начал показывать свой возраст. Предполагаемая замена ему, новый транспортный протокол под названием [QUIC](https://www.rfc-editor.org/rfc/rfc9000.html), настолько отличается от TCP несколькими ключевыми моментами, что запустить HTTP / 2 непосредственно поверх него было бы очень сложно. Таким образом, HTTP / 3 сам по себе является относительно небольшой адаптацией HTTP / 2, чтобы сделать его совместимым с новым протоколом QUIC, который включает в себя большинство новых функций, от которых люди в восторге.

QUIC необходим, потому что TCP, который существует с первых дней существования Интернета, **на самом деле не был создан с учетом максимальной эффективности**. Например, TCP требует “[рукопожатия](https://developer.mozilla.org/en-US/docs/Glossary/TCP_handshake)” для установки нового соединения. Это делается для того, чтобы гарантировать, что клиент и сервер существуют и что они желают и способны обмениваться данными. Однако, прежде чем что-либо еще можно будет сделать по соединению, также требуется **полное подключение к сети в оба конца**. Если клиент и сервер географически удалены друг от друга, время прохождения туда и обратно (RTT) может составлять более 100 миллисекунд, что приводит к заметным задержкам.

В качестве второго примера TCP воспринимает все передаваемые им данные как **один “файл” или поток байтов**, даже если мы фактически используем его для передачи нескольких файлов одновременно (например, при загрузке веб-страницы, состоящей из множества ресурсов). На практике это означает, что если TCP-пакеты, содержащие данные одного файла, будут потеряны, то все остальные файлы также будут отложены до тех пор, пока эти пакеты не будут восстановлены.

Это называется *блокировкой head-of-line (HoL)*. Хотя на практике эти недостатки вполне преодолимы (в противном случае мы бы не использовали TCP более 30 лет), они заметно влияют на протоколы более высокого уровня, такие как HTTP.

Со временем мы пытались развивать и модернизировать TCP, чтобы устранить некоторые из этих проблем и даже внедрить новые функции повышения производительности. Например, [TCP Fast Open](https://squeeze.isobar.com/2019/04/11/the-sad-story-of-tcp-fast-open) избавляет от накладных расходов на подтверждение связи, позволяя протоколам более высокого уровня отправлять данные с самого начала. Еще одна попытка называется [MultiPath TCP](https://www.multipath-tcp.org/). Идея заключается в том, что ваш мобильный телефон обычно имеет как Wi-Fi, так и сотовую связь (4G), так почему бы не использовать их оба одновременно для повышения пропускной способности и надежности?

Реализовать эти расширения TCP не так уж сложно. Однако на самом деле их **сложно развернуть в масштабе Интернета**. Поскольку TCP настолько популярен, почти каждое подключенное устройство имеет свою собственную реализацию протокола. Если эти реализации слишком старые, не имеют обновлений или содержат ошибки, то расширения практически не будут использоваться. Иными словами, все реализации должны знать о расширении, чтобы оно было полезным.

Это не было бы большой проблемой, если бы мы говорили только об устройствах конечного пользователя (таких как ваш компьютер или веб-сервер), потому что их можно относительно легко обновить вручную. Однако между клиентом и сервером расположено множество других устройств, которые также имеют свой собственный TCP-код на борту (примеры включают брандмауэры, балансировщики нагрузки, маршрутизаторы, серверы кэширования, прокси и т.д.).

Эти **промежуточные блоки** часто сложнее обновлять, а иногда и более строгие в том, что они принимают. Например, если устройство является брандмауэром, оно может быть настроено на блокировку всего трафика, содержащего (неизвестные) расширения. На практике оказывается, что огромное количество активных промежуточных блоков делают определенные предположения о TCP, которые больше не выполняются для новых расширений.

Следовательно, могут пройти **годы или даже более десятилетия**, прежде чем будет обновлено достаточно (промежуточный блок) Реализации TCP для фактического использования расширений в больших масштабах. Можно сказать, что развить TCP стало *практически* невозможно.

В результате стало ясно, что для решения этих проблем нам потребуется протокол замены TCP, а не прямое обновление. Однако из-за огромной сложности функций TCP и их различных реализаций создание чего-то нового, но лучшего с нуля было бы грандиозным мероприятием. Поэтому в начале 2010-х годов было решено отложить эту работу.

В конце концов, были проблемы не только с TCP, [но и с HTTP / 1.1](https://hpbn.co/http2/#design-and-technical-goals). Мы решили разделить работу и сначала “исправить” HTTP / 1.1, что привело к тому, что теперь является HTTP / 2. Когда это будет сделано, можно будет начать работу по замене TCP, которым теперь является QUIC. Изначально мы надеялись, что сможем запускать HTTP / 2 непосредственно поверх QUIC, но на практике это сделало бы реализацию слишком неэффективной (в основном из-за дублирования функций).

Вместо этого **HTTP / 2 был скорректирован в нескольких ключевых областях**, чтобы сделать его совместимым с QUIC. Эта измененная версия в конечном итоге была названа HTTP / 3 (вместо HTTP / 2-over-QUIC), главным образом по маркетинговым соображениям и для наглядности. Таким образом, различия между HTTP/ 1.1 и HTTP/2 гораздо существеннее, чем между HTTP/2 и HTTP/3.

Ключевой вывод здесь заключается в том, что на самом деле нам нужен был не HTTP / 3, а **“TCP / 2”**, и в процессе мы получили HTTP / 3 "бесплатно". Основные возможности, которыми мы восхищаемся в HTTP / 3 (более быстрая настройка соединения, меньше блокировок доступа, миграция соединений и так далее), На самом деле исходят от QUIC.

## Что Такое QUIC?

Возможно, вам интересно, почему это важно? Какая разница, есть ли эти функции в HTTP / 3 или QUIC? Я чувствую, что это важно, потому что **QUIC - это универсальный транспортный протокол**, который, как и TCP, может и будет использоваться для многих вариантов использования в дополнение к HTTP и загрузке веб-страниц. Например, DNS, SSH, SMB, RTP и так далее могут работать через QUIC. Поэтому давайте рассмотрим QUIC немного подробнее, потому что именно отсюда проистекает большинство неправильных представлений о HTTP / 3, которые я прочитал.

Возможно, вы слышали о том, что QUIC работает поверх еще одного протокола, называемого *User Datagram Protocol* (UDP). Это правда, но не по причинам (производительности), как утверждают многие люди. В идеале QUIC был бы полностью независимым новым транспортным протоколом, работающим непосредственно поверх IP в стеке протоколов, показанном на изображении, которым я поделился выше.

Однако это привело бы к той же проблеме, с которой мы столкнулись при попытке разработать TCP: все устройства в Интернете сначала пришлось бы обновить, чтобы распознавать и разрешать QUIC. К счастью, мы можем построить QUIC поверх еще одного широко поддерживаемого протокола транспортного уровня в Интернете: UDP.

***Знаете ли вы?***  
  
***UDP - это самый простой транспортный протокол из всех возможных. На самом деле он не предоставляет никаких функций, кроме так называемых номеров портов (например, HTTP использует порт 80, HTTPS находится на 443, а DNS использует порт 53). Он не устанавливает соединение с помощью квитирования и не является надежным: если UDP-пакет потерян, он не передается автоматически повторно. Таким образом, подход UDP “с максимальными усилиями” означает, что он максимально эффективен: нет необходимости ждать подтверждения связи и нет блокировки доступа. На практике UDP в основном используется для оперативного трафика, который обновляется с высокой скоростью и, следовательно, мало страдает от потери пакетов, поскольку отсутствующие данные в любом случае быстро устаревают (примеры включают видеоконференции в реальном времени и игры). Это также полезно в случаях, когда требуется небольшая предварительная задержка; например, поиск доменных имен в DNS действительно должен занимать всего один цикл туда и обратно.***

Многие источники утверждают, что HTTP / 3 построен поверх UDP из-за высокой производительности. Говорят, что HTTP / 3 быстрее, потому что, как и UDP, он не устанавливает соединение и не ожидает повторной передачи пакетов. **Эти утверждения неверны.** Как мы уже говорили выше, UDP используется QUIC и, следовательно, HTTP / 3 главным образом потому, что мы надеемся, что это упростит их развертывание, поскольку оно уже известно и реализовано (почти) на всех устройствах в Интернете.

Таким образом, **QUIC, по сути, переопределяет почти все функции, которые делают TCP таким мощным и популярным (но несколько более медленным) протоколом**. QUIC абсолютно надежен, он использует [подтверждения о принятых пакетах](https://www.rfc-editor.org/rfc/rfc9000.html#name-generating-acknowledgments) и [повторные передачи](https://www.rfc-editor.org/rfc/rfc9000.html#name-retransmission-of-informati), чтобы убедиться, что потерянные пакеты все еще поступают. QUIC также по-прежнему устанавливает соединение и имеет [очень сложное рукопожатие](https://www.rfc-editor.org/rfc/rfc9000.html#name-cryptographic-and-transport).

Наконец, QUIC также использует так называемые механизмы [управления потоками](https://www.rfc-editor.org/rfc/rfc9000.html#name-flow-control) и [контроля перегрузки](https://www.rfc-editor.org/rfc/rfc9002.html), которые предотвращают перегрузку сети отправителем или получателем, но также делают TCP медленнее, чем то, что вы могли бы сделать с помощью raw UDP. Главное, что QUIC реализует эти функции более разумным и производительным способом, чем TCP. Он сочетает в себе многолетний опыт развертывания и лучшие практики TCP с некоторыми основными новыми функциями. Мы обсудим эти функции более подробно позже в этой статье.

Ключевой вывод здесь заключается в том, что бесплатного обеда не существует. HTTP / 3 волшебным образом не быстрее HTTP / 2 только потому, что мы заменили TCP на UDP. Вместо этого мы переосмыслили и внедрили гораздо более продвинутую версию TCP и назвали ее QUIC. И поскольку мы хотим упростить развертывание QUIC, мы запускаем его через UDP.

## Большие Перемены

Итак, как именно QUIC улучшает TCP? В чем же разница? В QUIC появилось несколько новых конкретных функций и возможностей (данные 0-RTT, миграция соединений, повышенная устойчивость к потере пакетов и медленные сети), которые мы подробно обсудим в следующей части серии. Однако все эти нововведения в основном сводятся к четырем основным изменениям:

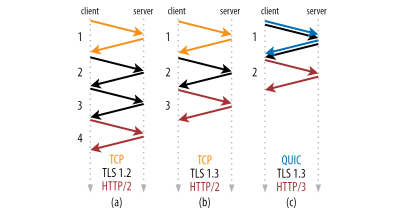
* QUIC глубоко интегрирован с TLS.
* QUIC поддерживает несколько *независимых* потоков байтов.
* QUIC использует идентификаторы соединений.
* QUIC использует фреймы.

Давайте подробнее рассмотрим каждый из этих пунктов.

### БЕЗ TLS НЕТ QUIC

Как уже упоминалось, TLS ([протокол безопасности транспортного уровня](https://www.cloudflare.com/en-gb/learning/ssl/transport-layer-security-tls/)) отвечает за защиту и шифрование данных, передаваемых через Интернет. Когда вы используете HTTPS, ваши текстовые HTTP-данные сначала шифруются с помощью TLS, а затем передаются по TCP.

***Знаете ли вы?***  
  
[***Технические детали***](https://hpbn.co/transport-layer-security-tls/) *TLS, к счастью, здесь на самом деле не нужны; вам просто нужно знать, что шифрование выполняется с использованием довольно сложной математики и очень больших (простых) чисел. Эти математические параметры согласовываются между клиентом и сервером во время отдельного криптографического обмена данными, специфичного для TLS. Так же, как и при обмене данными TCP, это согласование может занять некоторое время. В более старых версиях TLS (скажем, версии 1.2 и ниже) для этого обычно требуется два сетевых обхода в оба конца. К счастью, более новые версии TLS (последняя версия 1.3) сокращают этот процесс до одного обхода в оба конца. Это происходит главным образом потому, что TLS 1.3 серьезно ограничивает различные математические алгоритмы, которые могут быть согласованы, всего несколькими (наиболее безопасными). Это означает, что клиент может просто сразу угадать, какие из них будет поддерживать сервер, вместо того, чтобы ждать явного списка, экономя время в пути туда и обратно.*

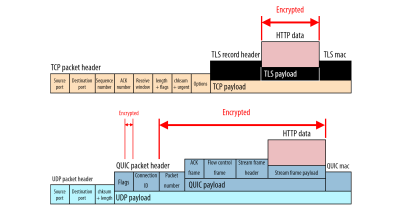
*Длительность рукопожатия TLS, TCP и QUIC (*[*большой предварительный просмотр*](https://archive.smashing.media/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/f2240cb4-eb62-4054-ad19-0e72190e0a4f/connection-setup.png)*)*

На заре Интернета шифрование трафика было довольно дорогостоящим с точки зрения обработки. Кроме того, оно также не считалось необходимым для всех вариантов использования. Исторически сложилось так, что TLS был полностью отдельным протоколом, который можно *необязательно* использовать поверх TCP. Вот почему мы проводим различие между HTTP (без TLS) и HTTPS (с TLS).

Со временем наше отношение к безопасности в Интернете, конечно, изменилось на “[безопасное по умолчанию](https://blog.chromium.org/2021/03/a-safer-default-for-navigation-https.html)”. Таким образом, хотя HTTP / 2 теоретически может запускаться непосредственно по TCP без TLS (и это даже определено в спецификации RFC как [открытый текст HTTP / 2](https://tools.ietf.org/html/rfc7540#section-3.1)), ни один (популярный) веб-браузер фактически не поддерживает этот режим. В некотором смысле производители браузеров сознательно пошли на компромисс в целях повышения безопасности за счет снижения производительности.

Учитывая очевидную эволюцию в сторону постоянного подключения к TLS (особенно для веб-трафика), неудивительно, что разработчики QUIC решили вывести эту тенденцию на новый уровень. Вместо того, чтобы просто не определять режим открытого текста для HTTP / 3, они решили глубоко внедрить шифрование в сам QUIC. В то время как первые версии QUIC для Google использовали для этого специальную настройку, стандартизированный QUIC напрямую использует существующий протокол TLS 1.3.

Для этого он как бы нарушает типичное чистое **разделение между протоколами в стеке протоколов**, как мы можем видеть на предыдущем изображении. Хотя TLS 1.3 по-прежнему может выполняться независимо поверх TCP, QUIC вместо этого как бы инкапсулирует TLS 1.3. Иными словами, нет способа использовать QUIC без TLS; QUIC (и, соответственно, HTTP / 3) всегда полностью зашифрован. Более того, QUIC также шифрует почти все поля заголовка пакета; информация транспортного уровня (например, номера пакетов, которые никогда не шифруются для TCP) больше не читается посредниками в QUIC (даже некоторые флаги заголовка пакета зашифрованы).

*В отличие от TCP + TLS, QUIC также шифрует метаданные транспортного уровня в заголовке пакета и полезной нагрузке. (Примечание: размеры полей не масштабируются.) (*[*Большой предварительный просмотр*](https://archive.smashing.media/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/fbf86b42-8f20-4b27-aea5-f1fc164b2683/tcp-vs-quic-packetization.png)*)*

Для всего этого QUIC сначала использует рукопожатие TLS 1.3, более или менее аналогичное TCP, для установления математических параметров шифрования. После этого, однако, QUIC берет управление на себя и шифрует пакеты самостоятельно, тогда как при использовании TLS поверх TCP TLS выполняет собственное шифрование. Это, казалось бы, небольшое отличие представляет собой фундаментальное концептуальное изменение в сторону постоянного шифрования, которое применяется на все более низких уровнях протокола.

Такой подход дает QUIC ряд преимуществ:

* **QUIC стал более безопасным для своих пользователей.**  
  **Нет способа запустить cleartext QUIC, поэтому у злоумышленников и подслушивающих устройств также меньше возможностей для прослушивания.** (Недавние исследования показали, [насколько опасной может быть опция открытого текста HTTP / 2](https://labs.bishopfox.com/tech-blog/h2c-smuggling-request-smuggling-via-http/2-cleartext-h2c).)
* **Настройка соединения в QUIC происходит быстрее.**  
  **В то время как для TLS-over-TCP обоим протоколам требуются свои собственные рукопожатия, QUIC вместо этого объединяет транспортное и криптографическое рукопожатие в одно, экономя время в оба конца (см. Изображение выше).** Мы обсудим это более подробно в [части 2](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/).
* **QUIC может развиваться легче.**  
  **Поскольку он полностью зашифрован, промежуточные блоки в сети больше не могут наблюдать и интерпретировать его внутреннюю работу, как они это делают с TCP.** Следовательно, они также больше не могут прерываться (случайно) в более новых версиях QUIC, поскольку их не удалось обновить. Если мы хотим добавить новые функции в QUIC в будущем, нам “всего лишь” нужно обновить конечные устройства, а не все промежуточные модули.

Однако, помимо этих преимуществ, у расширенного шифрования есть и некоторые потенциальные недостатки:

* **Многие сети не решатся разрешить QUIC.**  
  **Компании могут захотеть заблокировать это в своих брандмауэрах, поскольку обнаружение нежелательного трафика становится более сложным.** Интернет-провайдеры и промежуточные сети могут заблокировать его, поскольку такие показатели, как средние задержки и процент потери пакетов, больше недоступны, что затрудняет обнаружение и диагностику проблем. Все это означает, что QUIC, вероятно, никогда не станет общедоступным, о чем мы подробнее поговорим в [части 3](https://www.smashingmagazine.com/2021/09/http3-practical-deployment-options-part3/).
* **У QUIC более высокие затраты на шифрование.**  
  **QUIC шифрует каждый отдельный пакет с помощью TLS, тогда как TLS-over-TCP может шифровать несколько пакетов одновременно.** Это потенциально замедляет работу QUIC в сценариях с высокой пропускной способностью (как мы увидим в [части 2](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/)).
* **QUIC делает Интернет более централизованным.**  
  **Жалоба, с которой я часто сталкивался, звучит примерно так: “Google продвигает QUIC, потому что он предоставляет им полный доступ к данным, не делясь ими с другими”.** Я в основном с этим не согласен. Во-первых, QUIC скрывает от внешних наблюдателей не больше (или меньше!) информации на уровне пользователя (например, какие URL-адреса вы посещаете), чем TLS-over-TCP (QUIC сохраняет статус-кво).

Во-вторых, в то время как Google инициировала проект QUIC, окончательные протоколы, о которых мы говорим сегодня, были разработаны гораздо более широкой командой в составе Internet Engineering Task Force (IETF). QUIC от IETF технически сильно отличается от QUIC от Google. Тем не менее, это правда, что люди в IETF в основном из крупных компаний, таких как Google и Facebook, и CDN, таких как Cloudflare и Fastly. Из-за сложности QUIC это будут в основном те компании, которые обладают необходимыми ноу-хау для правильного и производительного внедрения, например, HTTP / 3 на практике. Вероятно, это приведет к большей централизации в этих компаниях, что *является* реальной проблемой.

Ключевым выводом здесь является то, что QUIC **по умолчанию глубоко зашифрован**. Это не только улучшает его характеристики безопасности и конфиденциальности, но также способствует его развертыванию и эволюции. Это немного усложняет запуск протокола, но, в свою очередь, позволяет другие оптимизации, такие как более быстрое установление соединения.

### QUIC ЗНАЕТ О МНОГОБАЙТОВЫХ ПОТОКАХ

### Второе большое различие между TCP и QUIC немного более техническое, и мы более подробно рассмотрим его последствия в [части 2](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/). Однако на данный момент мы можем понять основные аспекты на высоком уровне.

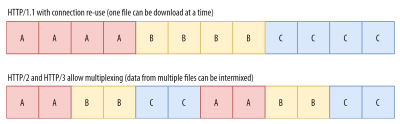
***Знаете ли вы?***  
  
***Сначала подумайте о том, что даже простая веб-страница состоит из множества независимых файлов и ресурсов. Существуют HTML, CSS, JavaScript, изображения и так далее. Каждый из этих файлов можно рассматривать как простой “двоичный двоичный объект” — набор нулей и единиц, которые определенным образом интерпретируются браузером. При отправке этих файлов по сети мы не передаем их все сразу. Вместо этого они подразделяются на более мелкие фрагменты (обычно примерно по 1400 байт каждый) и отправляются отдельными пакетами. Таким образом, мы можем рассматривать каждый ресурс как отдельный “поток байтов”, поскольку данные загружаются или “передаются” по частям с течением времени.***

Для HTTP/ 1.1 процесс загрузки ресурсов довольно прост, потому что **каждому файлу присваивается собственное TCP-соединение** и он загружается полностью. Например, если у нас есть файлы A, B и C, у нас будет три TCP-соединения. Первый будет видеть поток байтов AAAA, второй BBBB, третий CCCC (при этом каждое повторение буквы является TCP-пакетом). Это работает, но также очень неэффективно, потому что каждое новое соединение сопряжено с некоторыми накладными расходами.

На практике **браузеры накладывают ограничения** на количество одновременно используемых подключений (и, следовательно, на количество файлов, которые могут быть загружены параллельно) — обычно от 6 до 30 на загрузку страницы. Затем соединения используются повторно для загрузки нового файла после полной передачи предыдущего. Эти ограничения в конечном итоге начали снижать производительность Интернета на современных страницах, которые часто загружают намного больше 30 ресурсов.

Улучшение этой ситуации было одной из главных целей HTTP / 2. Протокол делает это, больше не открывая новое TCP-соединение для каждого файла, а вместо этого загружая различные ресурсы по одному TCP-соединению. Это достигается за счет **“мультиплексирования” различных потоков байтов**. Это причудливый способ сказать, что мы смешиваем данные из разных файлов при их транспортировке. Для наших трех файлов-примеров мы получим одно TCP-соединение, и входящие данные могут выглядеть как AABBCCAABBCCCC (хотя [возможны многие другие схемы упорядочения](https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web/)). Это кажется достаточно простым и действительно работает довольно хорошо, делая HTTP / 2 обычно таким же быстрым или немного быстрее, чем HTTP / 1.1, но с гораздо меньшими накладными расходами.

Давайте подробнее рассмотрим разницу:

*HTTP / 1.1 не допускает мультиплексирования, в отличие от HTTP / 2 и HTTP / 3. (*[*Большой предварительный просмотр*](https://archive.smashing.media/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/900ea8f0-3782-4505-b1b6-99ca2954bbce/multiplexing-basic.png)*)*

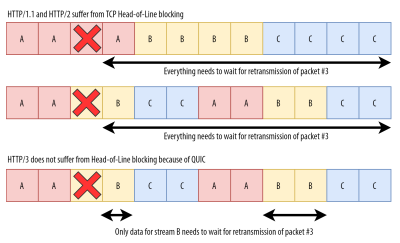
Однако на стороне TCP существует проблема. Видите ли, поскольку TCP - гораздо более старый протокол, созданный не только для загрузки веб-страниц, он не знает об A, B или C. Внутри **TCP думает, что он передает только один файл**, X, и его не волнует, что то, что он рассматривает как XXXXXXXXXXXXXX, на самом деле является AABBCCAABBCC на уровне HTTP. В большинстве ситуаций это не имеет значения (и на самом деле это делает TCP довольно гибким!), Но это меняется, когда происходит, например, потеря пакетов в сети.

Предположим, что третий TCP-пакет потерян (тот, который содержит первые данные для файла B), но все остальные данные доставлены. Протокол TCP устраняет эту потерю, **повторно передавая новую копию потерянных данных** в новом пакете. Однако для повторной передачи может потребоваться некоторое время (по крайней мере, один RTT). Вы можете подумать, что это не такая уж большая проблема, поскольку мы видим, что потери ресурсов A и C. Таким образом, мы можем начать их обработку, ожидая недостающих данных для B, верно?

К сожалению, это не так, потому что логика повторной передачи выполняется на уровне TCP, а TCP не знает об A, B и C! TCP вместо этого думает, что часть одного X-файла была потеряна, и поэтому считает, что должен препятствовать обработке остальных данных X до тех пор, пока дыра не будет заполнена. Иными словами, хотя на уровне HTTP / 2 мы знаем, что уже можем обрабатывать A и C, TCP этого не знает, в результате чего работа выполняется **медленнее, чем потенциально могла бы быть**. Эта неэффективность является примером [проблемы с блокировкой “head-of-line (HoL)”](https://calendar.perfplanet.com/2020/head-of-line-blocking-in-quic-and-http-3-the-details/).

**Решение проблемы блокировки HoL на транспортном уровне было одной из главных целей QUIC**. В отличие от TCP, QUIC прекрасно понимает, что он мультиплексирует несколько *независимых* потоков байтов. Он, конечно, не знает, что передает CSS, JavaScript и изображения; он просто знает, что потоки разделены. Таким образом, QUIC может выполнять логику обнаружения потери пакетов и восстановления для каждого потока.

В приведенном выше сценарии он будет удерживать данные только для потока B, и, в отличие от TCP, он будет доставлять любые данные для A и C на уровень HTTP / 3 как можно скорее. (Это проиллюстрировано ниже.) Теоретически это может привести к повышению производительности. Однако на практике история гораздо более тонкая, о чем мы поговорим в [части 2](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/).

*QUIC позволяет HTTP / 3 обойти проблему с блокировкой в начале строки. (*[*Большой предварительный просмотр*](https://archive.smashing.media/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/7981cb82-395c-4484-8873-46fd92804b4d/hol-blocking-basic.png)*)*

Мы видим, что теперь у нас есть фундаментальное различие между TCP и QUIC. Это, кстати, также одна из основных причин, почему мы не можем просто запускать HTTP / 2 как есть поверх QUIC. Как мы уже говорили, HTTP / 2 также включает в себя концепцию запуска нескольких потоков по одному (TCP) соединению. Таким образом, HTTP / 2-over-QUIC будет иметь две разные и конкурирующие потоковые абстракции друг над другом.

Заставить их работать вместе было бы очень сложно и подвержено ошибкам; итак, одно из ключевых различий между HTTP / 2 и HTTP / 3 заключается в том, что последний **удаляет логику HTTP stream и вместо этого повторно использует потоки QUIC**. Однако, как мы увидим в [части 2](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/), это имеет и другие последствия для реализации таких функций, как передача данных с сервера, сжатие заголовков и расстановка приоритетов.

Ключевым выводом здесь является то, что протокол TCP никогда не предназначался для передачи нескольких независимых файлов по одному соединению. Поскольку это именно то, что требуется для просмотра веб-страниц, это привело ко многим неэффективностям на протяжении многих лет. QUIC решает эту проблему, делая потоки с несколькими байтами основной концепцией на транспортном уровне и обрабатывая потери пакетов для каждого потока.

### QUIC SUPPORTS CONNECTION MIGRATION

The third major improvement in QUIC is the fact that connections can stay alive longer.

***Did You Know?***  
  
***We often use the concept of a “connection” when talking about web protocols. However, what exactly is a connection? Typically, people speak of a TCP connection once there has been a handshake between two endpoints (say, the browser or client and the server). This is why UDP is often (somewhat misguidedly) said to be “connectionless”, because it doesn’t do such a handshake. However, the handshake is really nothing special: It’s just a few packets with a specific form being sent and received. It has a few goals, main among them being to make sure there is something on the other side and that it’s willing and able to talk to us. It’s worth repeating here that QUIC also performs a handshake, even though it runs over UDP, which by itself doesn’t.***

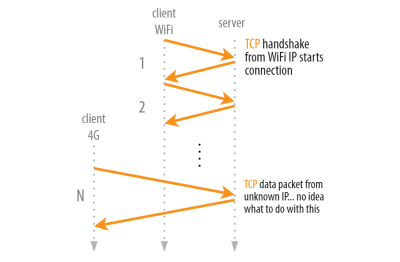
Итак, возникает вопрос, как эти пакеты попадают в нужное место назначения? В Интернете IP-адреса используются для маршрутизации пакетов между двумя уникальными компьютерами. Однако простого наличия IP-адресов для вашего телефона и сервера недостаточно, потому что оба хотят иметь возможность запускать несколько сетевых программ на каждом конце одновременно.

Вот почему каждому отдельному соединению также присваивается **номер порта** на обеих конечных точках, чтобы различать соединения и приложения, которым они принадлежат. Серверные приложения обычно имеют фиксированный номер порта в зависимости от их функции (например, порты 80 и 443 для HTTP (ов) и порт 53 для DNS), в то время как клиенты обычно выбирают номера своих портов (полу-) случайным образом для каждого соединения.

Таким образом, для определения уникального соединения между компьютерами и приложениями нам нужны эти четыре вещи, так называемый **4-кортеж: IP-адрес клиента + порт клиента + IP-адрес сервера + порт сервера**.

В TCP соединения идентифицируются только по 4-му кортежу. Таким образом, если изменяется хотя бы один из этих четырех параметров, соединение становится недействительным и его необходимо восстановить (включая новое подтверждение связи). Чтобы понять это, представьте проблему с парковкой: в данный момент вы используете свой смартфон внутри здания с Wi-Fi. Таким образом, у вас есть IP-адрес в этой сети Wi-Fi.

Если вы сейчас выйдете на улицу, ваш телефон может переключиться на сотовую сеть 4G. Поскольку это новая сеть, он получит совершенно новый IP-адрес, поскольку он зависит от конкретной сети. Теперь сервер будет видеть TCP-пакеты, поступающие с IP-адреса клиента, которого он раньше не видел (хотя два порта и IP-адрес сервера, конечно, могут остаться прежними). Это проиллюстрировано ниже.

*Проблема с TCP на парковке: как только клиент получает новый IP, сервер больше не может связать его с подключением. (*[*Большой предварительный просмотр*](https://archive.smashing.media/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/9413b221-47e9-427b-b958-b0e62fe7f681/1-migration-tcp.png)*)*

Но как сервер может узнать, что эти пакеты с нового IP принадлежат "соединению”? Откуда известно, что эти пакеты не принадлежат *новому* соединению от другого клиента в сотовой сети, который выбрал тот же (случайный) клиентский порт (что легко может случиться)? К сожалению, он не может этого знать.

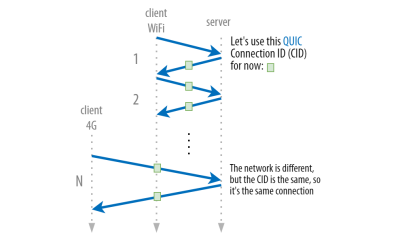
Поскольку TCP был изобретен еще до того, как мы начали мечтать о сотовых сетях и смартфонах, не существует, например, механизма, позволяющего клиенту сообщать серверу, что он сменил IP-адрес. Нет даже способа “закрыть” соединение, потому что команда TCP reset или fin, отправленная старому 4-кортежу, больше даже не дошла бы до клиента. Таким образом, на практике каждое изменение сети означает, что **существующие TCP-соединения больше нельзя использовать**.

Для установки нового соединения необходимо выполнить новое подтверждение связи TCP (и, возможно, TLS), и, в зависимости от протокола уровня приложения, потребуется перезапустить действия в процессе. Например, если вы загружали большой файл по протоколу HTTP, то этот файл, возможно, придется повторно запрашивать с самого начала (например, если сервер не поддерживает [запросы диапазона](https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests)). Другой пример - видеоконференции в режиме реального времени, когда при переключении сетей может произойти кратковременное отключение питания.

Обратите внимание, что существуют другие причины, по которым 4-кортеж может измениться (например, [повторная привязка NAT](https://blog.cloudflare.com/the-road-to-quic/#onenattobringthemallandinthedarknessbindthem)), которые мы подробнее обсудим в [части 2](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/).

Таким образом, перезапуск TCP-соединений может иметь серьезные последствия (ожидание новых рукопожатий, перезапуск загрузок, восстановление контекста). Чтобы решить эту проблему, QUIC вводит новую концепцию под названием **идентификатор соединения (CID)**. Каждому соединению присваивается еще один номер поверх 4-го кортежа, который однозначно идентифицирует его между двумя конечными точками.

Crucially, because this CID is defined at the transport layer in QUIC itself, it doesn’t change when moving between networks! This is shown in the image below. To make this possible, the CID is included at the front of each and every QUIC packet (much like how the IP addresses and ports are also present in each packet). (It’s actually one of the few things in the QUIC packet header that aren’t encrypted!)

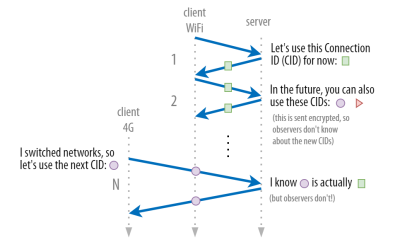
*QUIC использует идентификаторы соединений (CID), позволяющие соединениям выживать при смене сети. (*[*Большой предварительный просмотр*](https://archive.smashing.media/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/e6ae0ec1-3b85-49a9-9707-ee21ce5b02b3/2-migration-single-cid.png)*)*

При такой настройке, даже когда что-то меняется в 4-х кортежах, серверу и клиенту QUIC **нужно только взглянуть на CID**, чтобы понять, что это то же самое старое соединение, и затем они могут продолжать его использовать. Нет необходимости в новом рукопожатии, и состояние загрузки может быть сохранено без изменений. Обычно эта функция называется *переносом соединения*. Теоретически это лучше для производительности, но, как мы обсудим в [части 2](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/), это, конечно, опять же история с нюансами.

С CID приходится преодолевать и другие проблемы. Например, если бы мы действительно использовали только один CID, хакерам и перехватчикам было бы чрезвычайно легко следить за пользователем по сетям и, соответственно, определять его (приблизительное) физическое местоположение. Чтобы предотвратить этот кошмар конфиденциальности, **QUIC меняет CID каждый раз, когда используется новая сеть**.

Это может вас смутить: разве я только что не сказал, что CID должен быть одинаковым во всех сетях? Ну, это было чрезмерное упрощение. Что действительно происходит внутри, так это то, что клиент и сервер согласовывают **общий список (случайно сгенерированных) идентификаторов CID**, которые все сопоставляются с одним и тем же концептуальным "соединением”.

Например, они оба знают, что идентификаторы CID K, C и D на самом деле все соответствуют соединению X. Таким образом, хотя клиент может помечать пакеты с помощью K в Wi-Fi, он может переключиться на использование C в 4G. Эти общие списки согласовываются полностью зашифрованными в QUIC, поэтому потенциальные злоумышленники не будут знать, что K и C на самом деле являются X, но клиент и сервер будут знать это и смогут поддерживать соединение в рабочем состоянии.

*QUIC использует несколько согласованных идентификаторов подключения (CID) для предотвращения отслеживания пользователей. (*[*Большой предварительный просмотр*](https://archive.smashing.media/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/715f189e-4ae6-4c4c-8db8-9fd8170049d8/3-migration-multi-cid.png)*)*

Это становится еще сложнее, потому что клиенты и серверы будут иметь разные списки CID, которые они выбирают сами (подобно тому, как у них разные номера портов). В основном это делается для поддержки маршрутизации и балансировки нагрузки при крупномасштабных настройках сервера, как мы увидим более подробно в [части 3](https://www.smashingmagazine.com/2021/09/http3-practical-deployment-options-part3/).

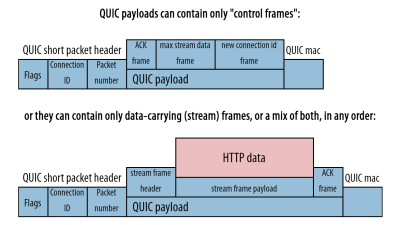
Ключевым выводом здесь является то, что в TCP соединения определяются **четырьмя параметрами**, которые могут меняться, когда конечные точки меняют сеть. Таким образом, эти соединения иногда приходится перезапускать, что приводит к некоторому простою. QUIC добавляет в набор еще один параметр, называемый идентификатором соединения. И клиент, и сервер QUIC знают, какие идентификаторы соединений каким соединениям соответствуют, и, таким образом, более устойчивы к изменениям в сети.

### QUIC ГИБОК И РАЗВИВАЕТСЯ

Последний аспект QUIC заключается в том, что он специально создан для того, чтобы его было легко развивать. Это достигается несколькими различными способами. Во-первых, как уже обсуждалось, тот факт, что QUIC почти полностью зашифрован, означает, что нам нужно обновить только конечные точки (клиенты и серверы), а не все промежуточные блоки, если мы хотим развернуть более новую версию QUIC. Это все еще требует времени, но обычно порядка месяцев, а не лет.

Во-вторых, в отличие от TCP, QUIC не использует один фиксированный заголовок пакета для отправки всех метаданных протокола. Вместо этого QUIC имеет короткие заголовки пакетов и использует [различные “фреймы”](https://www.rfc-editor.org/rfc/rfc9000.html#name-frames-and-frame-types) (что-то вроде миниатюрных специализированных пакетов) внутри полезной нагрузки пакета для передачи дополнительной информации. Существует, например, ACK фрейм (для подтверждения), NEW\_CONNECTION\_ID фрейм (для помощи в настройке переноса соединения) и STREAM фрейм (для передачи данных), как показано на рисунке ниже.

В основном это делается в целях оптимизации, потому что не каждый пакет содержит все возможные метаданные (и поэтому заголовок TCP-пакета обычно тратит довольно много байтов — см. Также Изображение выше). Однако очень полезным побочным эффектом использования фреймов является то, что в будущем будет очень легко определять новые типы фреймов как расширения QUIC. Например, очень важным является [DATAGRAM фрейм](https://datatracker.ietf.org/doc/html/draft-ietf-quic-datagram-02), который позволяет отправлять ненадежные данные по зашифрованному соединению QUIC.

*QUIC uses individual frames to send meta data, instead of a large fixed packet header. (*[*Large preview*](https://archive.smashing.media/assets/344dbf88-fdf9-42bb-adb4-46f01eedd629/88c76a7a-2752-4e5b-a829-290cd4951af3/quic-framing.png)*)*

Thirdly, QUIC uses a custom TLS extension to carry what are called [transport parameters](https://www.rfc-editor.org/rfc/rfc9000.html#name-transport-parameters). These allow the client and server to choose a configuration for a QUIC connection. This means they can negotiate which features are enabled (for example, whether to allow connection migration, which extensions are supported, etc.) and communicate sensible defaults for some mechanisms (for example, maximum supported packet size, flow control limits). While the QUIC standard defines a [long list](https://www.rfc-editor.org/rfc/rfc9000.html#name-transport-parameter-definit) of these, it also allows extensions to define new ones, again making the protocol more flexible.

Наконец, хотя QUIC сам по себе не является реальным требованием, большинство реализаций в настоящее время выполняются в “пользовательском пространстве" (в отличие от TCP, который обычно выполняется в “пространстве ядра”). Подробности обсуждаются в [части 2](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/), но в основном это означает, что экспериментировать с вариантами реализации QUIC и расширениями намного проще, чем с TCP.

Хотя QUIC теперь стандартизирован, на самом деле его следует рассматривать как **QUIC версии 1** (что также четко указано в [Запросе комментариев (RFC)](https://www.rfc-editor.org/rfc/rfc9000.html#name-overview)), и есть четкое намерение создать версию 2 и более довольно быстро. Кроме того, QUIC позволяет легко определять расширения, что позволяет реализовать еще больше вариантов использования.

## Заключение

Давайте подведем итог тому, что мы узнали в этой части. В основном мы говорили о вездесущем протоколе TCP и о том, как он был разработан в то время, когда многие из современных проблем были неизвестны. По мере того, как мы пытались развивать TCP, чтобы идти в ногу со временем, стало ясно, что на практике это будет сложно, потому что почти у каждого устройства есть своя собственная реализация TCP, которую необходимо обновить.

Чтобы обойти эту проблему и при этом улучшить TCP, мы создали **новый протокол QUIC** (который на самом деле представляет собой TCP 2.0 под капотом). Чтобы упростить развертывание QUIC, он запускается поверх протокола UDP (который также поддерживает большинство сетевых устройств), а чтобы убедиться, что он может развиваться в будущем, по умолчанию он почти полностью зашифрован и использует гибкий механизм фрейминга.

Помимо этого, **QUIC в основном отражает известные функции TCP**, такие как квитирование, надежность и контроль перегрузки. Двумя основными изменениями, помимо шифрования и фрейминга, являются понимание многобайтовых потоков и введение идентификатора соединения. Однако этих изменений было достаточно, чтобы помешать нам напрямую запускать HTTP / 2 поверх QUIC, что потребовало создания HTTP / 3 (который на самом деле является HTTP / 2 поверх QUIC под капотом).

Новый подход QUIC приводит к ряду улучшений производительности, но их потенциальный выигрыш более детализирован, чем обычно сообщается в статьях о QUIC и HTTP / 3. Теперь, когда мы знаем основы, мы можем обсудить эти нюансы более подробно в [следующей части](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/) этой серии.

* **Часть 1: История HTTP / 3 и основные концепции**  
  **Эта статья предназначена для людей, не знакомых с HTTP / 3 и протоколами в целом, и в ней в основном обсуждаются основы.**
* [Часть 2: Особенности производительности HTTP / 3](https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/)Эта более глубокая и техническая. Люди, которые уже знают основы, могут начать здесь.
* [Часть 3: Практические варианты развертывания HTTP / 3](https://www.smashingmagazine.com/2021/09/http3-practical-deployment-options-part3/)В этой третьей статье из серии объясняются проблемы, связанные с самостоятельным развертыванием и тестированием HTTP / 3. В ней подробно рассказывается, как и следует ли вам изменять свои веб-страницы и ресурсы.