# TCP против UDP или будущее сетевых протоколов

**27 мин** **171K**

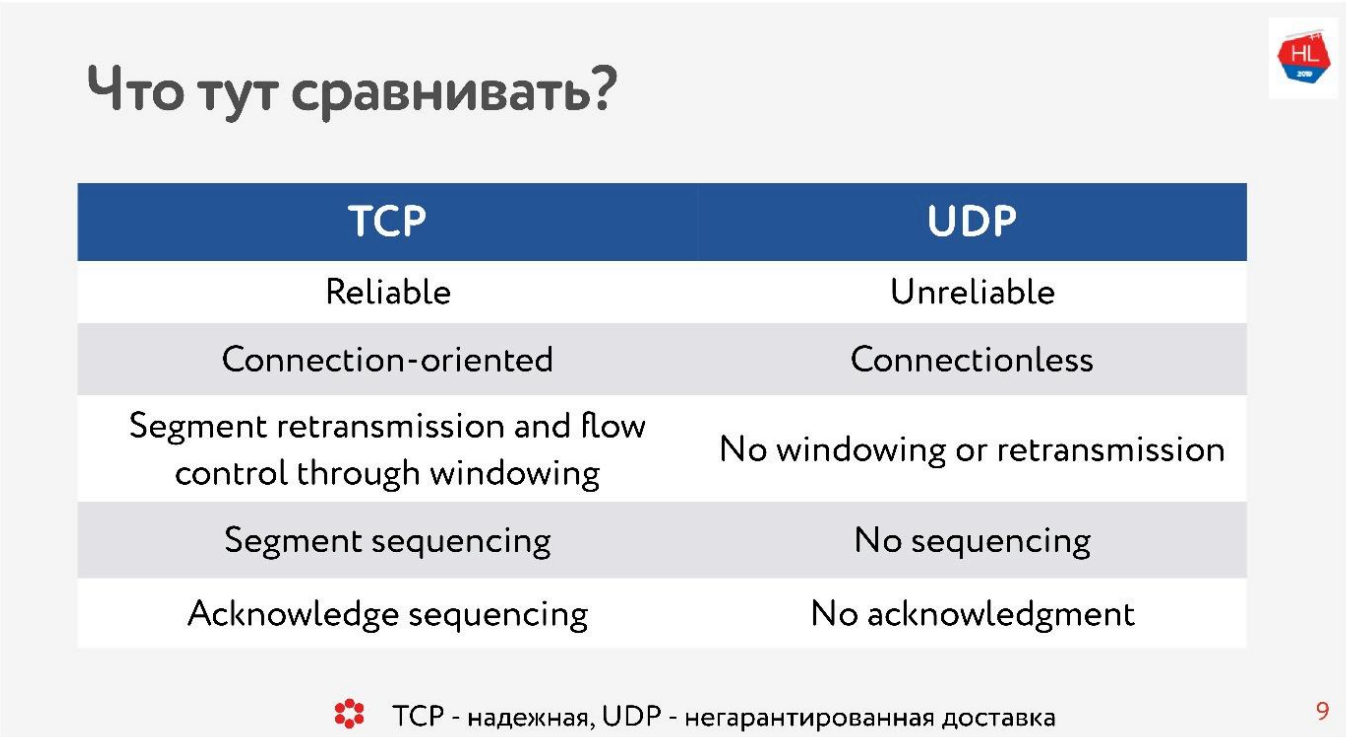
Блог компании Конференции Олега Бунина (Онтико)Высокая производительность*Программирование*
Анализ и проектирование систем*Разработка систем связи*

Перед каждым сервисом, генерирующим хотя бы 1 Мбит/сек трафика в интернете возникает вопрос: «Как? по TCP или по UDP?» В прикладных областях, в том числе и платформах доставки уже сложились предпочтения и традиции принятия подобных решений.

По идее, если бы, к примеру, однажды один ленивый разработчик не попробовал развернуть свой ML на Python (потому что только его и знал), мир скорее всего никогда не проникся бы такой любовью к презренному «супер-джава-кодерами» языку. А сегодня слабости этого языка в прошлом контексте применения безоговорочно обеспечивают ему первенство в развертывании и запуске многочисленных майнерских А/Б.

Сравнивать можно многое: ARM с Intel, iOS и Android, а Mortal Kombat с Injustice. И нарваться на космический холивар, поэтому вернемся к теме доставки огромных объемов разноформатного контента.

Десять лет назад все были абсолютно уверены, UDP — это что-то про негарантированную доставку. Если нужен надежный протокол — это TCP. И вопреки традициям в этой статье мы будем сравнивать такие, кажущиеся несравнимыми вещи, как TCP и UDP.



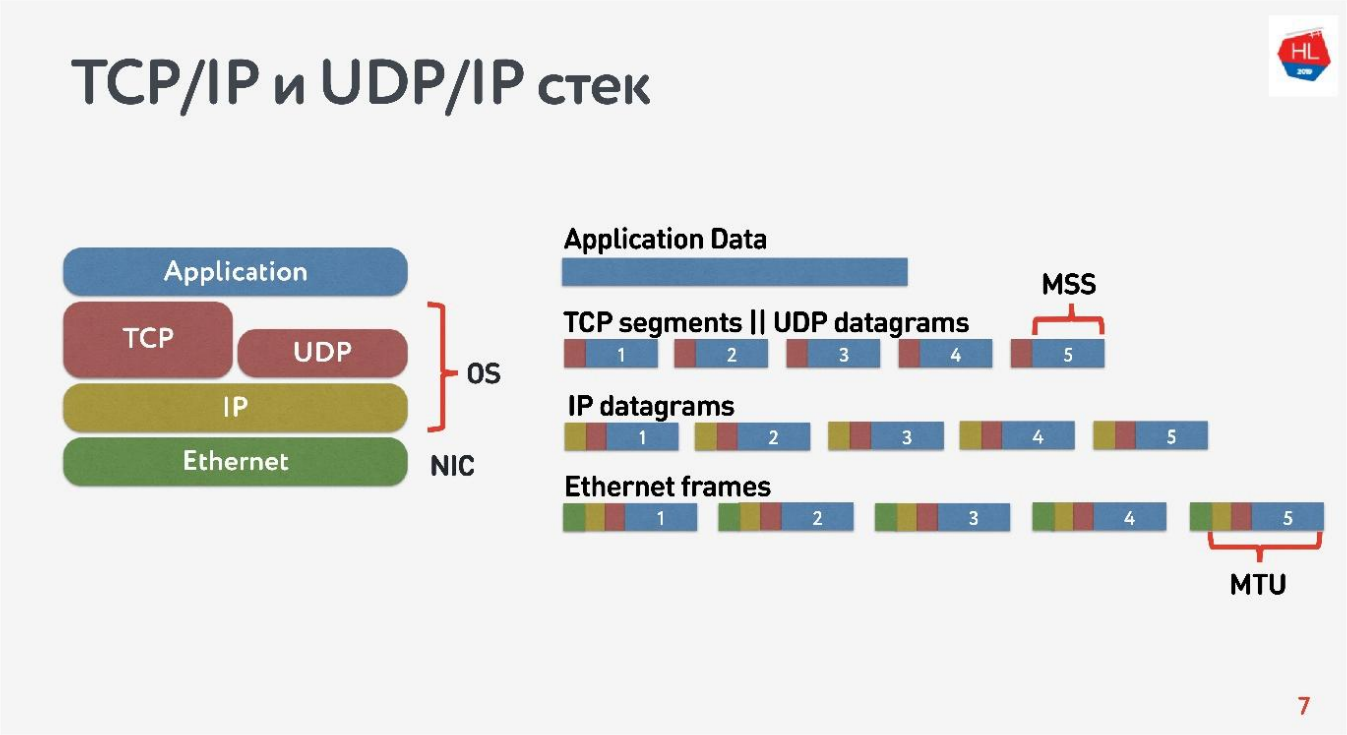

*Осторожно, под катом 99 иллюстраций и схем и все важные.*

Сравнение проводит руководитель разработки платформ Видео и Лента в ОК **Александр Тоболь** ( alatobol). Сервисы Видео и Лента Новостей в соцсети ОК — исключительно про контент и его доставку на все существующие клиентские платформы в сколько угодно плохих или отличных условиях сети, и вопрос, как его доставлять — по TCP или по UDP — имеет решающее значение.

## TCP vs UDP. Минимум теории

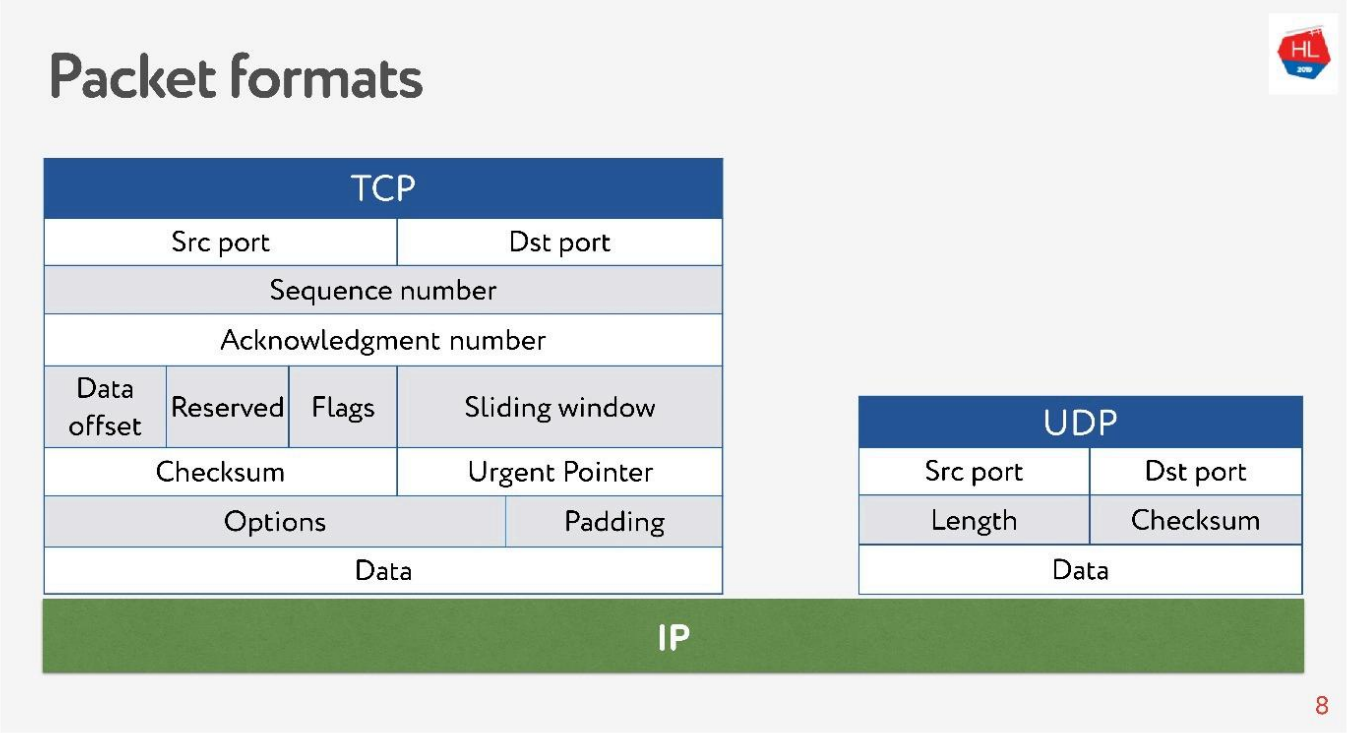
Чтобы перейти к сравнению, нам потребуется немного базовой теории.


Что мы знаем об IP сетях? Поток данных, который вы отправляете, разбивается на пакеты, какой-то черный ящик доставляет эти пакеты до клиента. Клиент собирает пакеты и получает поток данных. Обычно это все прозрачно и нет необходимости думать, что там на нижних уровнях.




На схеме представлены TCP/IP и UDP/IP стек. Внизу есть Ethernet-пакеты, IP-пакеты, и дальше на уровне ОС есть TCP и UDP. TCP и UDP в этом стеке не сильно друг от друга отличаются. Они инкапсулируются в IP-пакеты, и приложения могут ими пользоваться. Чтобы

увидеть отличия, нужно посмотреть внутрь TCP- и UDP-пакета.




И там, и там есть порты. Но **в UDP есть только контрольная сумма** — длина пакета, этот протокол максимально простой. А в TCP — очень много данных, которые явно указывают окно, acknowledgement, sequence, пакеты и так далее. Очевидно, **TCP более сложный**.

> Если говорить очень грубо, то TCP — это протокол надежной доставки, а UDP — ненадежной.

И всё же, несмотря на заявленную ненадёжность UDP, мы разберём, возможно ли доставить данные быстрее и надежнее чем с использованием TCP. Попробуем посмотреть на сеть изнутри и понять, как она работает. Попутно затронем следующие вопросы:

- зачем сравнивать TCP или что с ним не так;
- с чем и на чем надо сравнивать TCP;
- как поступил Google и какое решение принял;
- какое будущее сетевых протоколов нас ждет.

В этой статье не будет теории: уровней и моделей OSI, сложных математических моделей, хотя через них все можно посчитать. Будем по максимуму разбирать, как потрогать сеть не в теории, а своими руками.

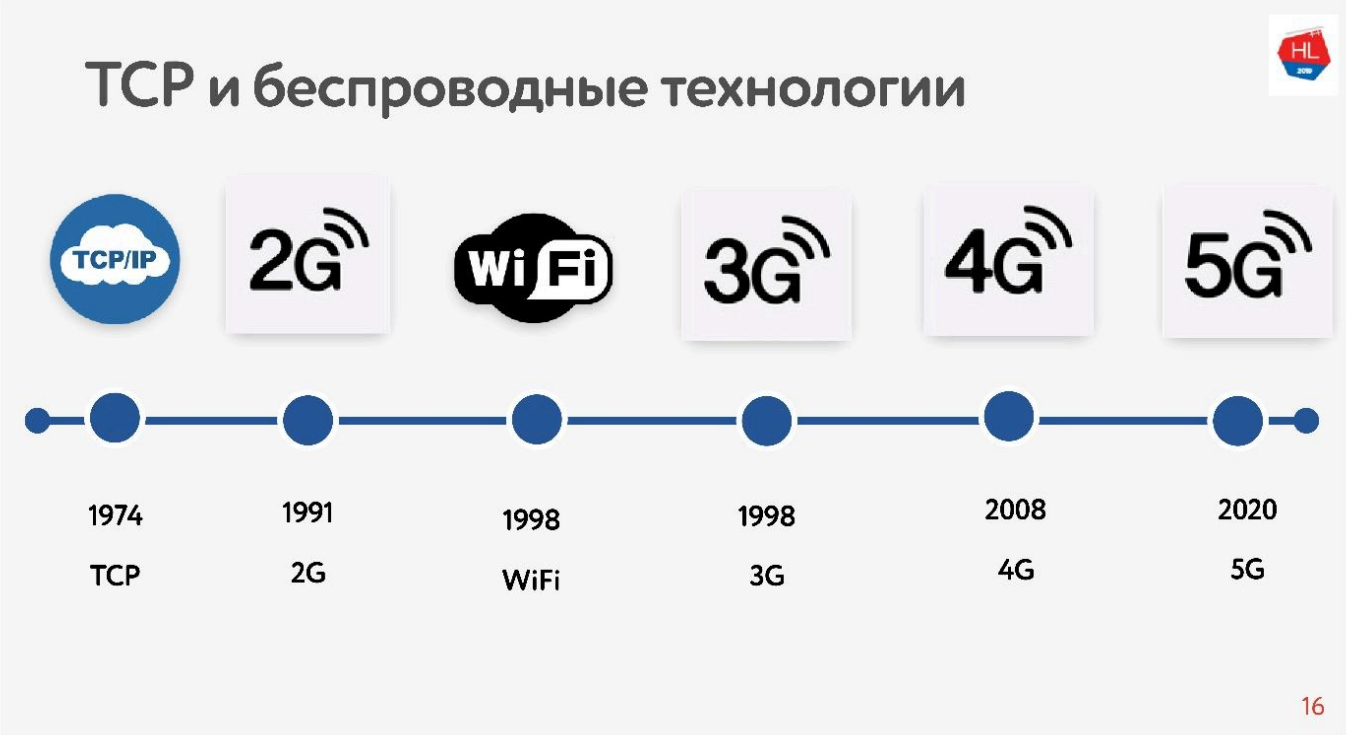
## Зачем сравнивать TCP или что с ним не так

TCP придумали в 1974 году, а лет через 20, когда я пошел в школу, я покупал интернет-карты, стирал код и куда-то звонил. Причем, если звонить с 2 ночи до 7 утра, то интернет был бесплатный, но дозвониться было трудно.

Прошло еще 20 лет, и пользователи на мобильных беспроводных сетях стали превалировать над «проводными» пользователями, при этом TCP концептуально не менялся.

> Мобильный мир победил, появились беспроводные протоколы, а TCP был по-прежнему неизменен.

Сегодня 80% пользователей используют Wi-Fi или беспроводную 3G-4G сеть.

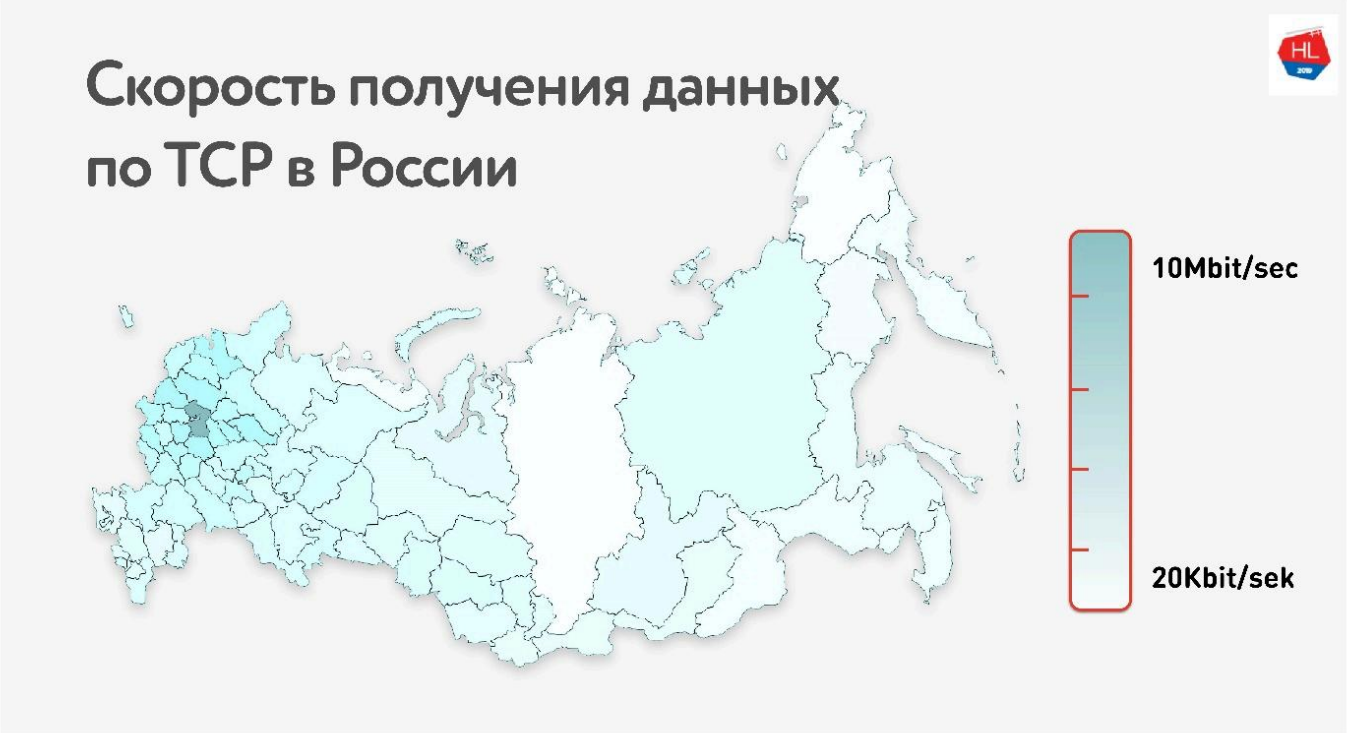



В беспроводных сетях существуют:

- packet loss — примерно 0,6% пакетов, которые мы отправляем, теряются по пути;
- reordering — перестановка пакетов местами, в реальной жизни довольно редкое явление, но случается в 0,2% случаев;
- jitter — когда пакеты отправляются равномерно, а приходят очередями с задержкой примерно в 50 мс.

Все эти особенности передачи данных в гетерогенных сетях TCP успешно скрывает от вас, и вам не нужно погружаться внутрь.

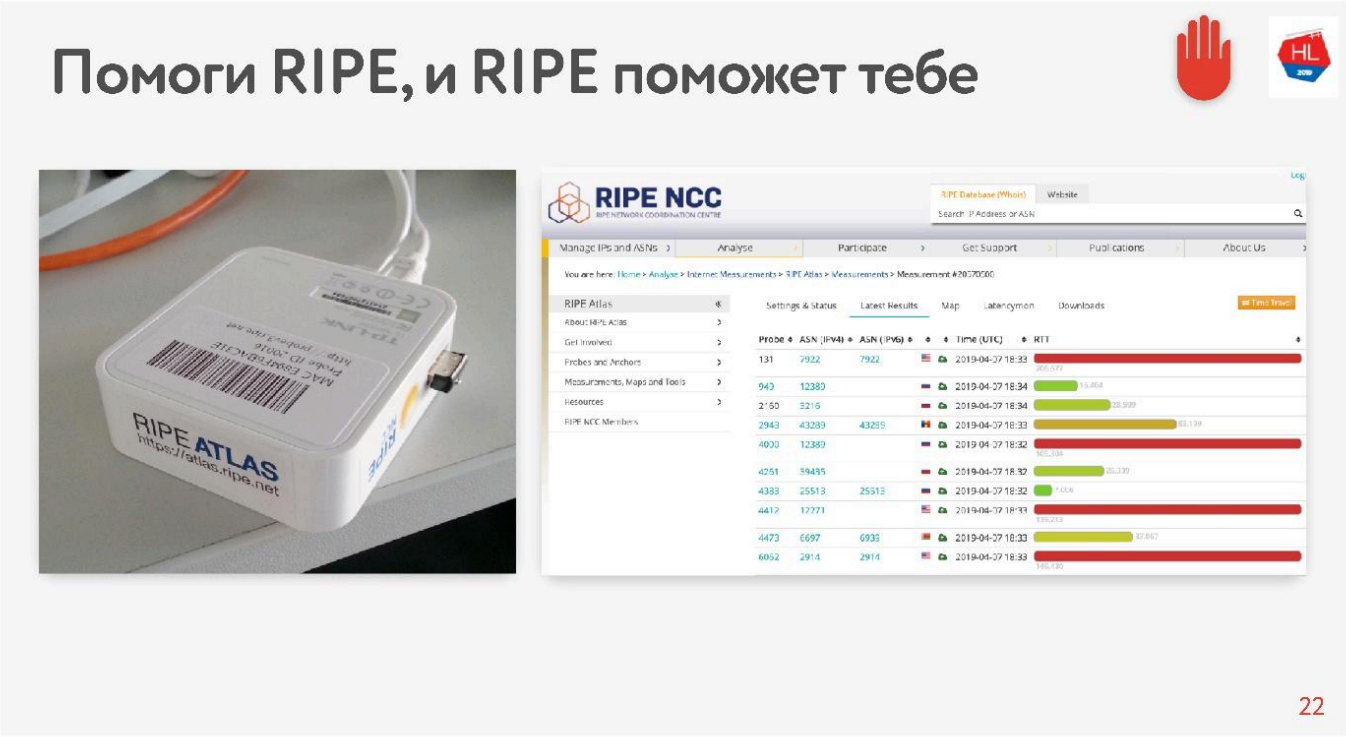
Ниже на карте средняя скорость получения данных по TCP в России. Если убрать западную часть, то видно, что скорость измеряется скорее в килобитах, чем в мегабитах.



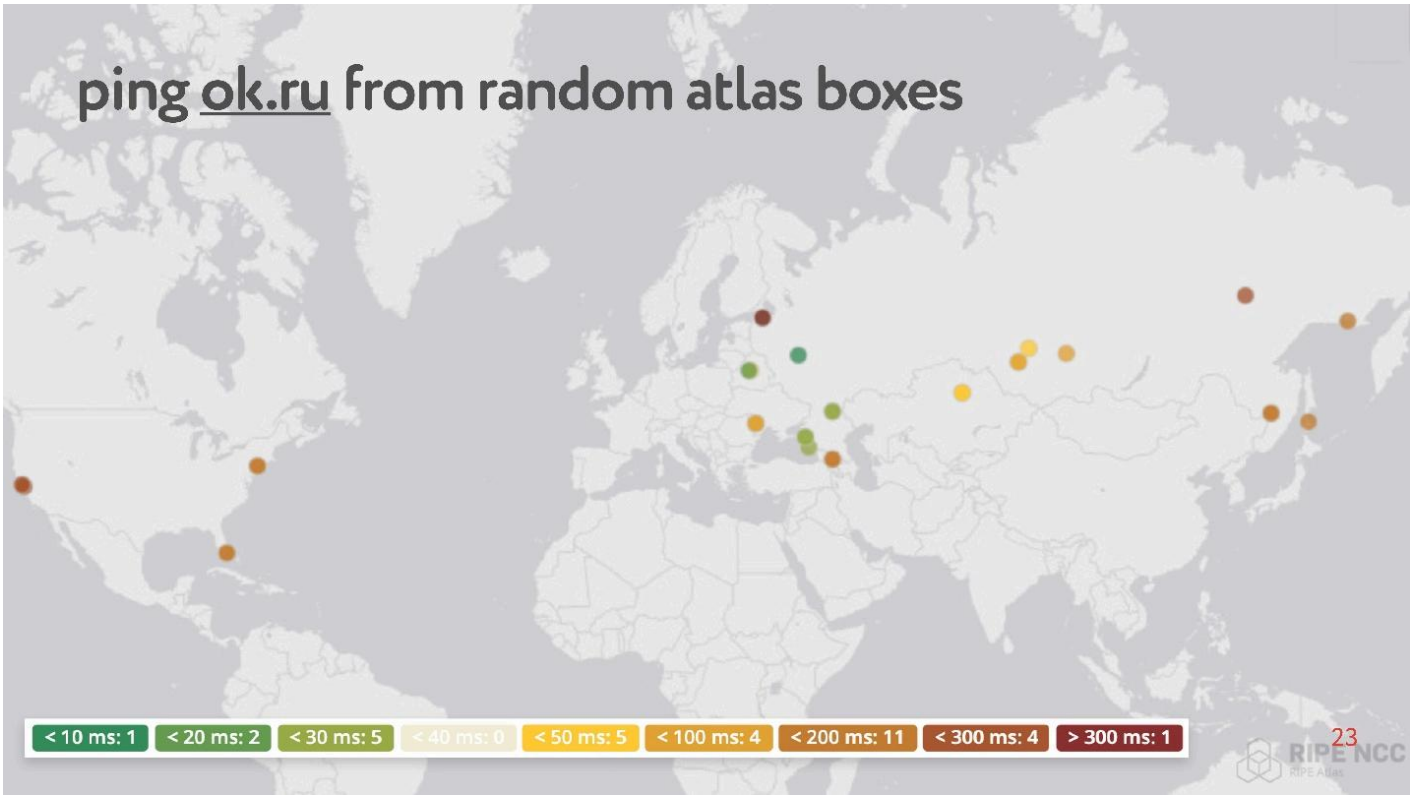

То есть в среднем у наших пользователей (если исключить западную часть России): пропускная способность 1,1 Мбит/сек, 0,6 % packet loss, RTT (round-trip time) порядка 200 мс.

## Как вычислить RTT

Когда я увидел среднее в 200мс, подумал что в статистике ошибка, и решил измерить RTT до наших серверов в МСК альтернативным способом с помощью RIPE Atlas. Это система сбора данных о состоянии Интернета. Устройство зонд от RIPE Atlas можно получить бесплатно.




Суть в том, что вы подключаете ее к домашнему интернету и собираете «карму». Она сутками работает, какие-то люди выполняют на ней какие-то свои запросы. Потом вы можете сами ставить различные задачи. Пример такой задачи: случайно взять 30 точек в интернете, и попросить померить RTT, то есть выполнить команду ping до сайта Одноклассники.




Как ни странно, среди случайных точек много таких, у которых ping от 200 до 300 мс.

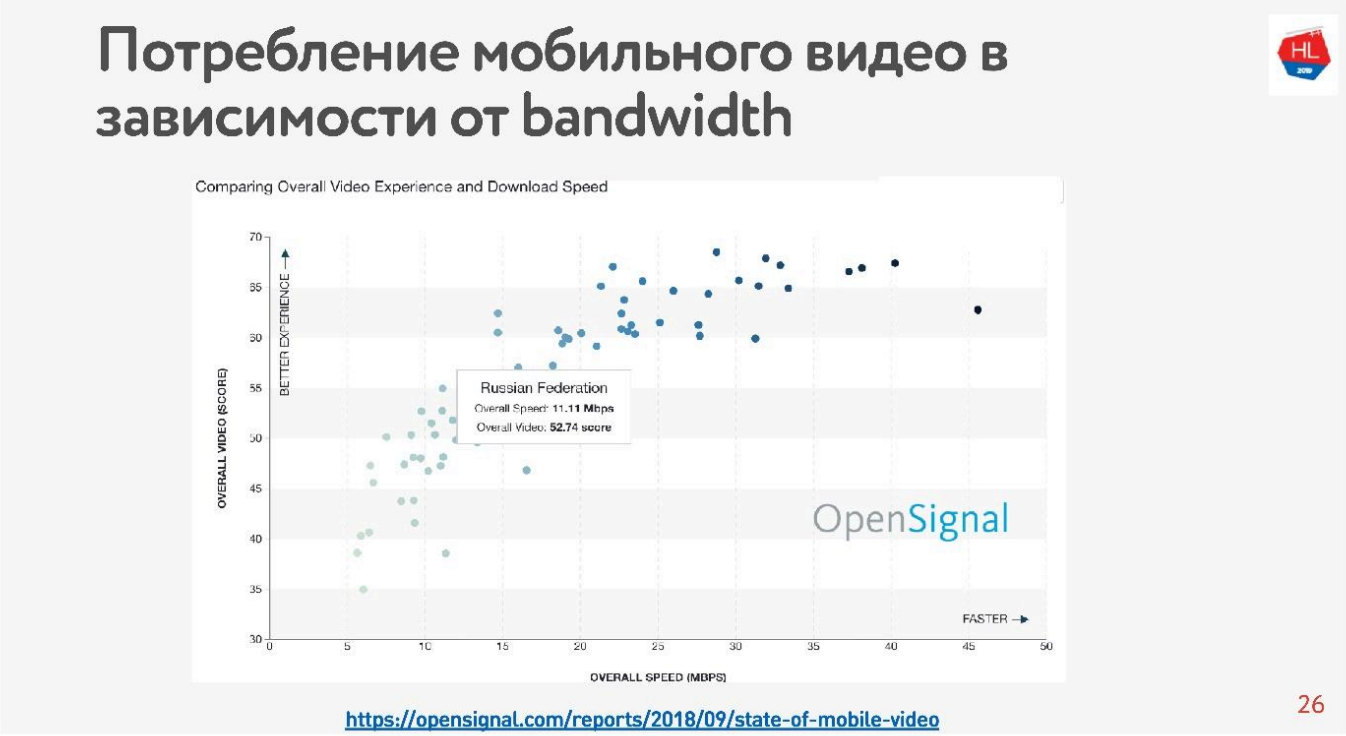
Итого, **беспроводные сети популярны и нестабильны** (хотя последнее обычно

игнорируется, так как считается, что с этим справляется TCP):

- Более 80% пользователей используют беспроводной интернет;
- Параметры беспроводных сетей динамично меняются в зависимости, например, от того, что пользователь повернул за угол;
- Беспроводные сети имеют высокие показатели packet loss, jitter, reordering;
- Фиксированный ассиметричный канал, смена IP-адреса.

## Потребление контента зависит от скорости интернета

Это очень легко проверить — есть много статистических данных. Я взял статистику по видео, которая говорит, что чем выше скорость интернета в стране, тем больше пользователи смотрят видео.




Согласно этой статистике в России достаточно быстрый Интернет, однако по нашим внутренним данным средняя скорость несколько ниже.

В пользу того, что скорость интернета в целом недостаточная, говорит то, что все создатели крупных приложений, социальных сетей, видеосервисов и так далее оптимизируют свои сервисы для работы в плохой сети. Уже после 10 Кбайт полученных данных можно увидеть минимум информации в ленте, а на скорости 500 Кбит можно смотреть видео.

## Как ускорить загрузку

В процессе разработки платформы Видео, мы поняли, что TCP не очень эффективен в беспроводных сетях. Как пришли к такому выводу?

Мы решили ускорить загрузку и сделали следующий трюк.

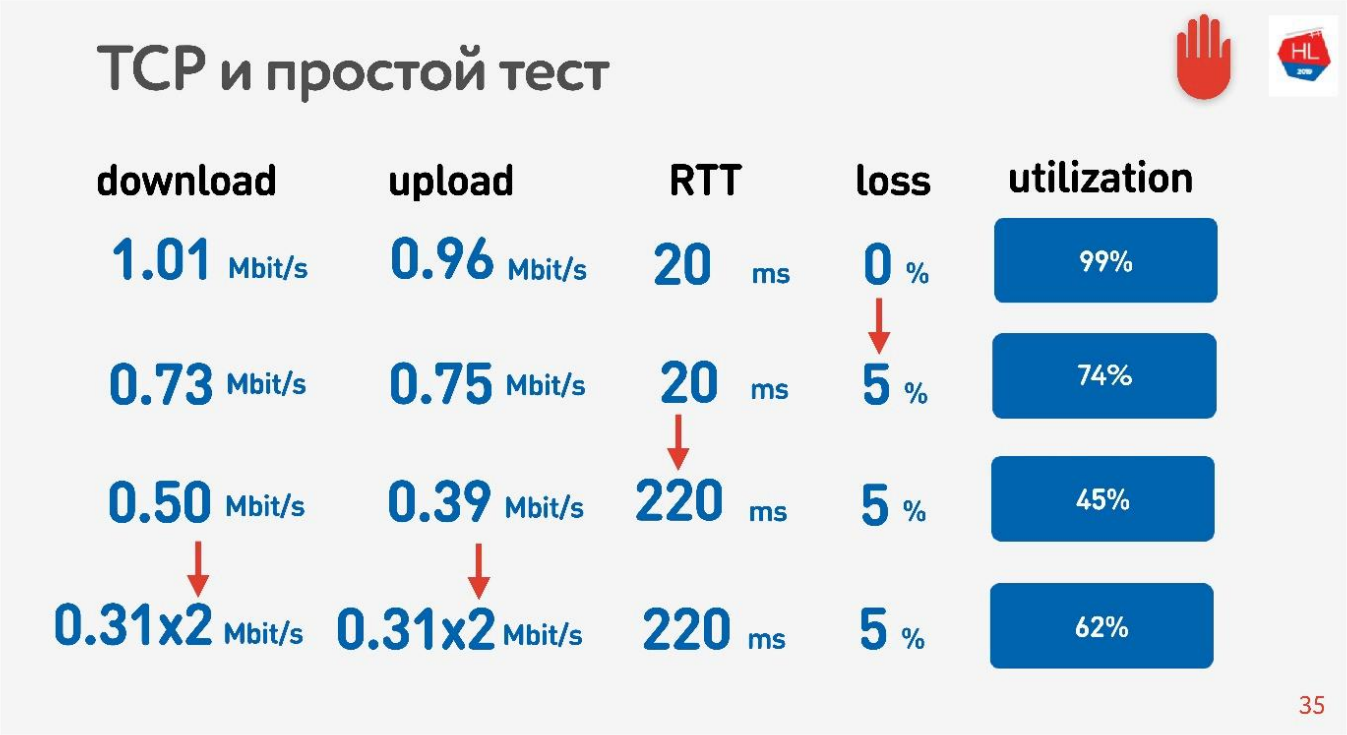

Грузили видео с клиента на сервер, в несколько потоков, то есть 40 Мбайт делим на 4 части по 10 Мбайт и загружаем их параллельно. Запустили это на Android и получили, что параллельно загружается быстрее, чем в одно соединение (демо в докладе). Самое интересное, что когда мы выкатили параллельную загрузку в продакшен, то увидели, что в некоторых регионах скорость загрузки выросла в 3 раза!

> По четырем TCP-соединениям реально можно загрузить данные на сервер в 3 раза быстрее.

Так мы повысили скорость загрузки видео и сделали вывод, что загрузку нужно распараллеливать.

## TCP в нестабильных сетях

Невероятный эффект с параллелизмом можно потрогать. Достаточно взять измеритель скорости получения/отправки данных (например Speed Test) и трафик шейпер (например network link Conditioner, если у вас Mac) Ограничиваем сеть параметрами 1 Мбит/сек на upload и download и начинаем растить потерю пакетов.

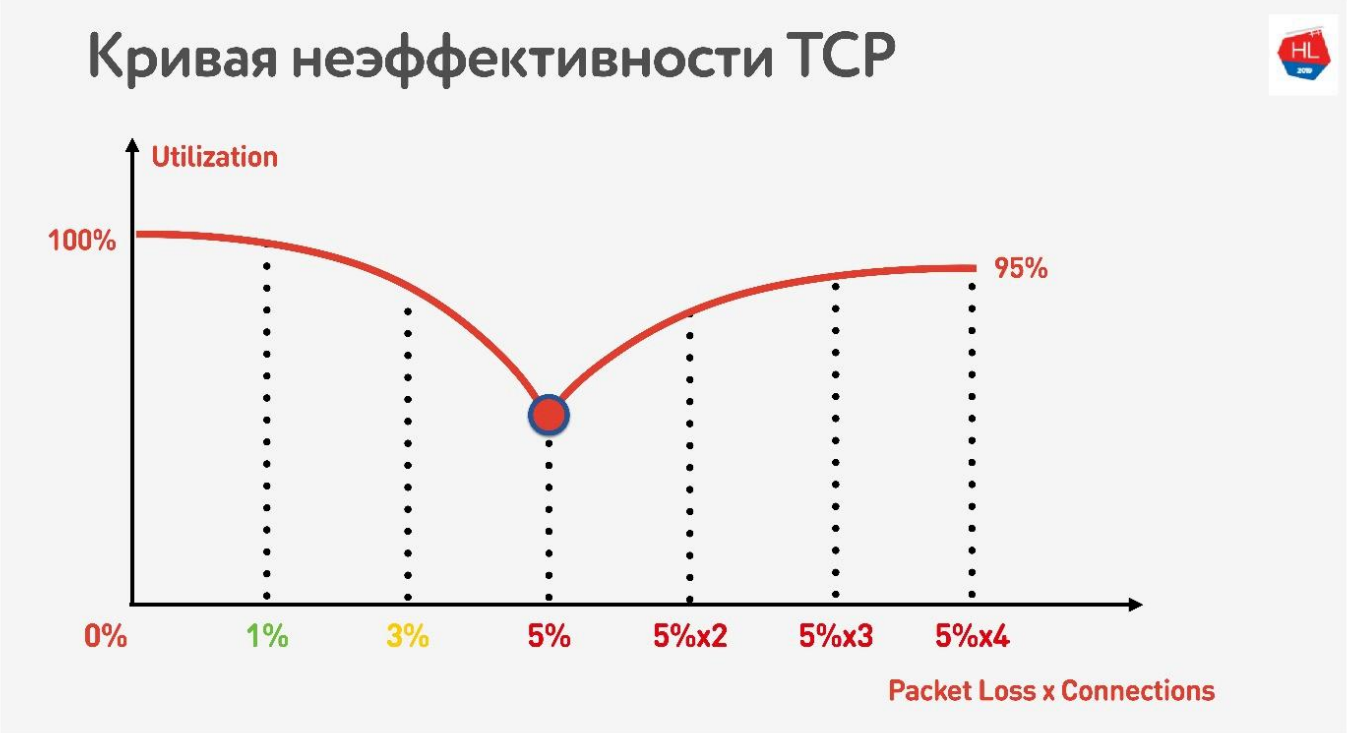



В таблице указаны RTT и потери. Видно, что в случае 0% потерь, сеть утилизирована на 100%.

Следующей итерацией увеличиваем packet loss на 5%, и видим, что сеть утилизируется всего на 74%. Вроде ничего страшного — при packet loss в 5% теряется 26% сети. Но если увеличить еще и ping, то останется **меньше половины канала**.

> Если канал с высоким RTT и большим packet loss, то одно TCP соединение не полностью утилизирует сеть.

Дальнейший трюк показывает, что если начать использовать параллельные TCP-соединения (вы можете просто запустить несколько Speed Test-ов одновременно), виден обратный рост утилизации канала.




С увеличением числа параллельных TCP-соединений утилизация сети становится почти равной пропускной способности, за вычетом процента потерь.

Таким образом, получилось:

- Беспроводные мобильные сети победили и нестабильны.
- TCP не до конца утилизирует канал в нестабильных сетях.
- Потребление контента зависит от скорости интернета: чем выше скорость интернета, тем больше пользователи смотрят, а мы очень любим наших пользователей и хотим, чтобы они смотрели больше.

Очевидно, надо куда-то двигаться и рассмотреть альтернативы TCP.

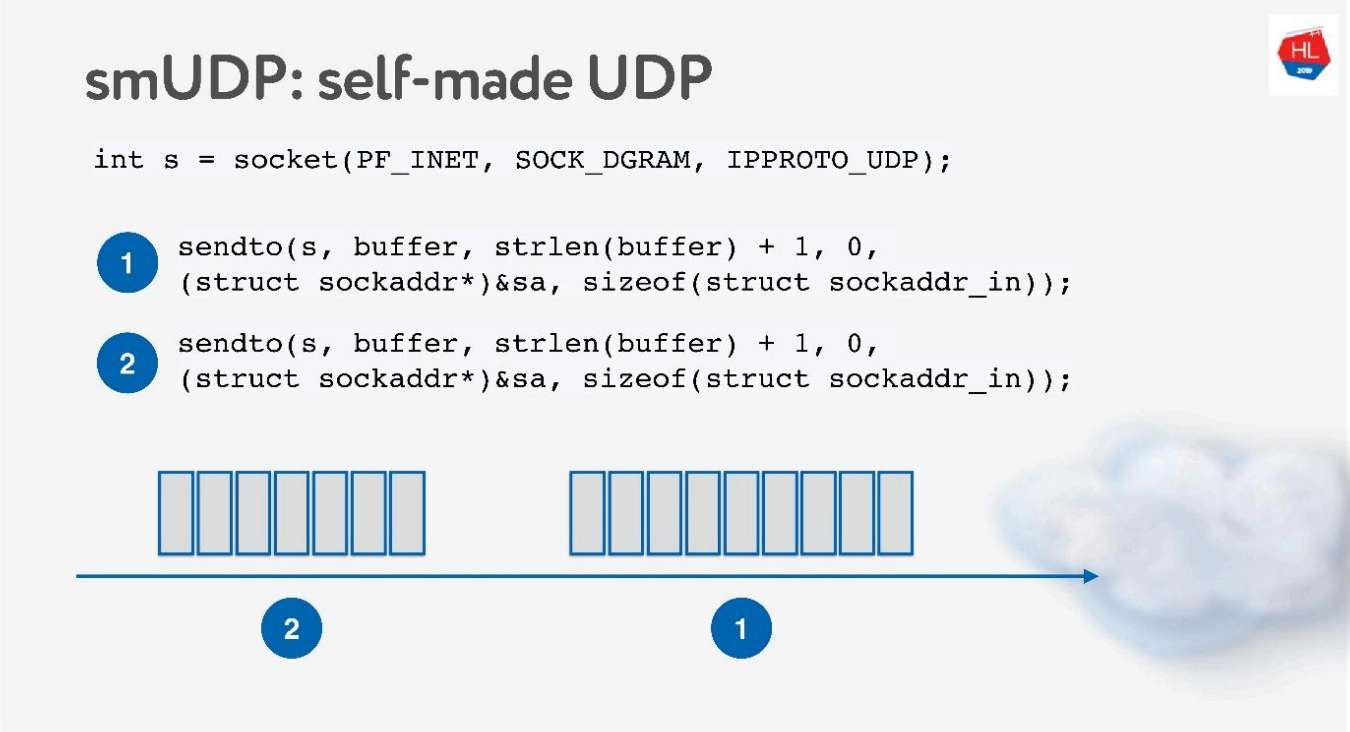
## TCP vs не TCP

С чем сравнить тёплое? Есть два варианта.

Первый вариант — на уровне IP есть TCP и UDP, мы можем позволить себе еще какой-то протокол сверху. Очевидно, что если параллельно с TCP и UDP запустить свой протокол, то про него не будут знать Firewall, Brandmauer, маршрутизаторы и весь остальной мир, участвующий в доставке пакетов. В итоге придется годами ждать, когда все оборудование обновится и начнет работать с новым протоколом.

Второй вариант — сделать свой надежный протокол доставки данных поверх ненадежного UDP. Очевидно, что ждать, пока Linux, Android и iOS добавят новый протокол в свое ядро можно долго, поэтому надо пилить протокол в User Space.

Такое решение кажется интересным, будем называть его self-made UDP-протокол. Чтобы начать его разрабатывать, не нужно ничего особенного: просто открываем UDP socket и отправляем данные.




Будем его развивать, параллельно изучая, как работает сеть.

## TCP vs self-made UDP

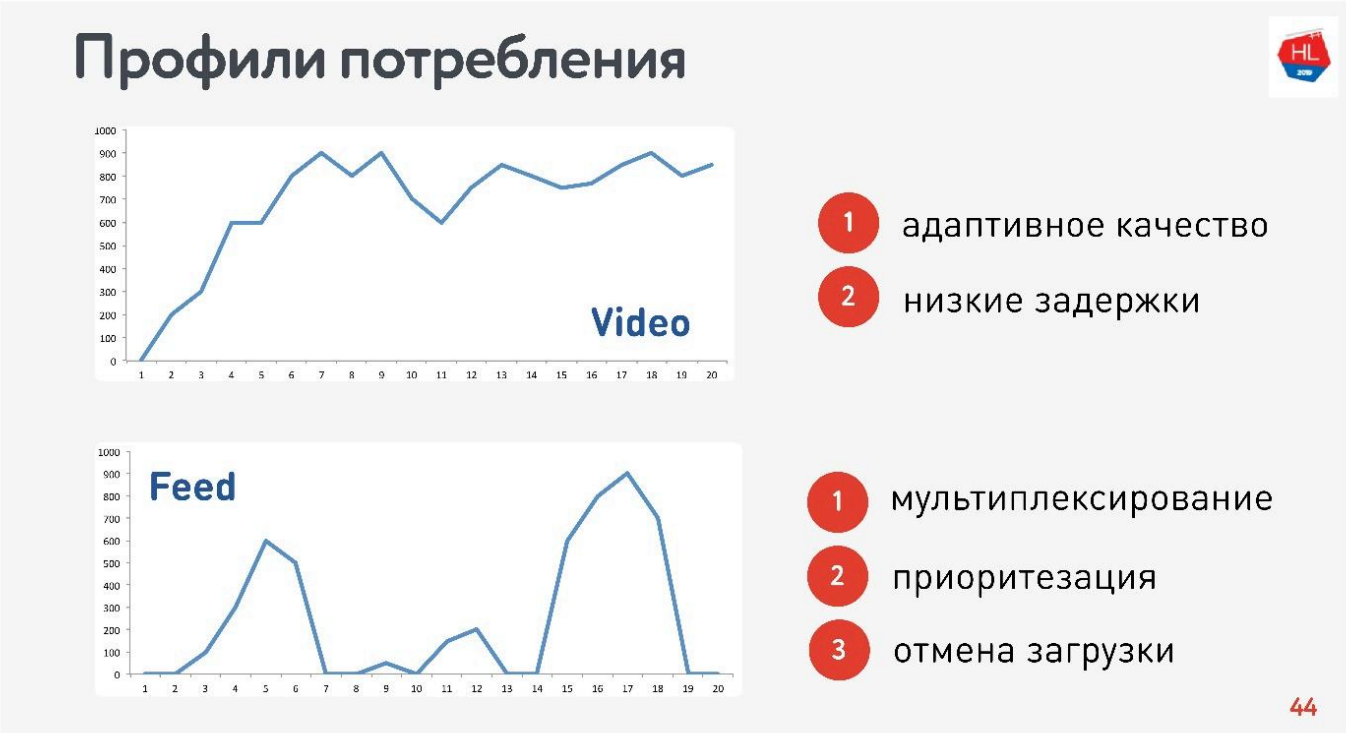
Хорошо, а на чем сравнивать?

Сети бывают разные:

- С перегрузками, когда пакетов очень много и некоторые из них дропаются из-за перегрузки каналов или оборудования.
- Высокоскоростные с большими round-trip (например когда сервер располагается относительно далеко).
- Странные — когда в сети вроде бы ничего не происходит, но пакеты все равно пропадают просто потому-что Wi-Fi точка доступа находится за стенкой.

Профили сети вы всегда можете потрогать сами: выбрать на своем телефоне тот или иной профиль и запустить Speed Test.


Кроме профилей сети, нужно еще определиться с профилем потребления трафика. Вот те, которые использовали мы:



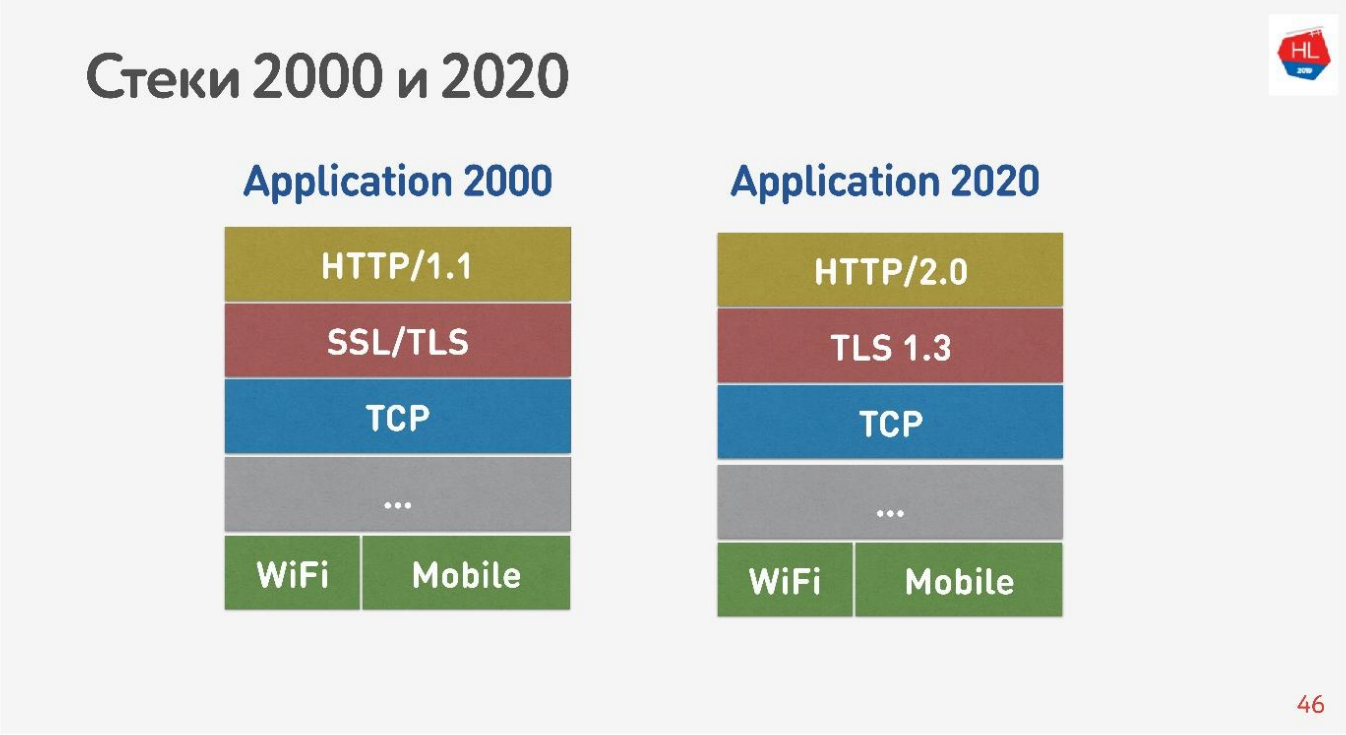
Так как я отвечаю за Видео и Ленту, то профили соответствующие:

- Профиль Видео, когда вы подключаетесь и стримите тот или иной контент. Скорость соединения увеличивается, как на верхнем графике. Требования к этому протоколу: низкие задержки и адаптация битрейта.
- Вариант просмотра Ленты: импульсная загрузка данных, фоновые запросы, промежутки простоя. Требования к этому протоколу: получаемые данные мультиплексируются и приоритизируются, приоритет пользовательского контента выше фоновых процессов, есть отмена загрузки.

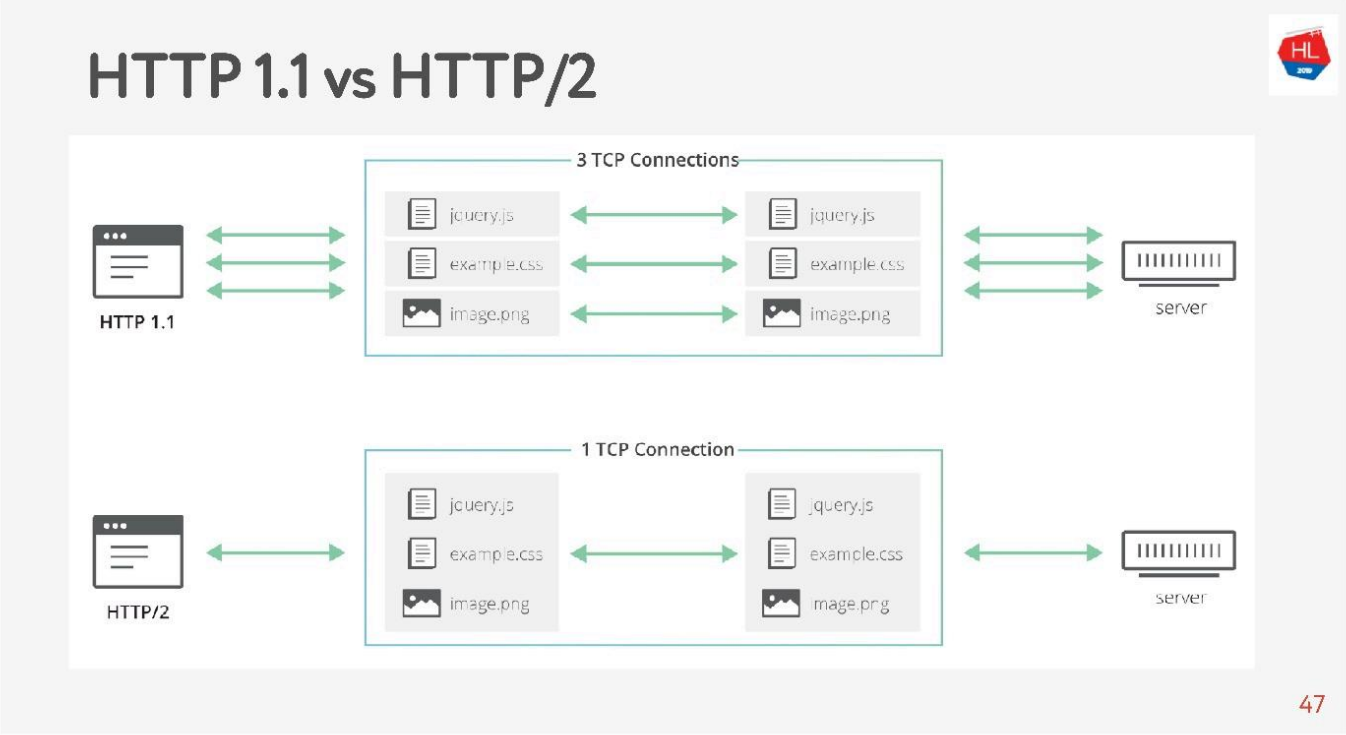
Конечно, сравнивать протоколы нужно на самых популярных HTTP.

## HTTP 1.1 и HTTP 2.0

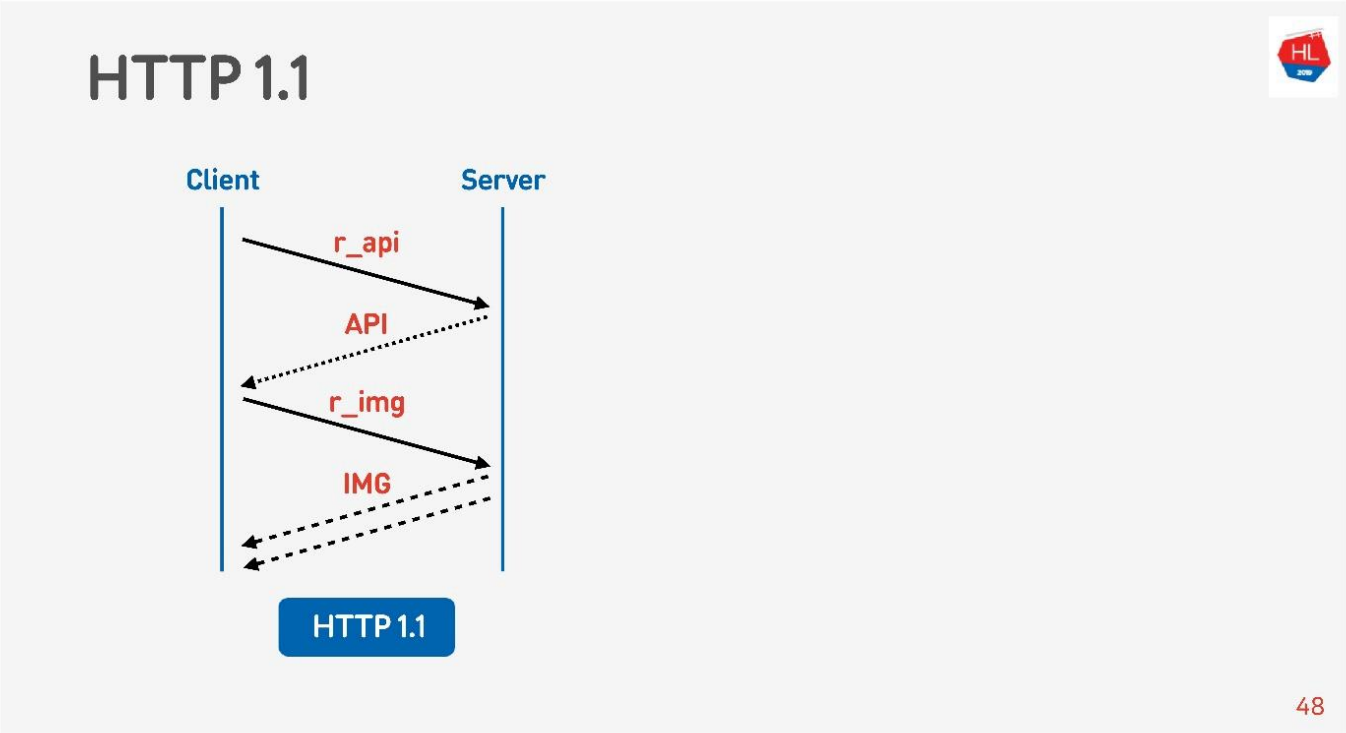
Стандартный стек 2000-х выглядел как HTTP 1.1 поверх SSL. Современный стек — это HTTP 2.0, TLS 1.3, и все это поверх TCP.



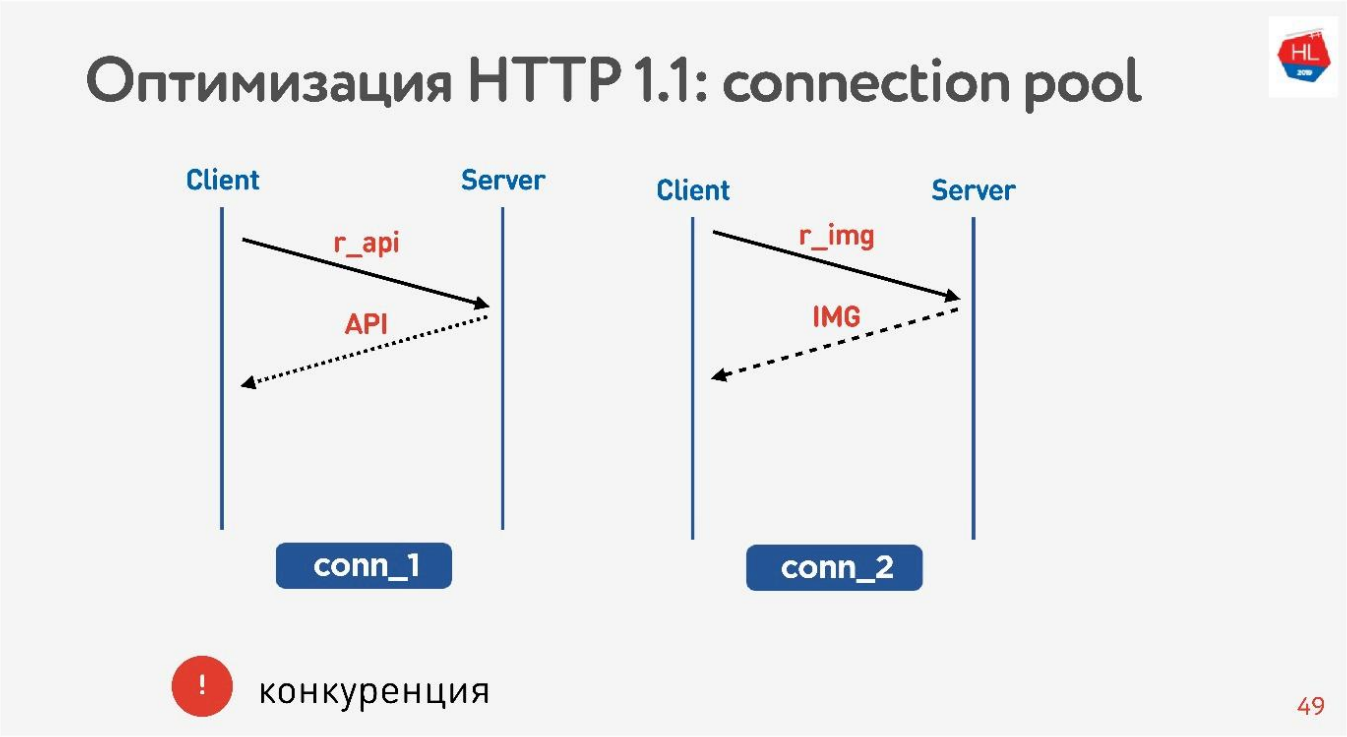

Основное отличие в том, что HTTP 1.1 использует ограниченный пул соединений в браузере к одному домену, поэтому делают отдельный домен для картинок, для данных и так далее. HTTP 2.0 предлагает одно мультиплексированное соединение, в котором передаются все эти данные.




HTTP 1.1 работает так: делаете запрос, получаете данные, делаете запрос, получаете данные.




Обычно браузер или мобильное приложение пулит, то есть соединение на получение картинок, данных по API, и вы параллельно выполняете запрос за картинкой, за API, за видео и так далее.




Основная проблема — конкуренция. Вы никак не управляете отправленными запросами. Вы понимаете, что пользователю уже не нужна картинка, которую он пролистал, но ничего не можете сделать.

> С HTTP 1.1 вы все равно получаете то, что запросили, отменить загрузку трудно.

Единственный шанс — socket close — это закрыть соединение. Дальше увидим, почему это плохо.

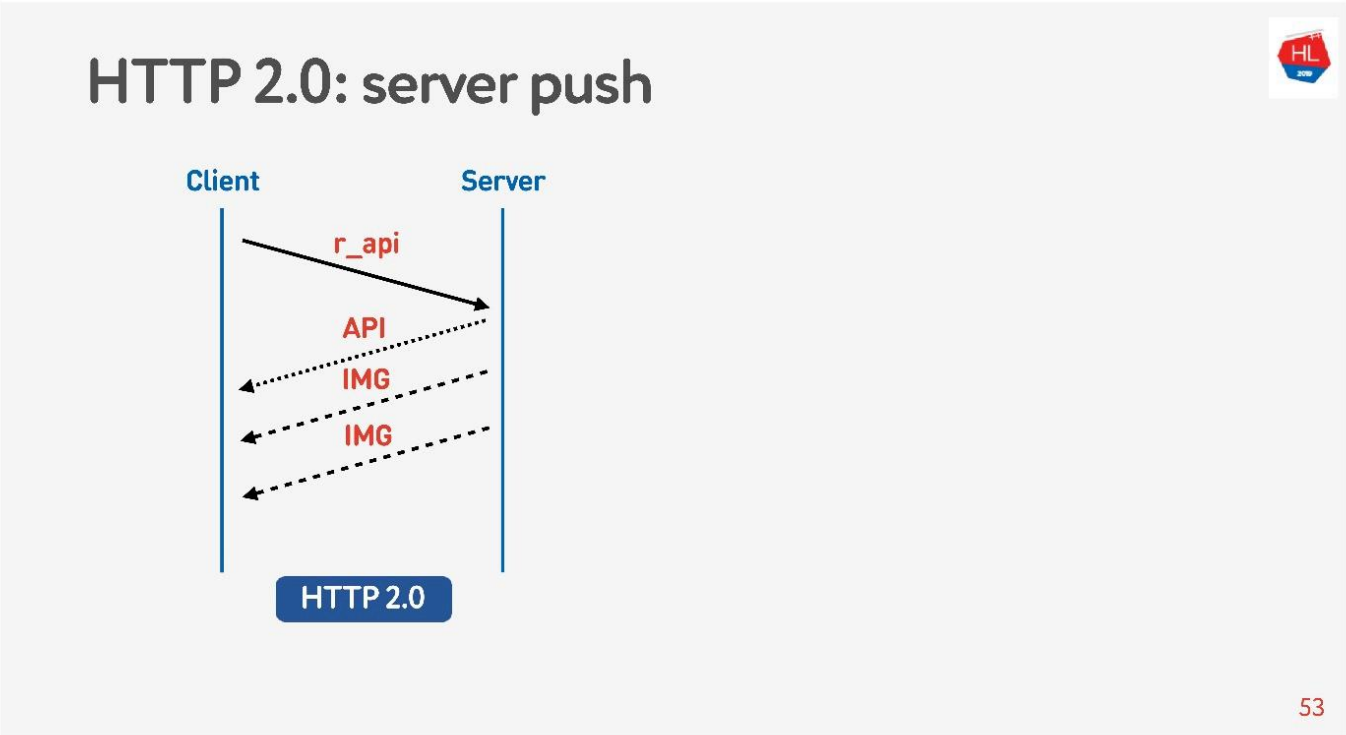
### Отличия HTTP 2.0

HTTP 2.0 решает эти проблемы:

- бинарный, сжатие заголовков;
- мультиплексирование данных;
- приоритизация;
- возможна отмена загрузки;
- server push

Рассмотрим более детально важные для нас моменты.


Запрашиваем картинку и API. Картинка сразу отдается, API подготовился через некоторое время. Отдался API — отдалась до конца картинка. Все это происходит прозрачно. **Высокоприоритетный контент загружается раньше.**




**Server push** — это такая штука, когда вы попросили что-то конкретное типа API, но еще в нагрузку на клиенте закэшировались картинки, которые точно понадобятся для просмотра, например, ленты.

Еще есть команда **Reset stream**, которую браузер выполняет сам, если вы переходите между страницами и т.д. Для мобильного клиента с её помощью можно отказаться от получения данных, при этом не разрывая соединение.

Таким образом будем сравнивать TCP на разных:

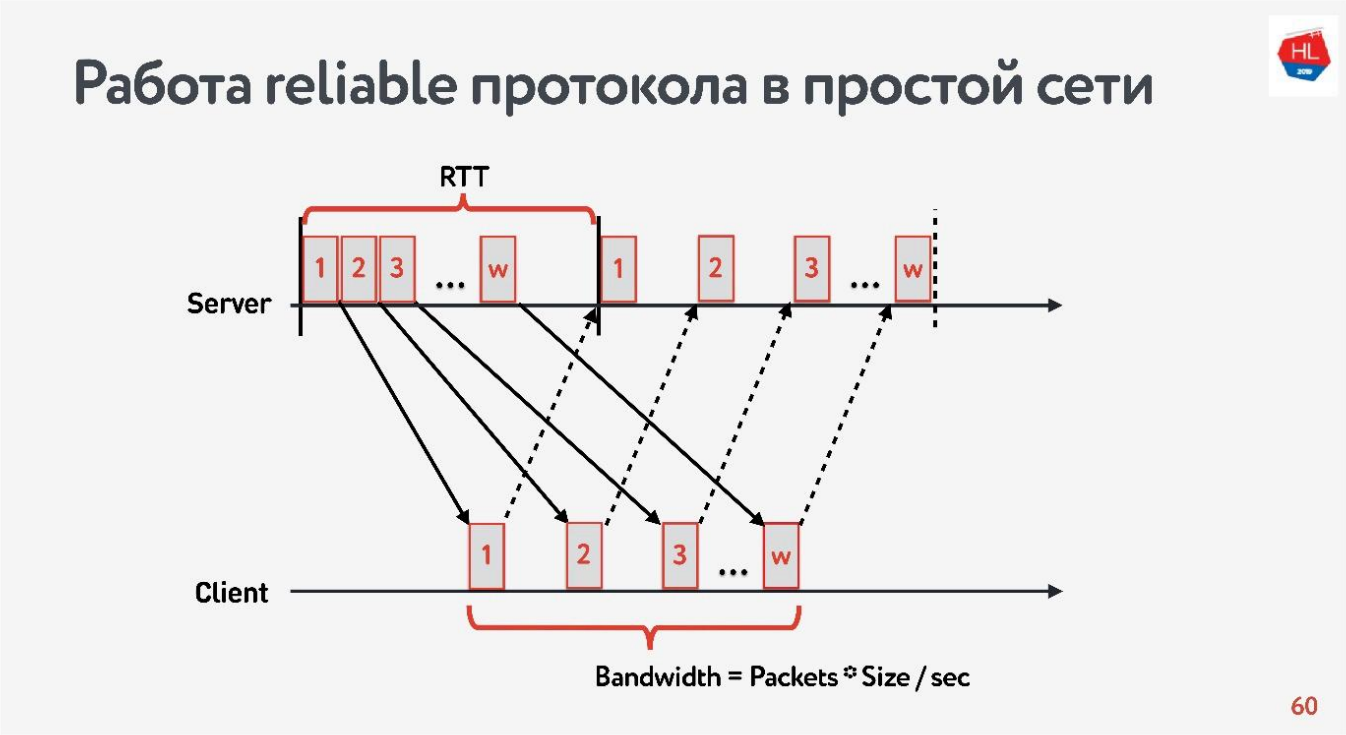
- Профилях сети: Wi-Fi, 3G, LTE.
- Профилях потребления: стриминг (видео), мультиплексирование и приоритизация с отменой загрузки (HTTP/2) для получения контента ленты.

## Модель без потерь

Начнем сравнение с простой сетью, в которой существует только два параметра: round-trip time и bandwidth.

**RTT** — это ping, время оборота пакета, получения acknowledgement или время эха на response.

Чтобы измерить **bandwidth** — пропускную способность сети — отправляем пачку пакетов и считаем количество прошедших пакетов на каком-то временном интервале.



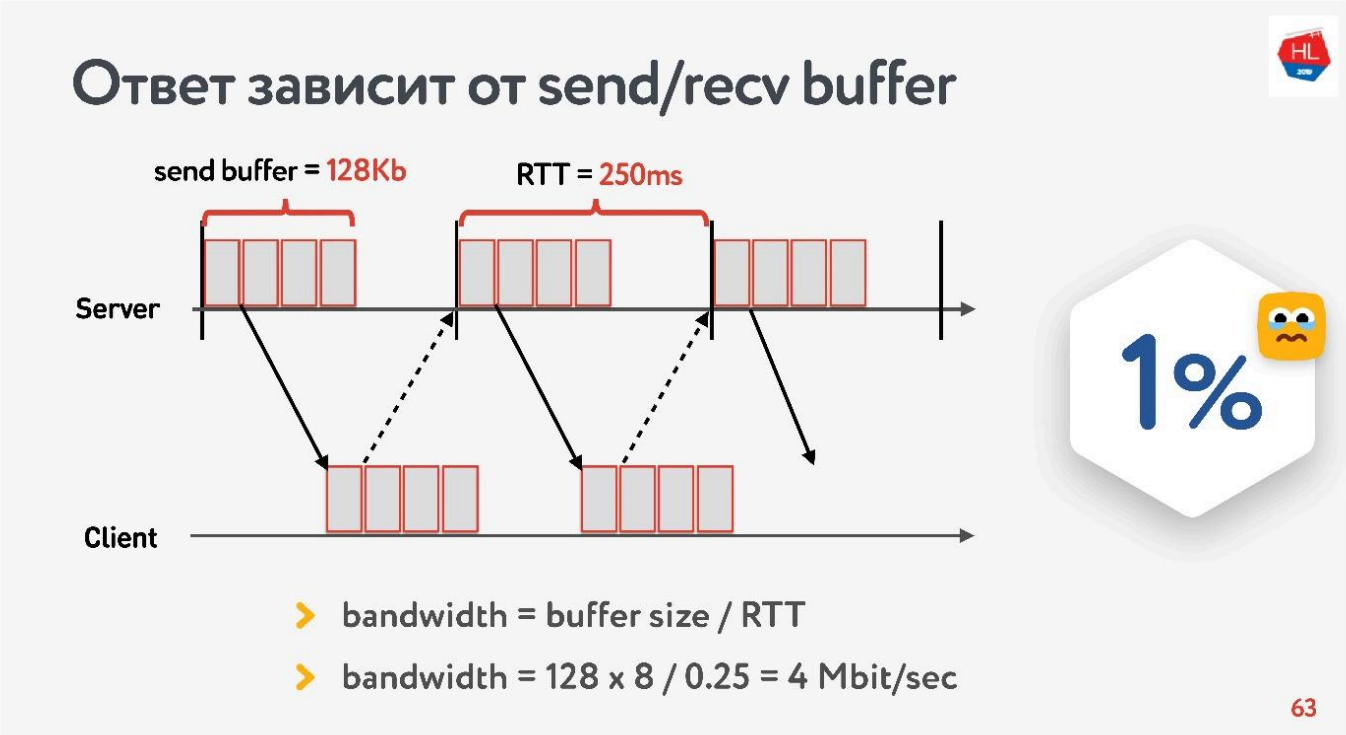

Так как мы работаем с надежными протоколами, то, конечно, есть acknowledgement —

отправляем пакеты и получаем подтверждение о получении.

## Задача про медленный интернет

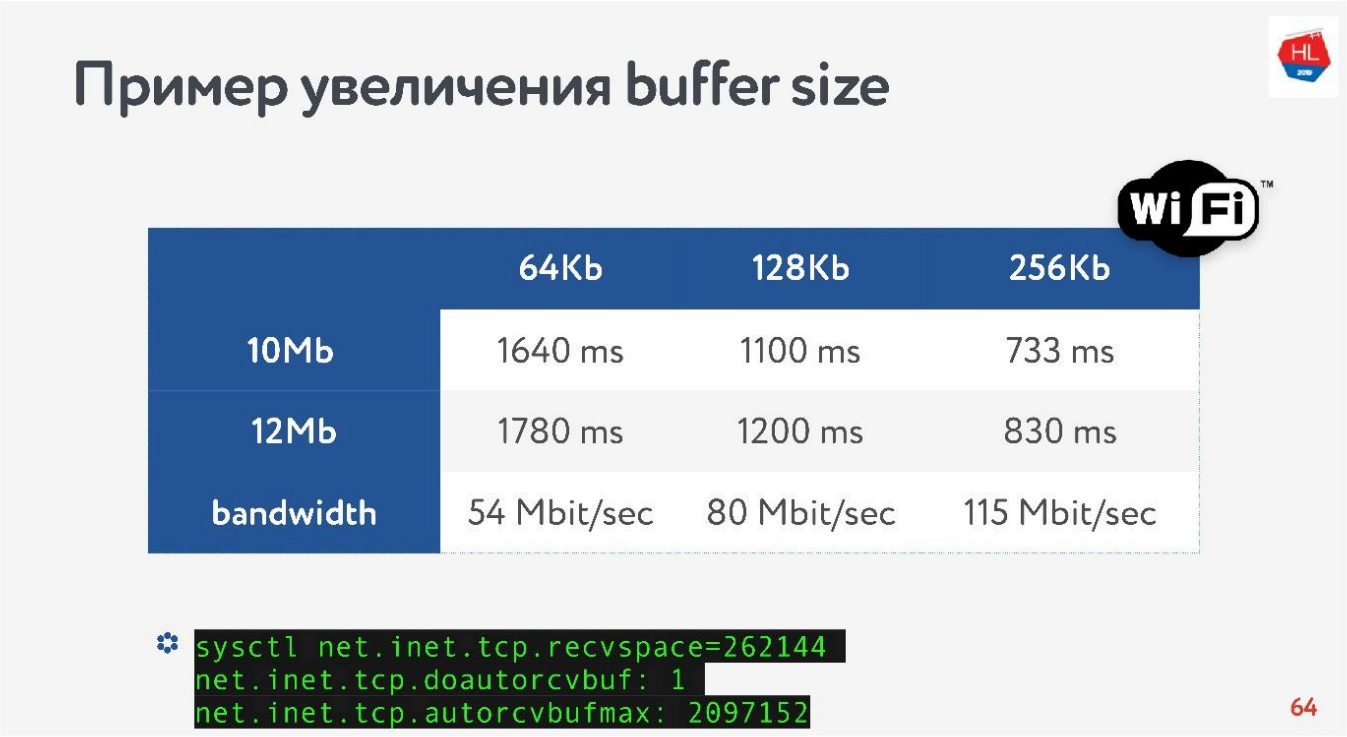
На заре разработки нашего видеосервиса в 2013 году мой друг поехал в Калифорнию и решил посмотреть новую серию своего любимого сериала на Одноклассниках. У него был RTT в 250 мс, идеальный Wi-Fi 400 Мбит/с в кампусе Google, он хотел посмотреть новую серию всего лишь в FullHD.

Как вы думаете, смог ли он посмотреть видео? Ответ зависит от настройки send/recv buffer на наших серверах.



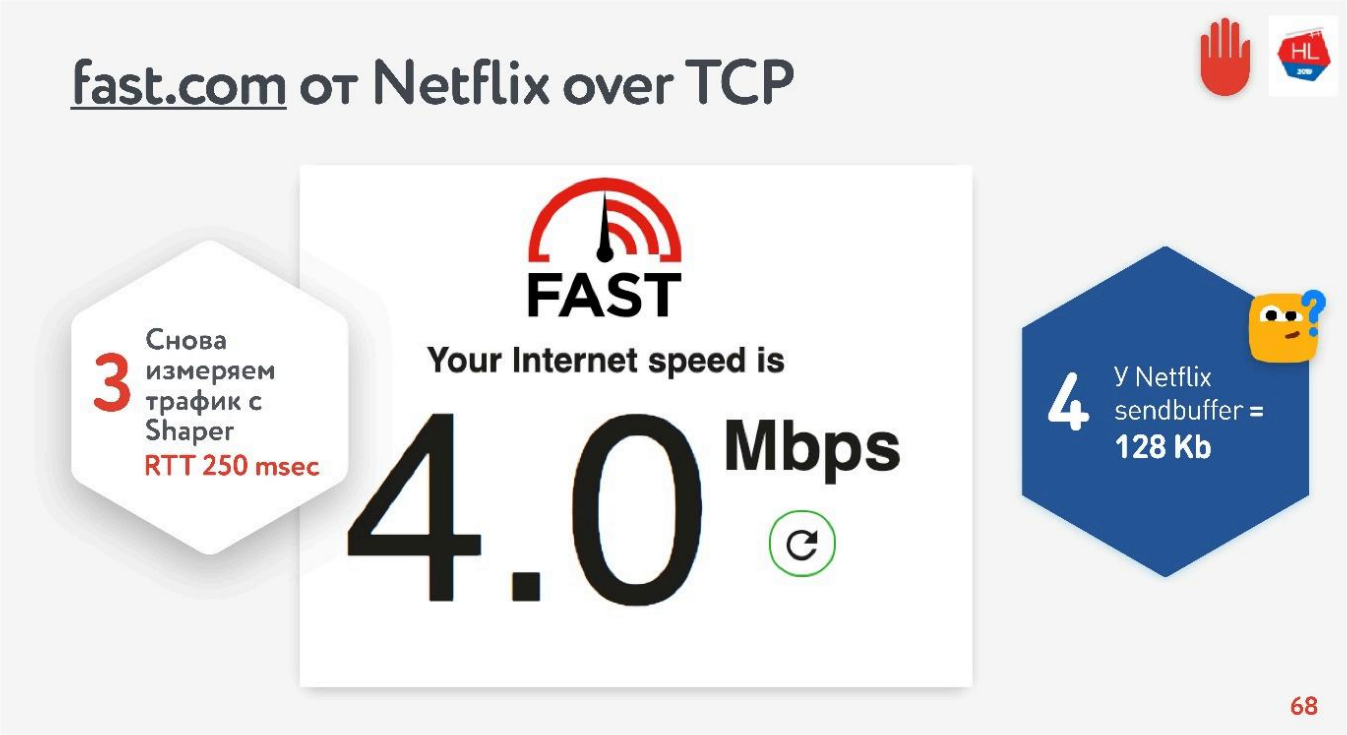

Так как у нас протокол с acknowledgement, то все данные, которые не получили подтверждения о доставке, хранятся в буфере. Если send buffer ограничен 128 Кб, то эти 128 Кб меньше, чем за RTT, мы отправить не можем. Таким образом, от нашей сети в 400 Мбит/с осталось 4 Мбит/с. Этого недостаточно, чтобы онлайн смотреть видео в FullHD.

Тогда я потюнил размер буфера и посмотрел, как действительно меняется скорость отдачи одного сегмента видео в зависимости от изменения размера буфера. Сразу оговорюсь, что recv buffer подстраивался автоматически, т.е. то, что отправлял сервер, клиент всегда мог принять.




Очевидный рецепт TCP: если передаёте высокоскоростные данные на большие расстояния, нужно увеличить буфер отправки.

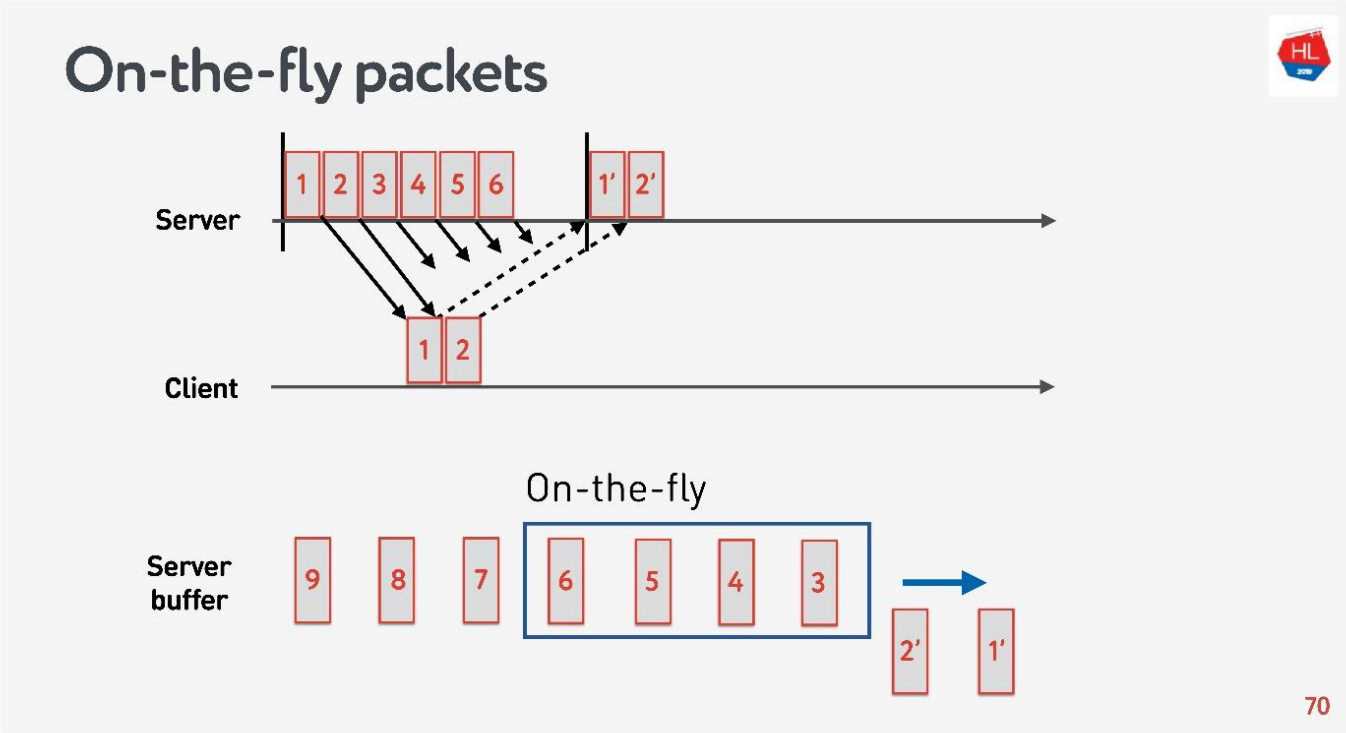
Кажется, все неплохо. Можно зайти на сервис fast.com, который померяет скорость вашего интернет до серверов Netflix. Из офиса я получил скорость 210 Мбит/с. А потом через net shaper настроил условия задачи и зашел на этот сайт еще раз. Магия — я получил 4 Мбит/с ровно.



Как я ни крутил, не получилось от Netflix добиться буфера больше 128 Кбайт.

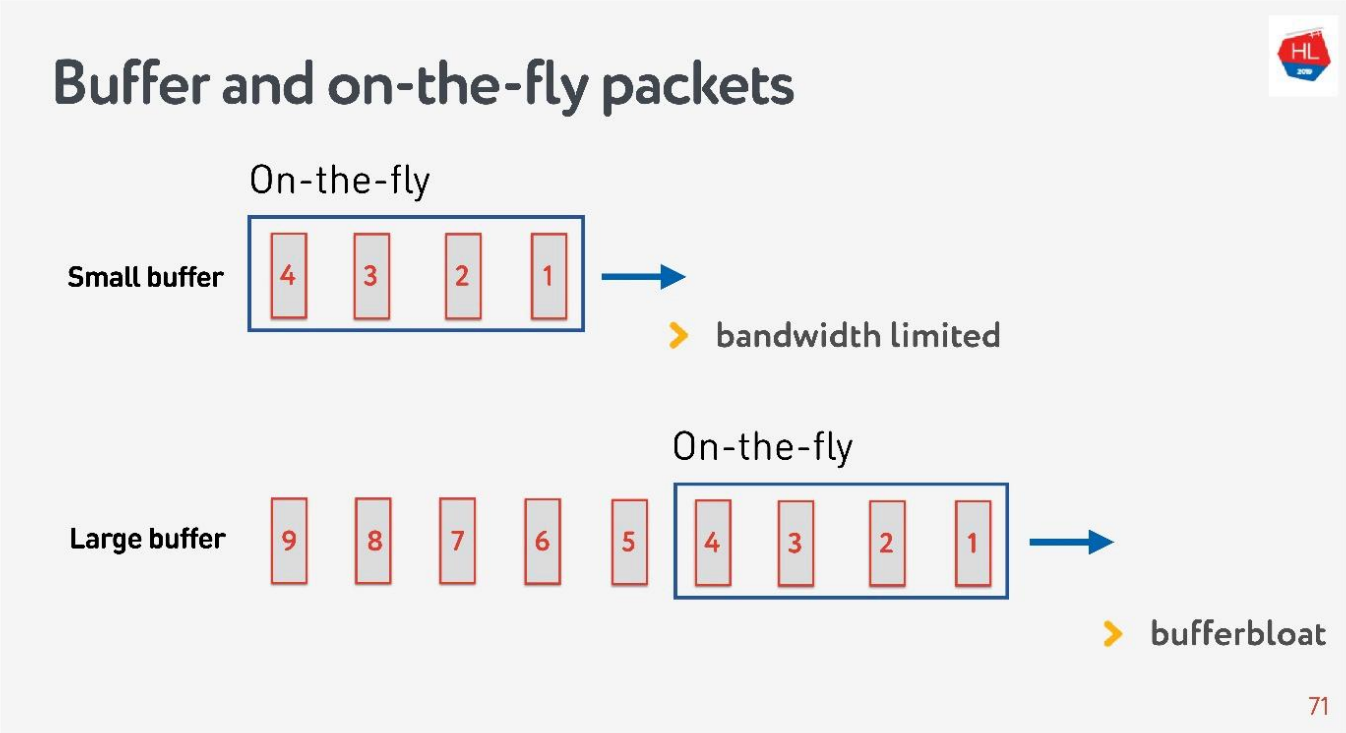
## Размер буфера

Для того чтобы разобраться с оптимальным размером буфера, нужно понять, что такое On-the-fly packets.



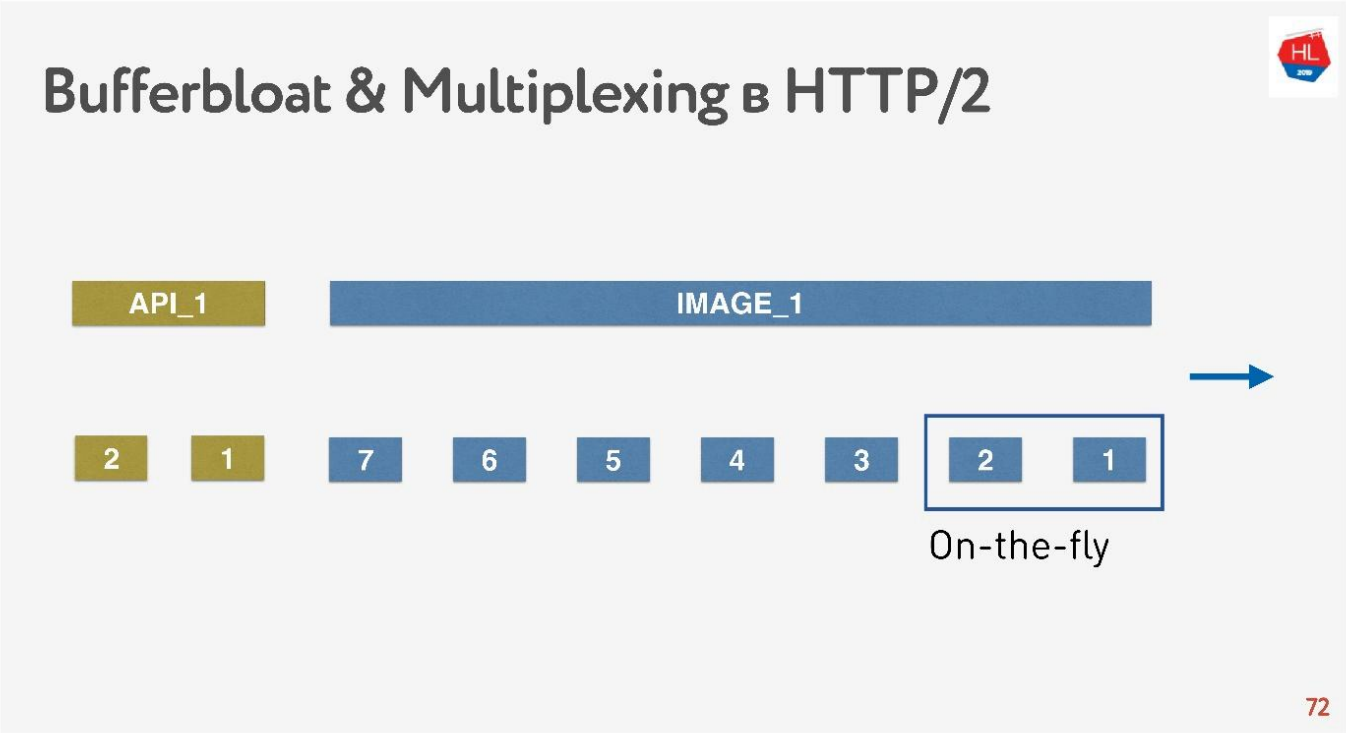

Есть состояние сети:

- пакеты 1 и 2 уже отправлены, для них получено подтверждение;
- пакеты 3, 4, 5, 6 отправлены, но результат доставки неизвестен (on-the-fly packets);
- остальные пакеты находятся в очереди.




Если количество пакетов в On-the-fly равно размеру буфера, то он недостаточного размера. В этом случае сеть голодает, не до конца используется.

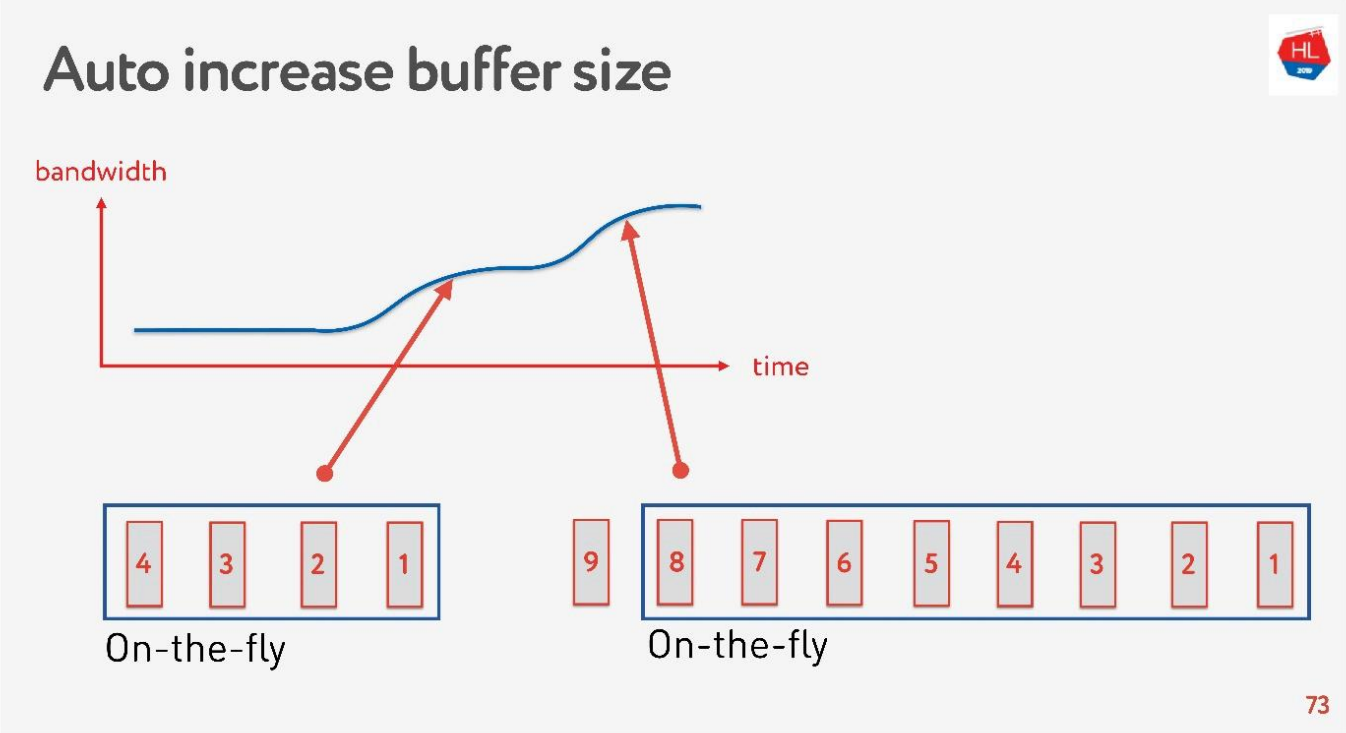
Возможна обратная ситуация — слишком большой буфер. В этом случае происходит распухание буфера. Чем это плохо?




Если говорить про мультиплексирование данных и отправлять несколько запросов

одновременно, например, картинки в это же соединение и API, то когда вся огромная мегабайтная картинка влезла в буфер, а мы пытаемся запихнуть еще и высокоприоритетный API, то буфер распухает. Придется очень долго ждать, когда картинка уйдет.

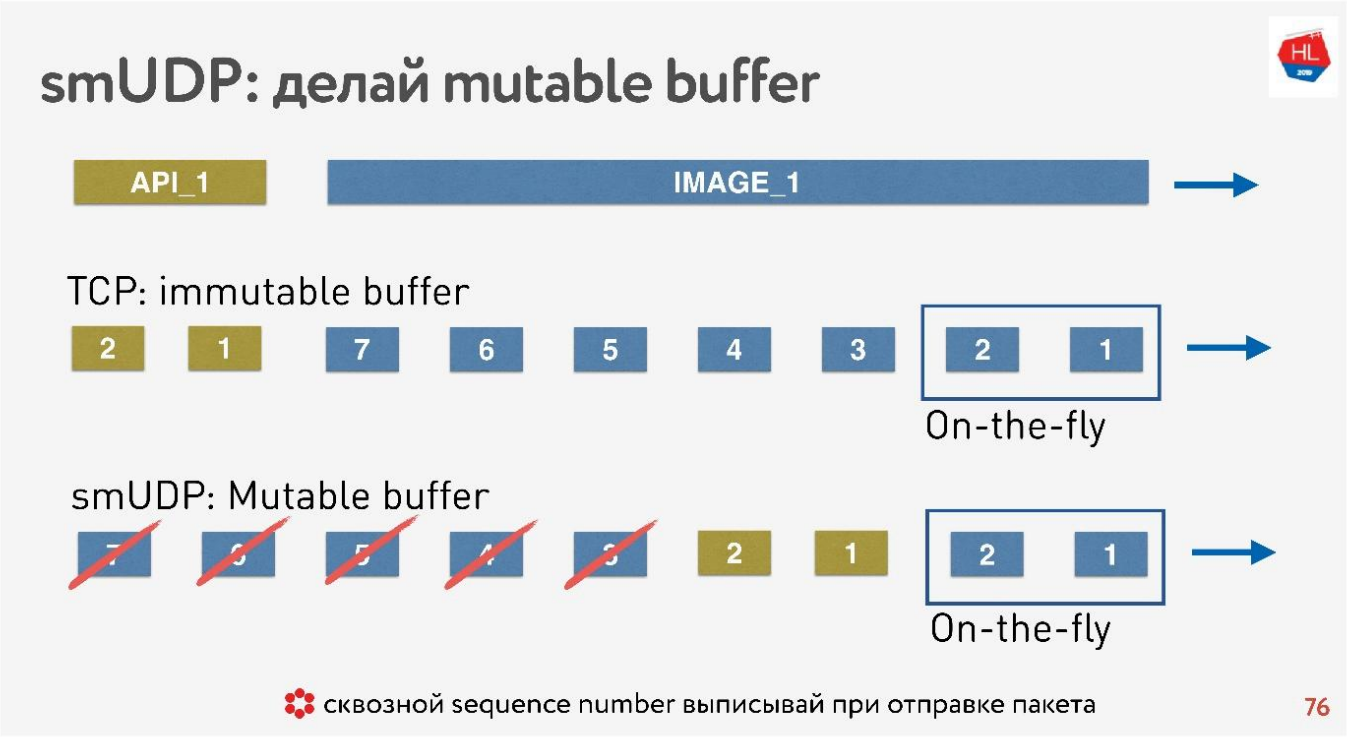
Простым решением является автоматическая настройка размера буфера. Сейчас это доступно на многих клиентах и работает примерно так.




Если сейчас может быть отправлено много пакетов, буфер увеличивается, передача данных ускоряется, размер буфера растет, вроде бы все здорово.

Но есть проблема. Если буфер увеличился, его нельзя так просто уменьшить. Это более сложная задача. Если скорость проседает, то происходит то самое распухание буфера. Буфер довольно большой и весь заполнен, нам нужно ждать, пока все данные отправятся на клиент.

Если мы пишем свой UDP-протокол, то все очень просто — у нас есть доступ к буферу.




Если TCP в таких ситуациях просто добавляет данные в конец, и вы ничего не можете сделать, то в self-made протоколе можно помещать данные, например, вперед, сразу же за On-the-fly packets.

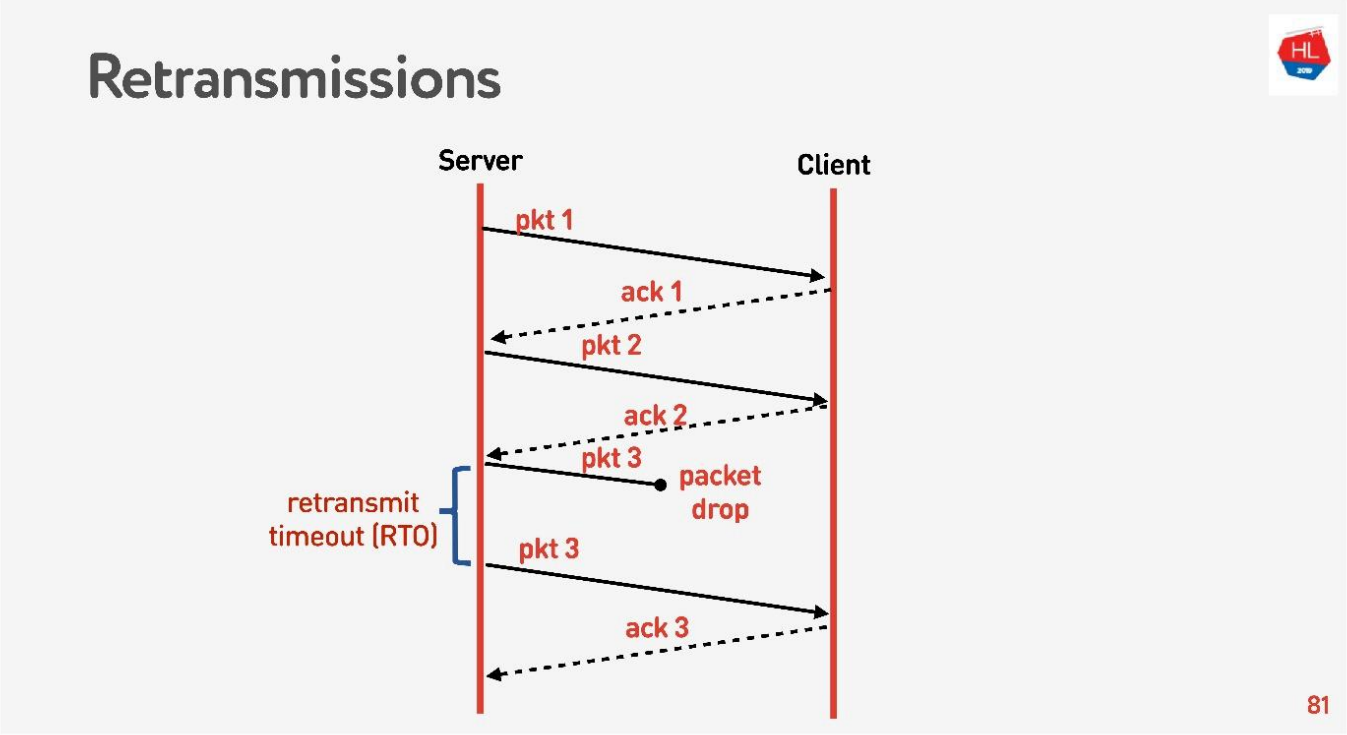
А если придет cancel, и клиент скажет, что эта картинка больше не нужна, ему нужны API данные, он пролистал контент дальше, можно все это выбросить из буфера и отправить нужное.

Как это делается? Известно, что чтобы восстанавливать пакеты, управлять доставкой, получать acknowledgements, нужен какой-то sequence_id пакетов. Sequence_id мы выписывается только для on-the-fly packets, то есть выдаем его только, когда отправляем пакеты. Все остальное в буфере можно передвигать как хотим до тех пор, пока пакеты не ушли.

**Вывод:** в TCP буфер надо правильно настроить, поймать баланс, чтобы не упираться в сеть и не раздувать буфер. Для собственного UDP-протокола все просто — этим можно управлять.

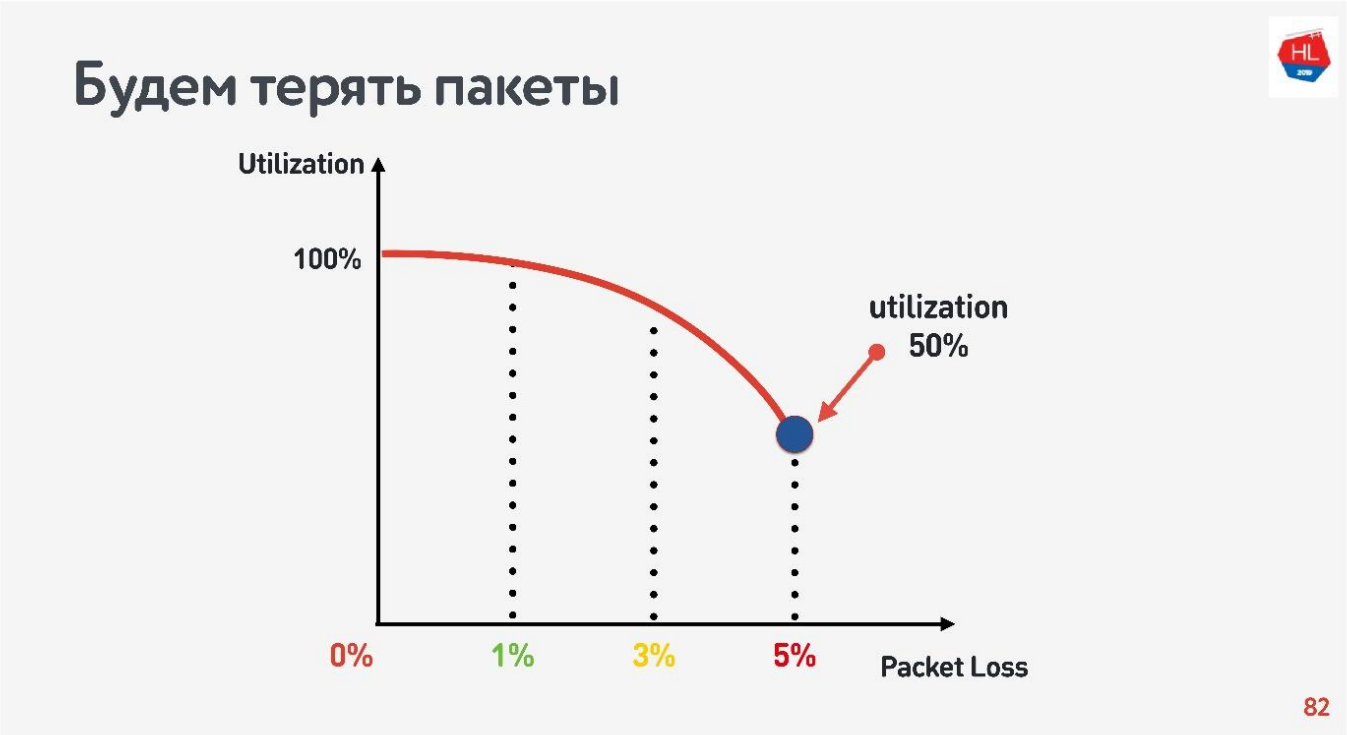
## Модель сети с потерями

Передвигаемся на уровень выше, сеть становится чуть-чуть сложнее, в ней появляется packet loss. Для мобильных сетей это обычная ситуация. Часть из отправленных пакетов не доходит до клиента. Стандартный алгоритм восстановления retransmit работает примерно так:




Отправляет пакеты, на каждый пакет получает acknowledgement. Если через Retransmit timeout (RTO) равному RTT плюс некоторые константы подтверждения нет, то перепосылает пакет.

Вернемся к кривой неэффективности TCP, когда теряется всего 5% пакетов, а утилизация сети равна 50%.

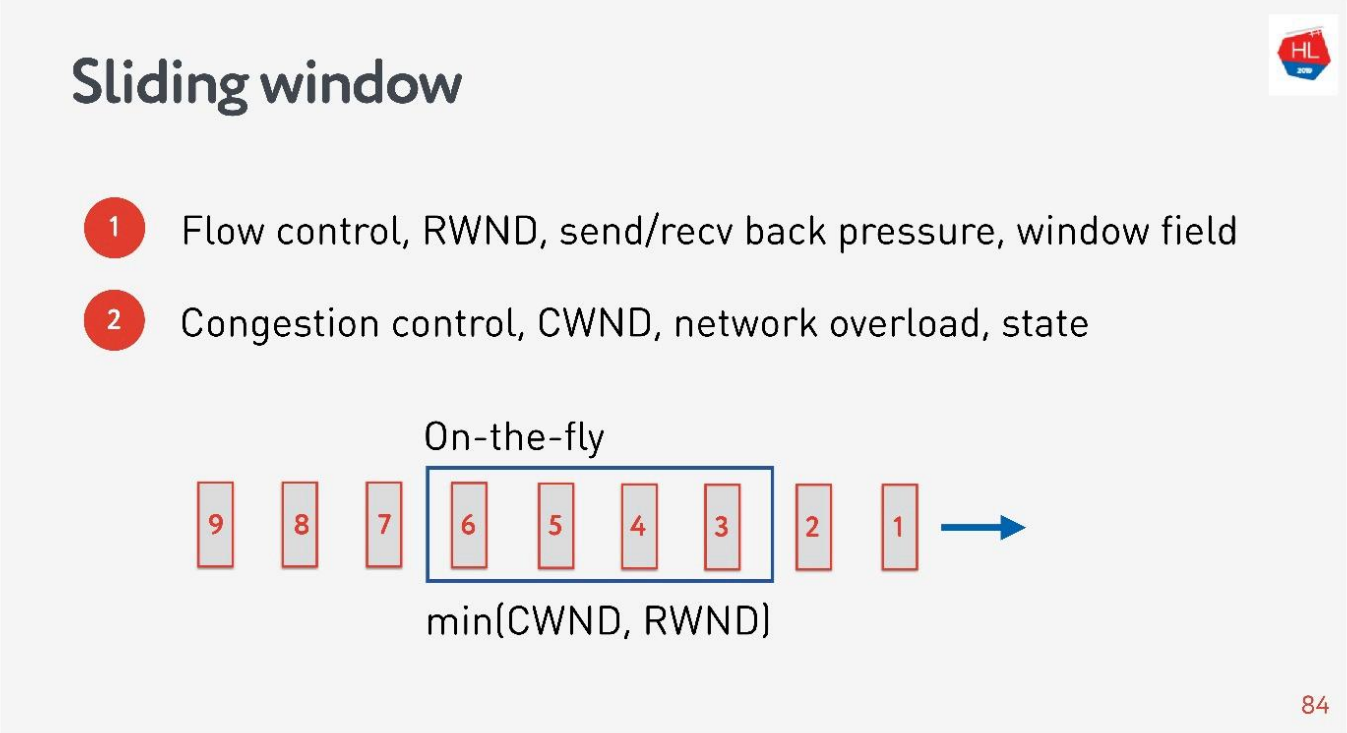



При retransmit, который просто досылает пакеты, мы не должны наблюдать такую проблему.

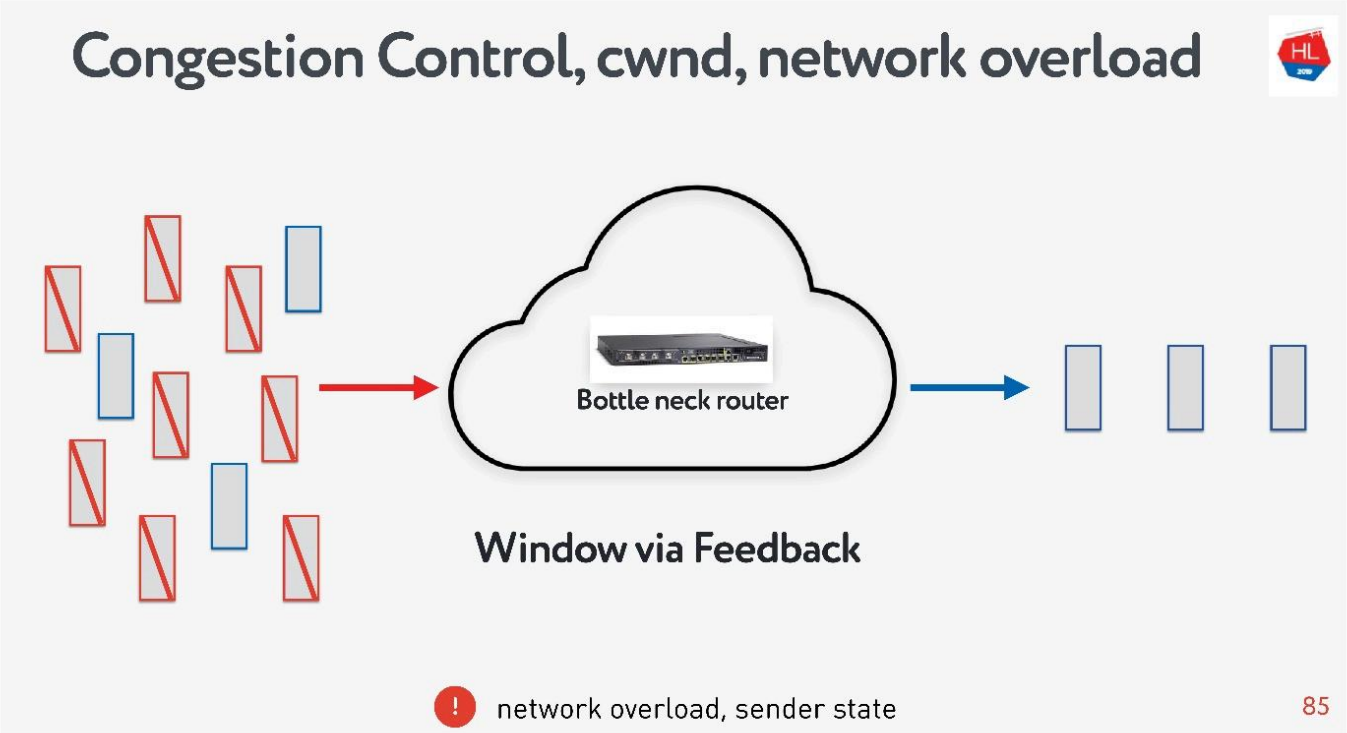
Чтобы разобраться в причинах, нужно понять, что такое Congestion control.

## Congestion control

Его очень часто путают с flow control, поэтому рассмотрим их оба.

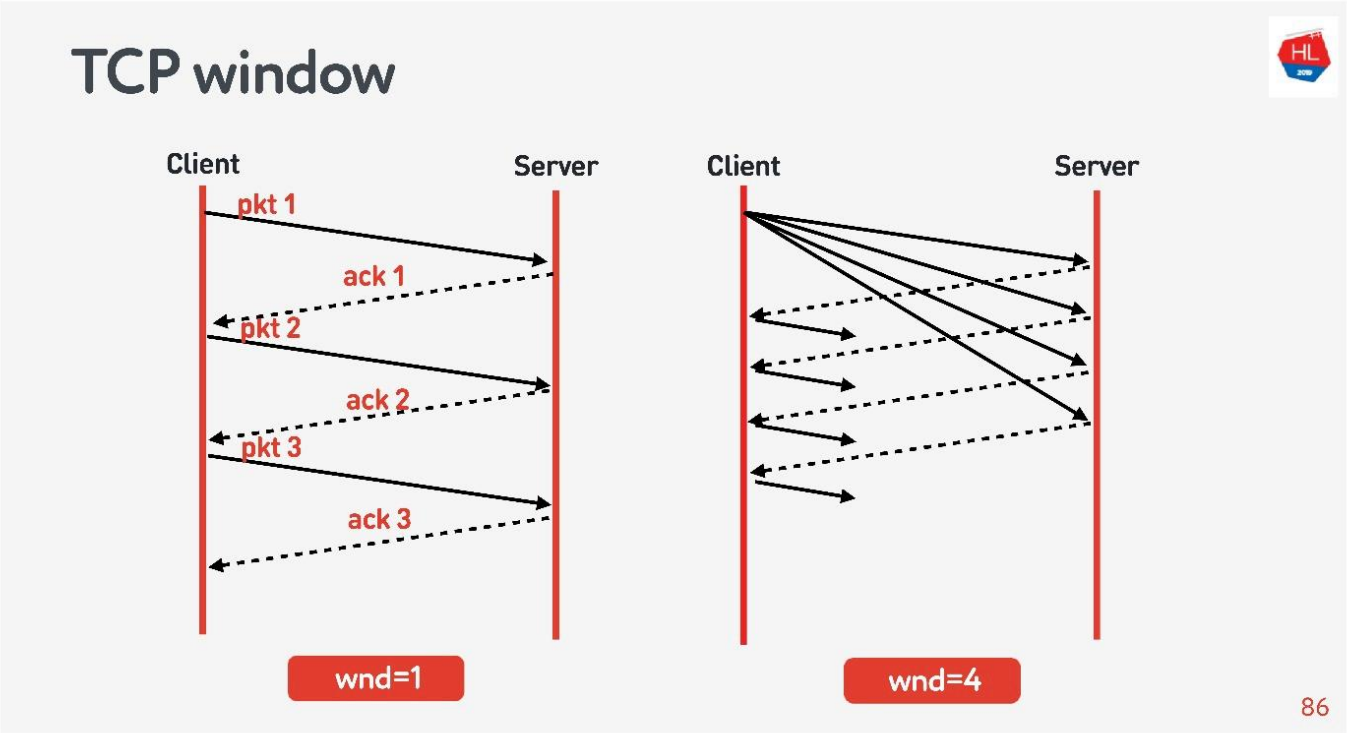



- **Flow control** — это некий механизм защиты от перегрузки. Получатель говорит, на какое количество данных у него реально есть место в буфере, чтобы он был готов их принять. Если передать сверх flow control или recv window, то эти пакеты просто будут выкинуты. Задача flow control — это back pressure от нагрузки, то есть просто кто-то не успевает вычитывать данные.
- У **congestion control** совершенно другая задача. Механизмы схожие, но задача — спасти сеть от перегрузки.




Если перегрузить сеть, то вполне вероятна такая ситуация: посылаете данные, часть пакетов не доходит, посылаете еще больше данных, и все эти данные опять пропадают. За то чтобы лимитировать выдачу данных некоторыми порциями, как раз и отвечает congestion control.

Существует так называемый TCP window.



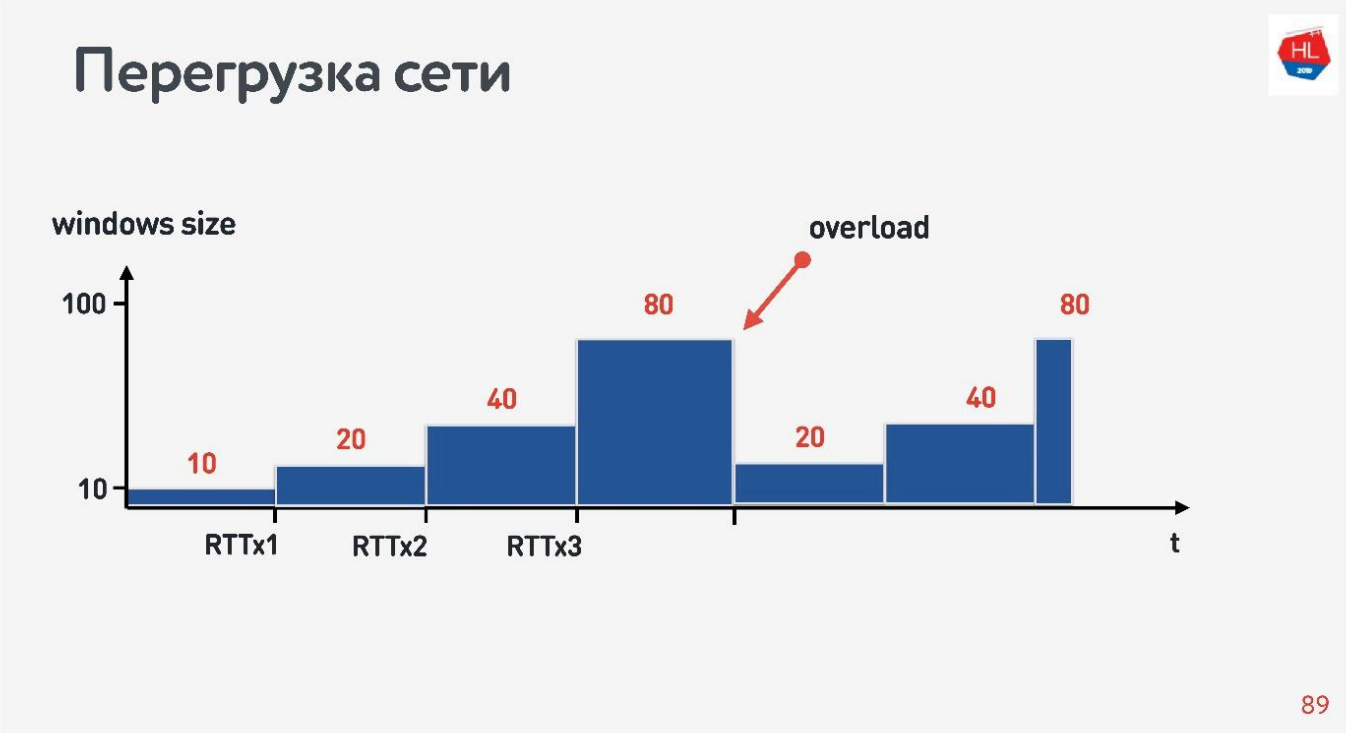

Это некоторый минимум из flow control и congestion control, то есть явно не превышает эти

значения.

Примеры:

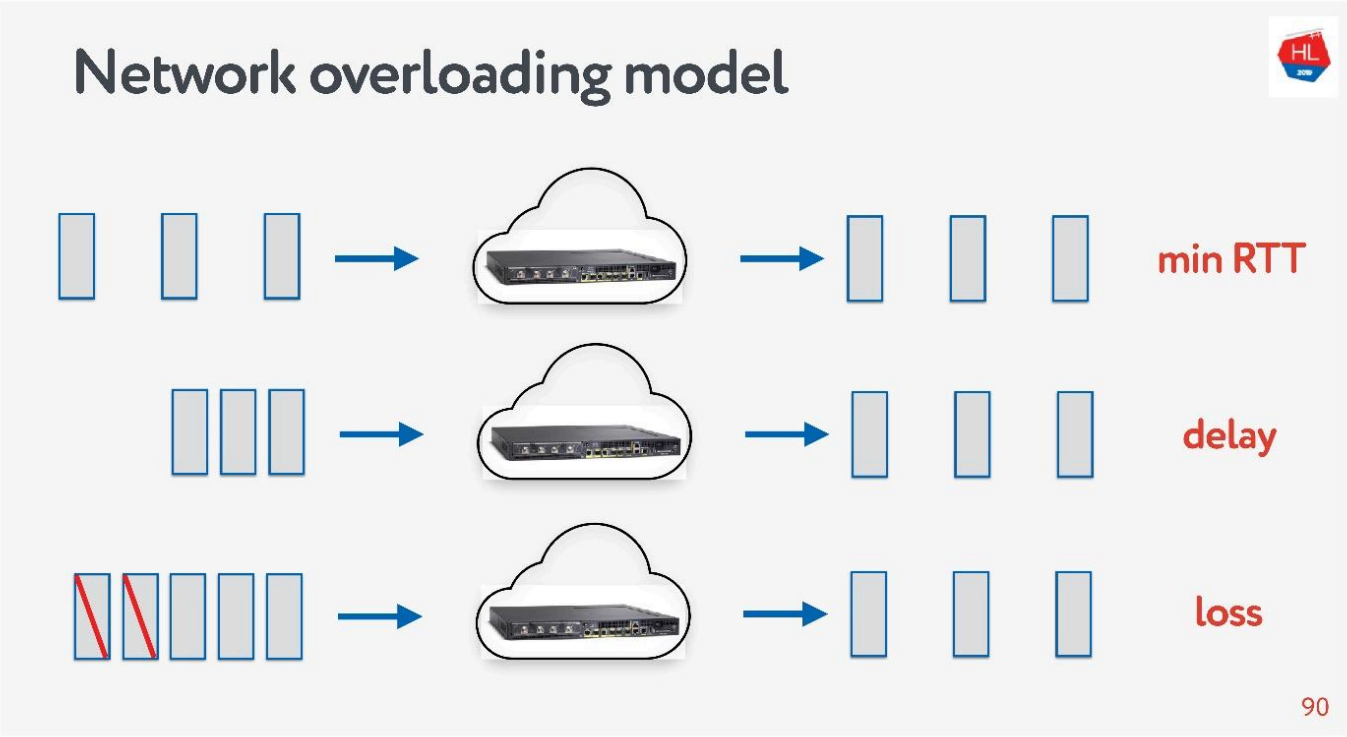
- Если TCP window = 1, то данные передаются как на схеме слева: дожидаемся acknowledgement, отправляем следующий пакет и т.д.
- Если TCP window = 4, то отправляем сразу пачку из четырех пакетов, дожидаемся acknowledgement и дальше работаем.

Когда соединение только стартует, размер окно постепенно увеличивается. Размер initial window в TCP = 10.



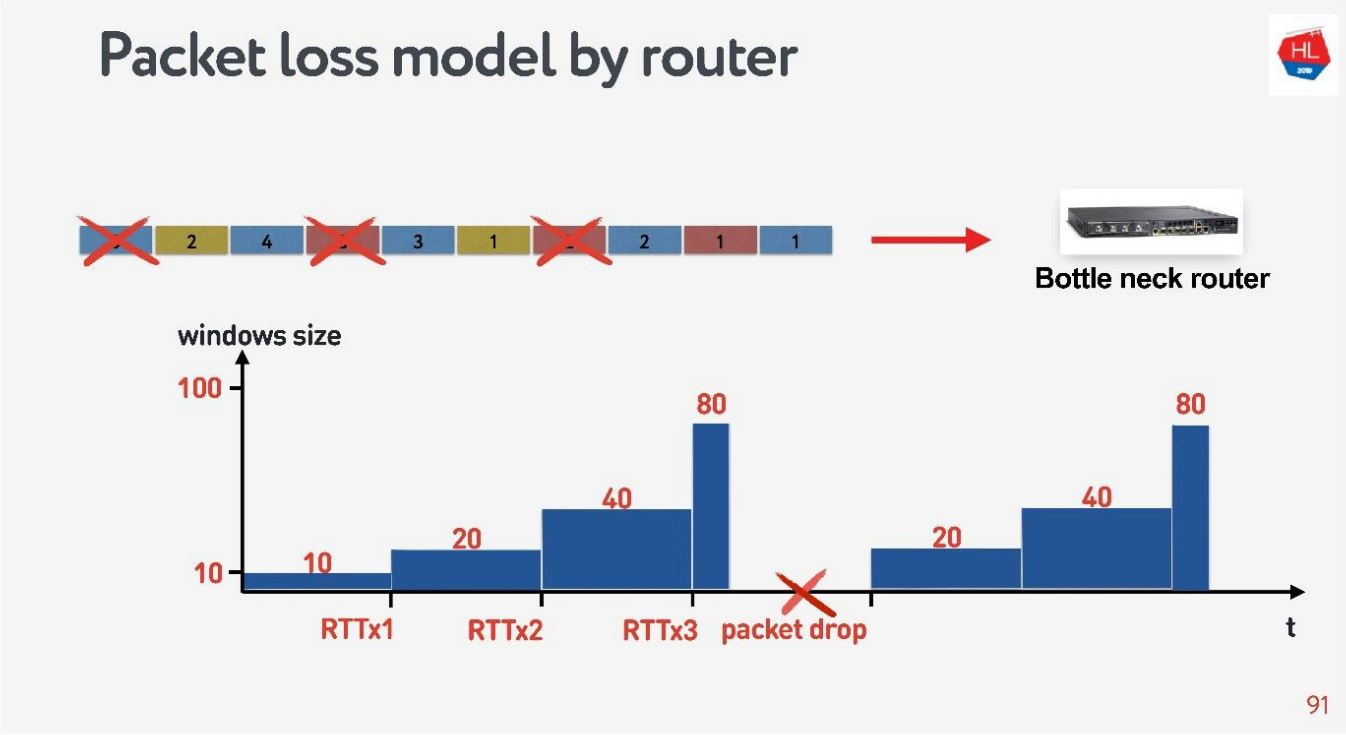

Если происходит перегрузка сети, пропадают пакеты, то окно обратно сужается и начинает разгоняться заново.

Как при этом выглядит сеть?

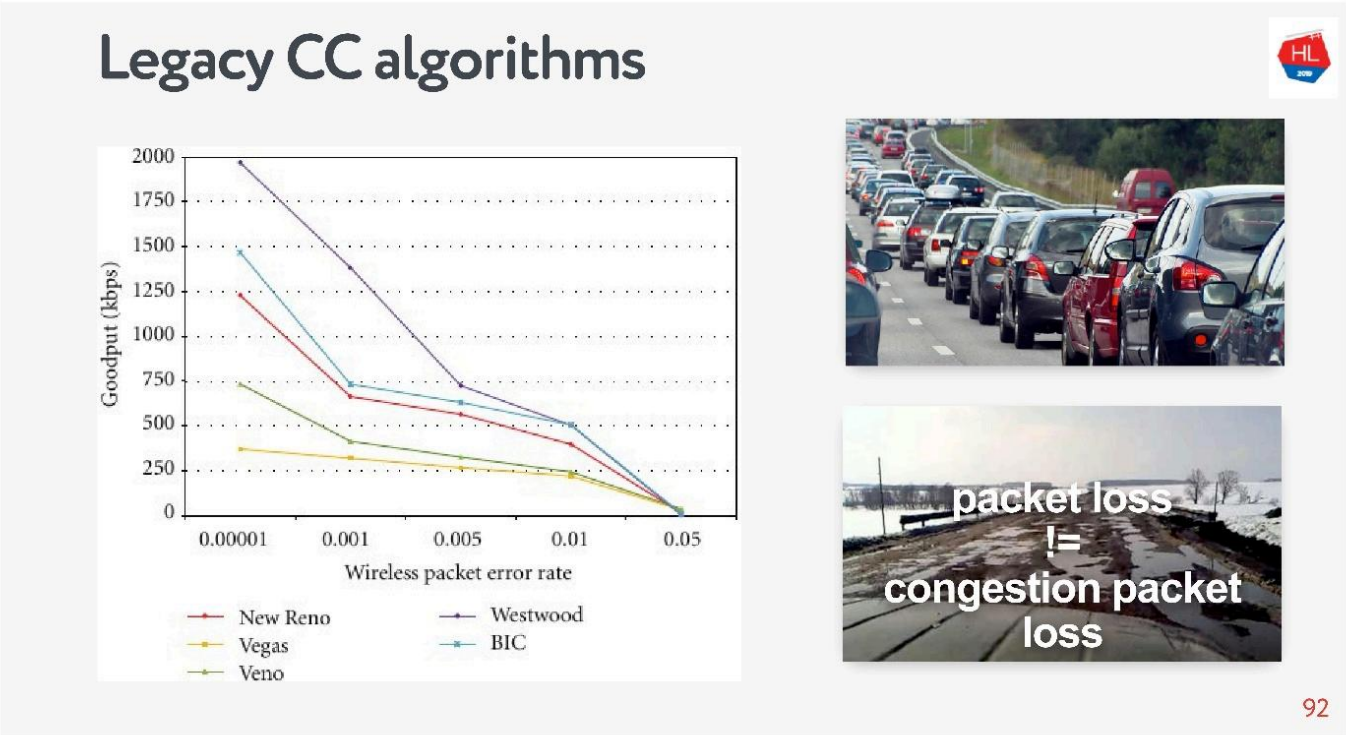



- На верхней схеме сеть, в которой все хорошо. Пакеты отправляются с заданной частотой, с такой же частотой возвращаются подтверждения.
- Во второй строке начинается перегруз сети: пакеты идут чаще, acknowledgements приходят с задержкой.
- Данные копятся в буферах на маршрутизаторах и других устройствах и в какой-то момент начинают пропускать пакеты, acknowledgements на эти пакеты не приходят (нижняя схема).

С точки зрения маршрутизатора это выглядит так.



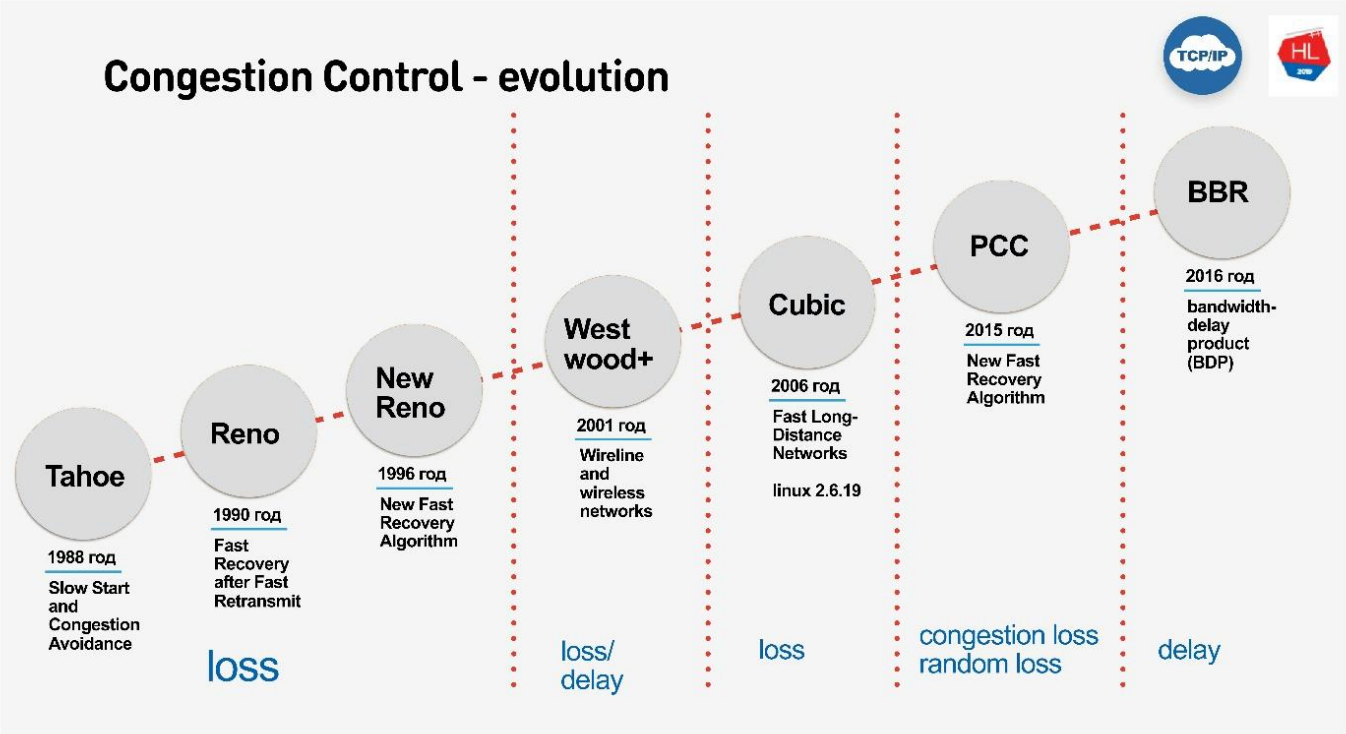

Маршрутизатор немножко умный, он не дожидается перегрузки, и сразу дропает. У него есть механизм тикетов: он выдает тикет на отправку, если канал освободится и т.д. Суть механизма в том, что он дропает пакеты чуть раньше. Тогда срабатывает congestion control, схлопывает TCP window, нагрузка на маршрутизатор падает, и все продолжает работать.




Так работали старые механизмы congestion control, которые были уверены, что сеть — это картинка сверху. На самом деле не любой packet loss — следствие того, что сеть перегружена. У нас есть сети как на нижней картинке, про которые говорят, что в них потеря

пакетов ничего не значит — это просто такая сеть, потому что она беспроводная.

Понятно, что TCP развивался, адаптировался, и первый congestion control оперировал только loss-функцией. После этого появились congestion control на loss delay, то есть и на потери, и на задержки.

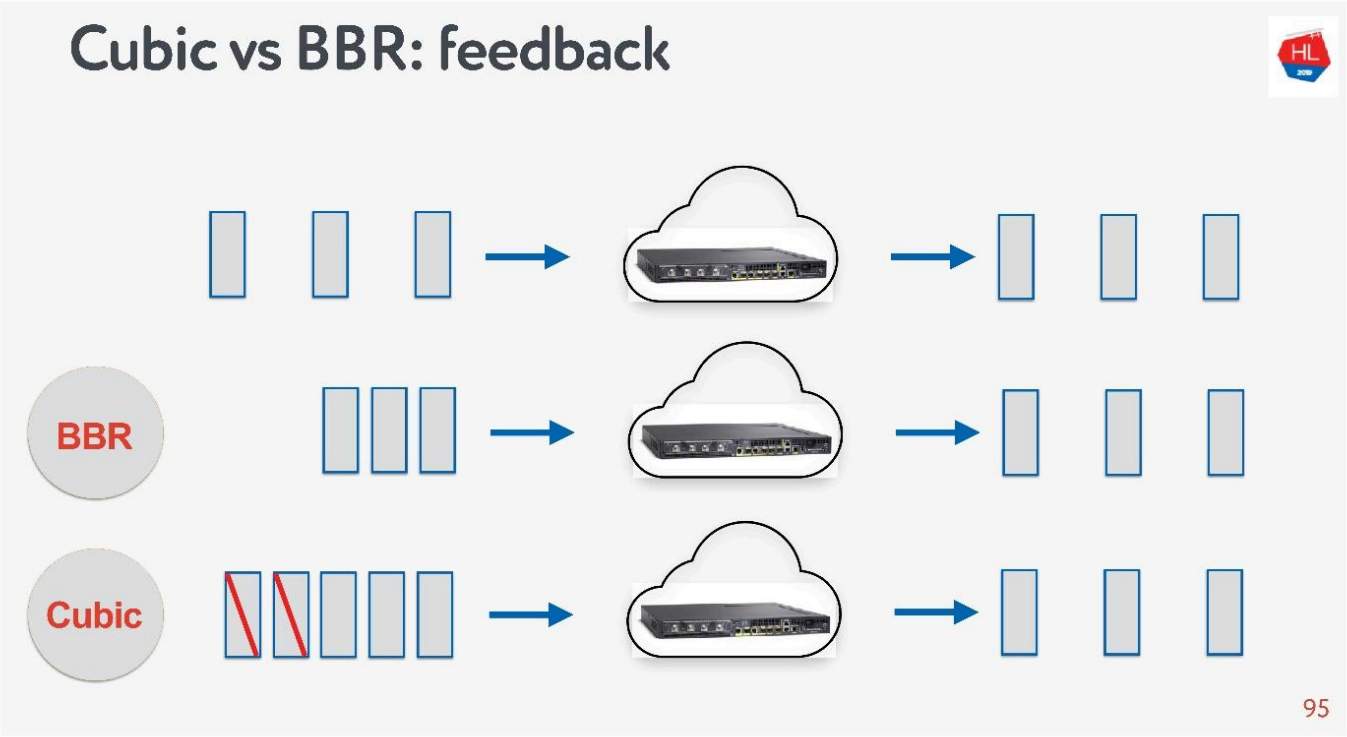



Рассмотрим:

- **Cubic** — дефолтный Congestion Control с Linux 2.6. Именно он используется чаще всего и работает примитивно: потерял пакет — схлопнул окно.
- **BBR** — более сложный Congestion Control, который придумали в Google в 2016 году. Учитывает размер буфера.

## BBR Congestion Control

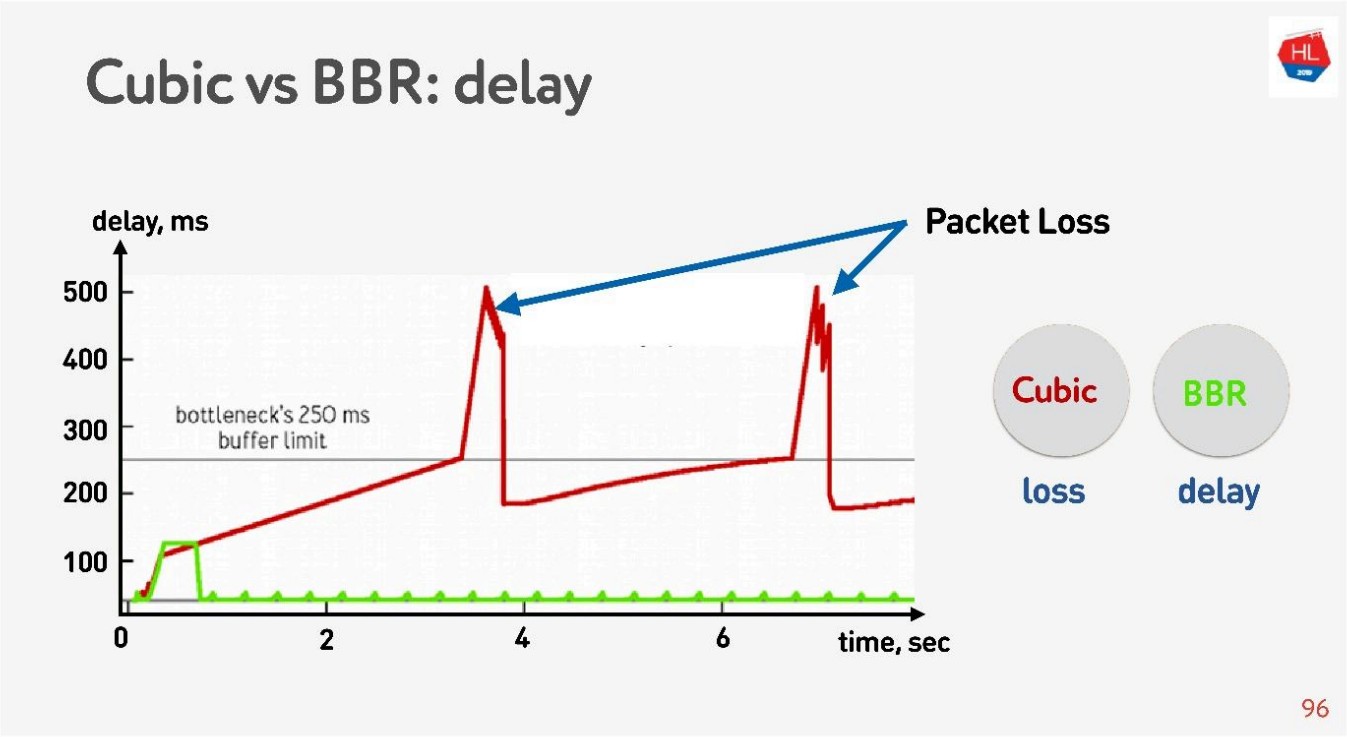
Посмотрим на Cubic и BBR по методам feedback.




На схеме сверху нормальный маршрутизатор и маршрутизатор, у которого очередь начинает копиться — каждый следующий acknowledgement приходит всё дольше и дольше относительно отправки. В этом случае:

- BBR понимает, что идет переполнение буфера, и пытается схлопнуть окно, уменьшить нагрузку на маршрутизатор.
- Cubic дожидается потери пакета и после этого схлопывает окно.

Ниже график зависимости задержки от времени соединения, из которого видно, что происходит на разных Congestion Control.

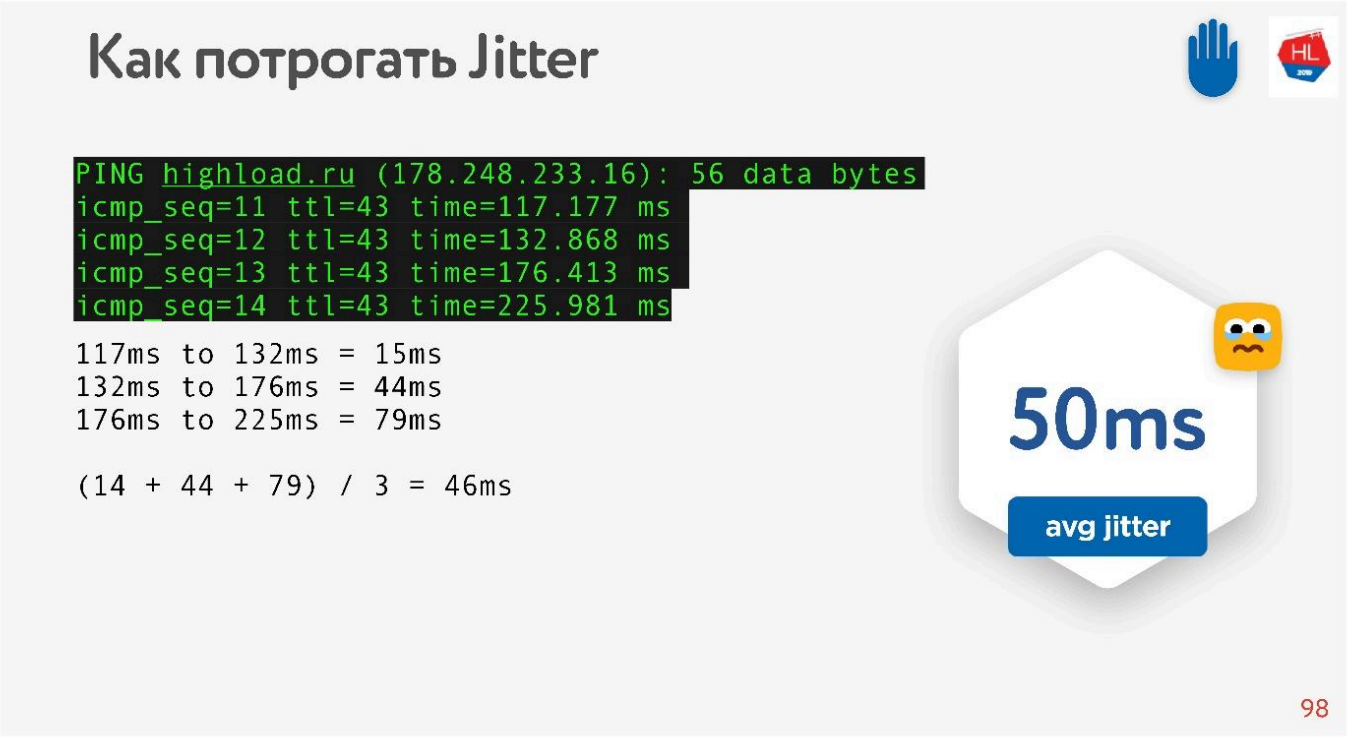



BBR вначале прощупывает время round-trip, отправляет больше и больше пакетов, потом понимает, что буфер забивается, и выходит на режим работы с минимальной задержкой.

Cubic работает агрессивно — он переполняет целиком буфер, и, когда буфер переполняется и случается packet loss, то cubic уменьшает окно.

Кажется, что с помощью BBR можно было бы решить все проблемы, но в сетях существует **jitter** — пакеты иногда задерживаются, иногда группируются пачками. Вы их отправляете с определенной частотой, а они приходят группами. Еще хуже, когда вы получаете acknowledgements обратно на эти пакеты, и они тоже как-то «jitter'ятся».

Так как я обещал, что все можно будет потрогать руками, то пингуем, например, сайт HighLoad++, смотрим ping и считаем jitter между пакетами.

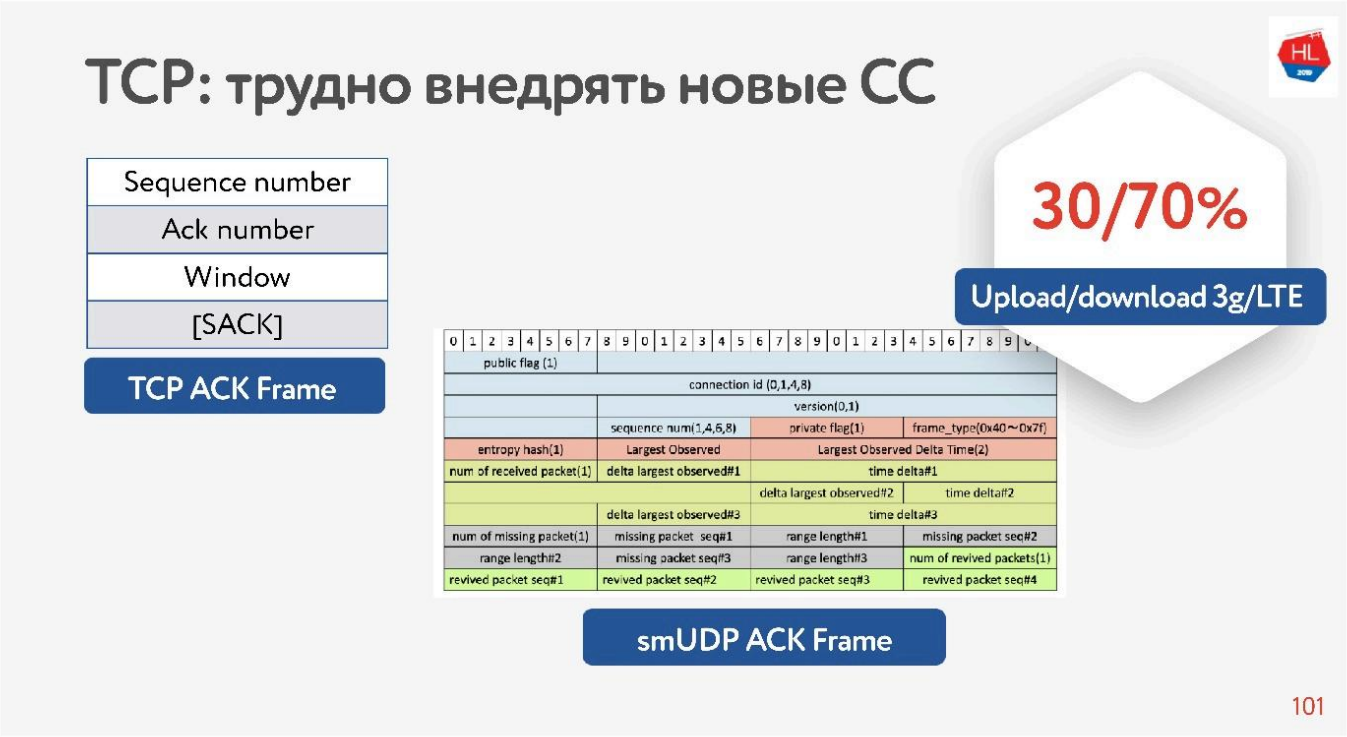



Видно, что пакеты приходят неравномерно, средний jitter порядка 50 мс. Естественно, BBR может при этом ошибиться.

BBR хорош тем, что различает: реальный congestion loss, потерю пакетов в виду переполнения буферов устройств, и random loss из-за плохой беспроводной сети. Но плохо работает в случае высокого jitter. Как можно ему помочь?

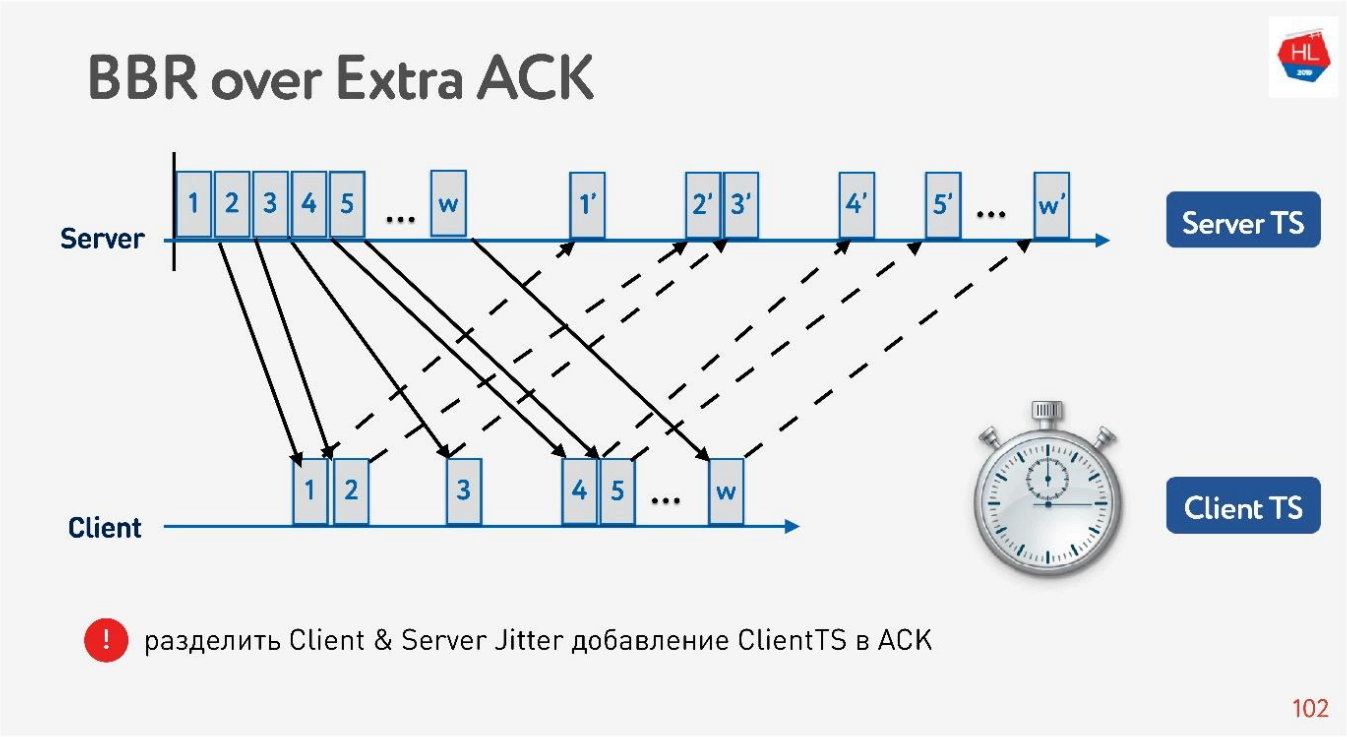
## Как сделать Congestion control лучше

На самом деле у TCP в acknowledgement достаточно мало информации, в ней есть только то, какие пакеты он видел. Есть еще selective acknowledgement, в котором говорится, какие пакеты подтверждены, какие еще не дошли. Но и этой информации недостаточно.

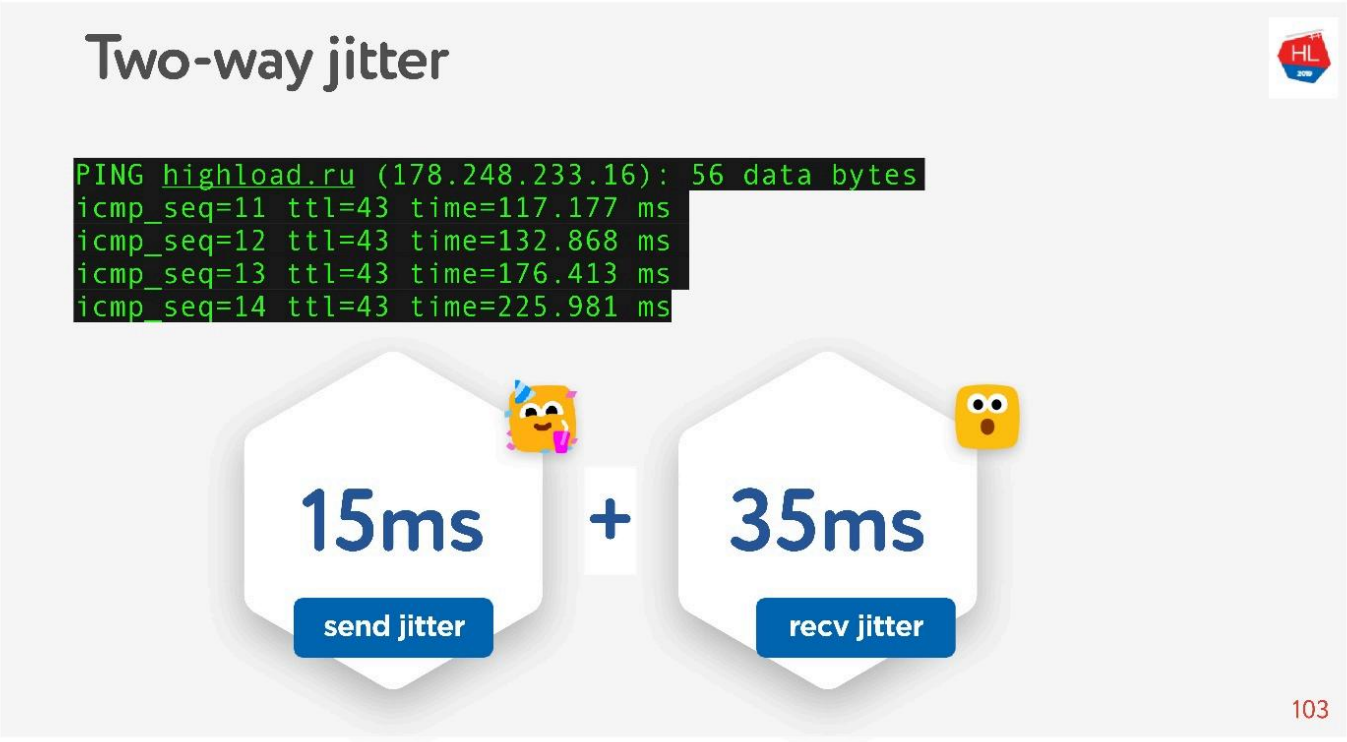



Если вы имеете возможность раздуть acknowledgement, то можете еще сохранить все времена — не только отправки этих пакетов, но и прихода их на клиент. То есть, по сути, на сервере собрать jitter клиента.

Почему вообще эффективно раздувать acknowledgement? Потому что мобильные сети асимметричны. Например, обычно у 3G или LTE 70% пропускной способности выделяется на скачивание данных и 30% — на upload. Передатчик переключается: upload — download, upload — download, и вы на это никак не влияете. Если вы ничего не выгружаете, то он просто простаивает. Поэтому если у вас есть какие-то интересные идеи, увеличивайте acknowledgement, не стесняйтесь — это не проблема.




Пример того, как можно с помощью acknowledgement поделить jitter на отправку и jitter на прием, и отслеживать их отдельно. Тогда мы становимся более гибкими, и понимаем, когда произошел congestion loss, а когда random loss. Например, можно понять, сколько jitter в каждую сторону, и более точно настроить окно.

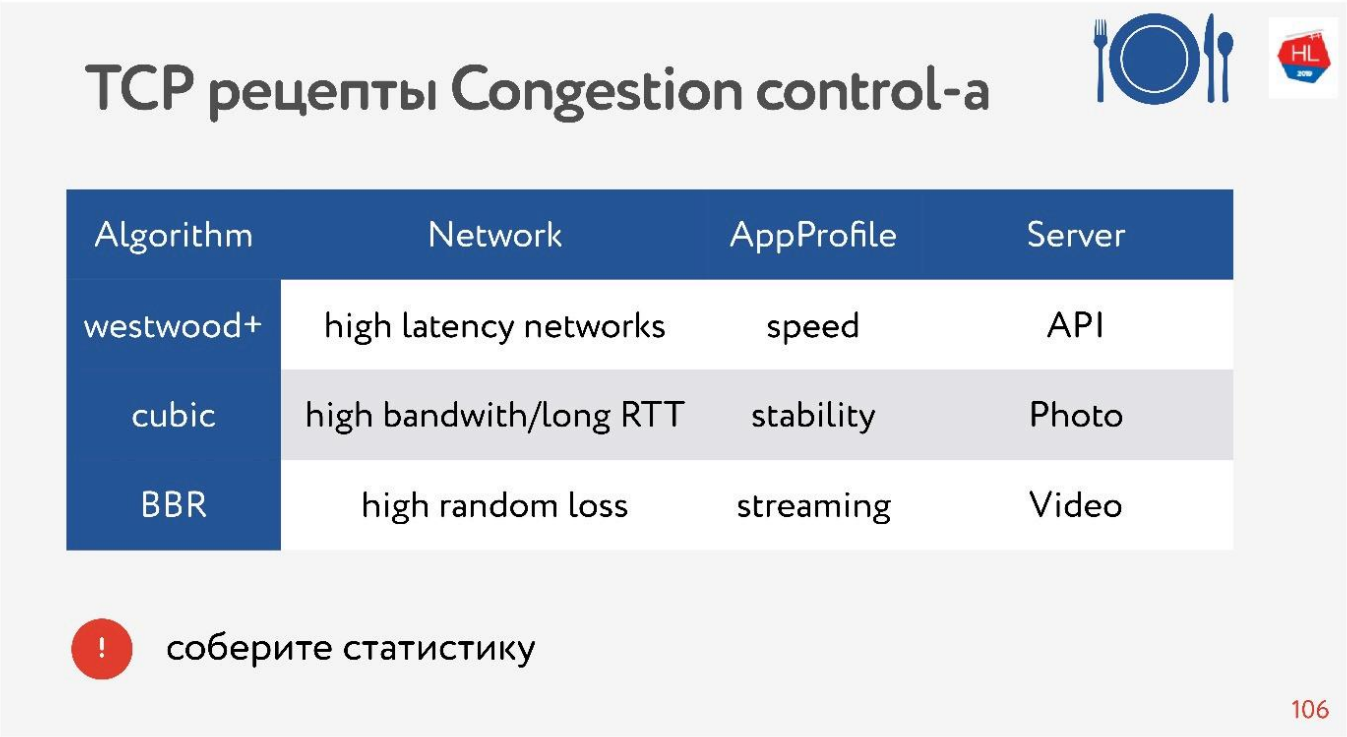



## Какой Congestion control выбрать

Одноклассники — большая сеть, в которой много разного трафика: видео, API, картинки. И

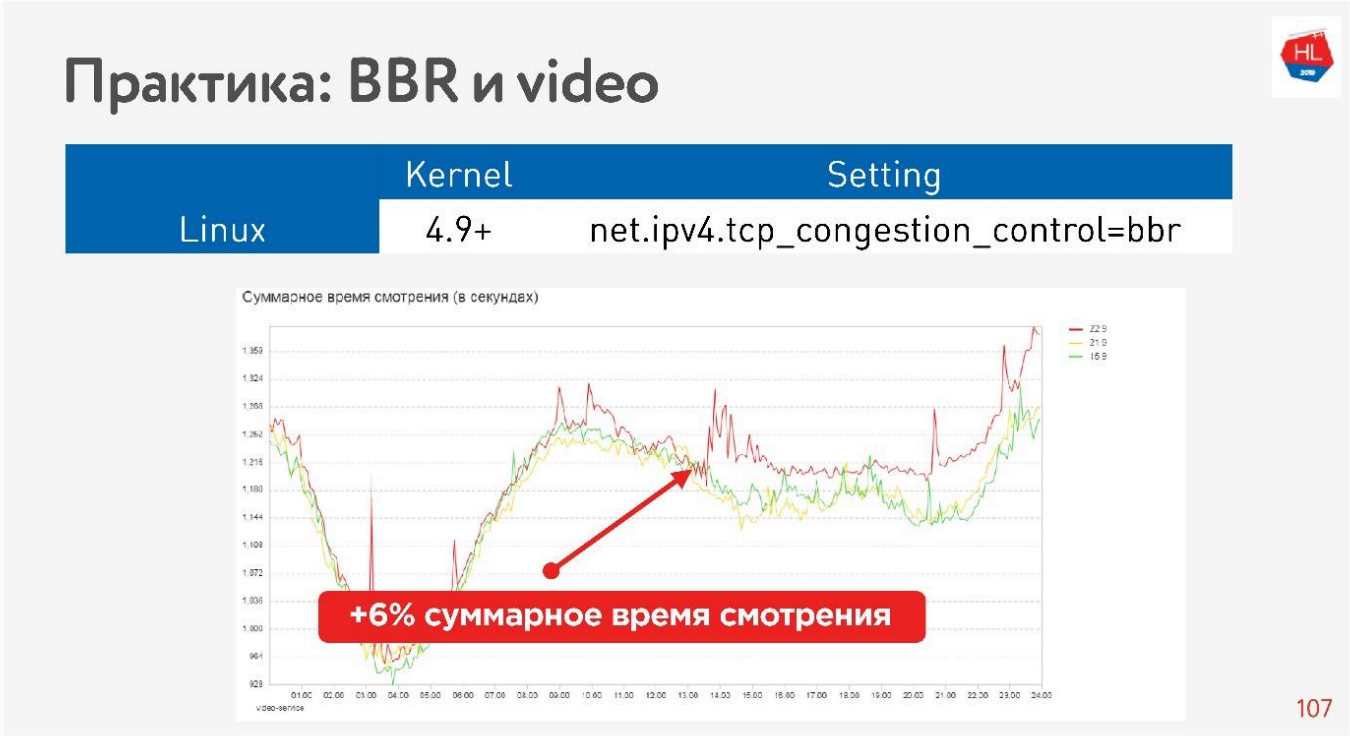
есть статистика, какие congestion control для чего лучше выбрать.

BBR всегда эффективен для видео, потому что уменьшает задержки. В остальных случаях обычно используется Cubic — он хорош для фотографий. Но есть другие варианты.



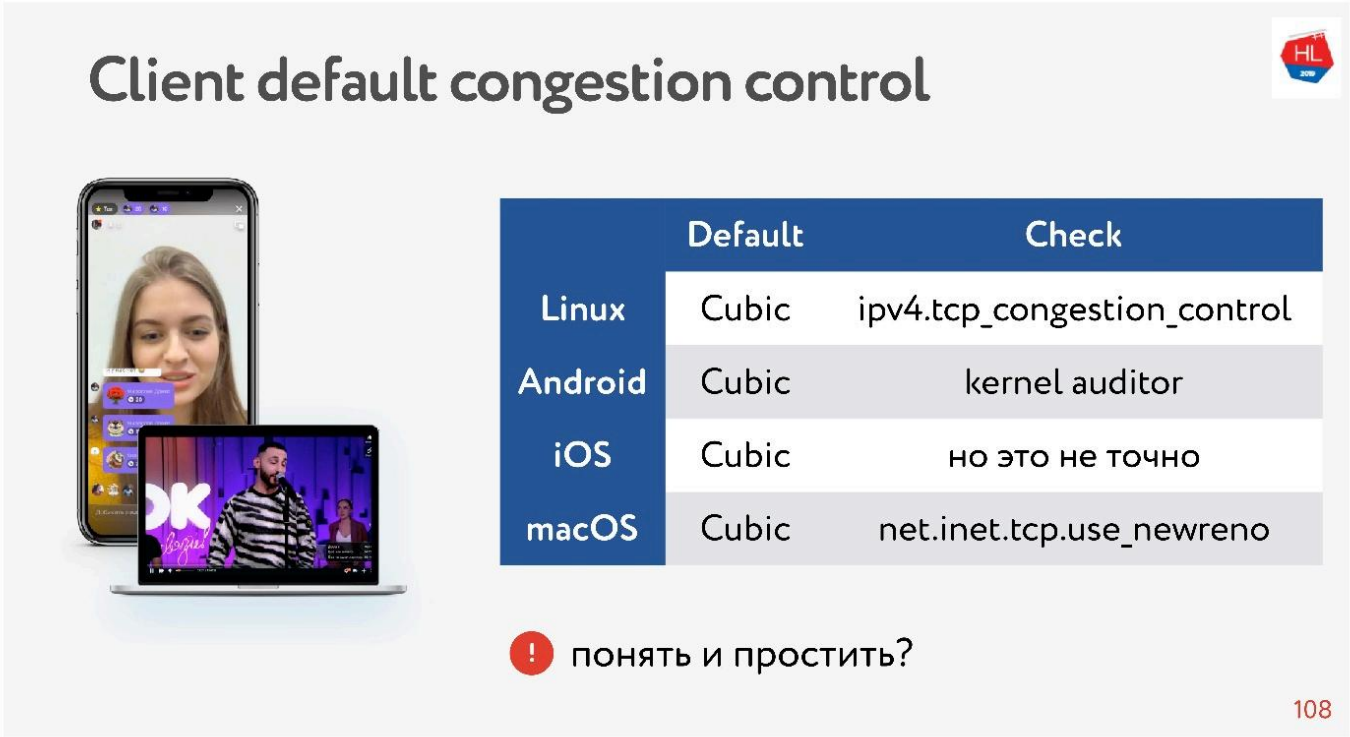

Есть десятки разных вариантов congestion control. Для того чтобы выбрать лучший, можно собрать статистику по клиенту и для разного типа профиля нагрузки попробовать тот или иной congestion control.

Например, это эффект от запуска BBR на видео.




Нам удалось серьезно увеличить глубину просмотра. Google говорит, что у них примерно на 10% уменьшается количество буферизации в плеере при использовании BBR.

Здорово, но что у нас на клиентах?




Клиенты немножко заторможенные, у них у всех Cubic, и вы на это не можете повлиять. Но ничего страшного, иногда можно параллелить данные, и будет хорошо.

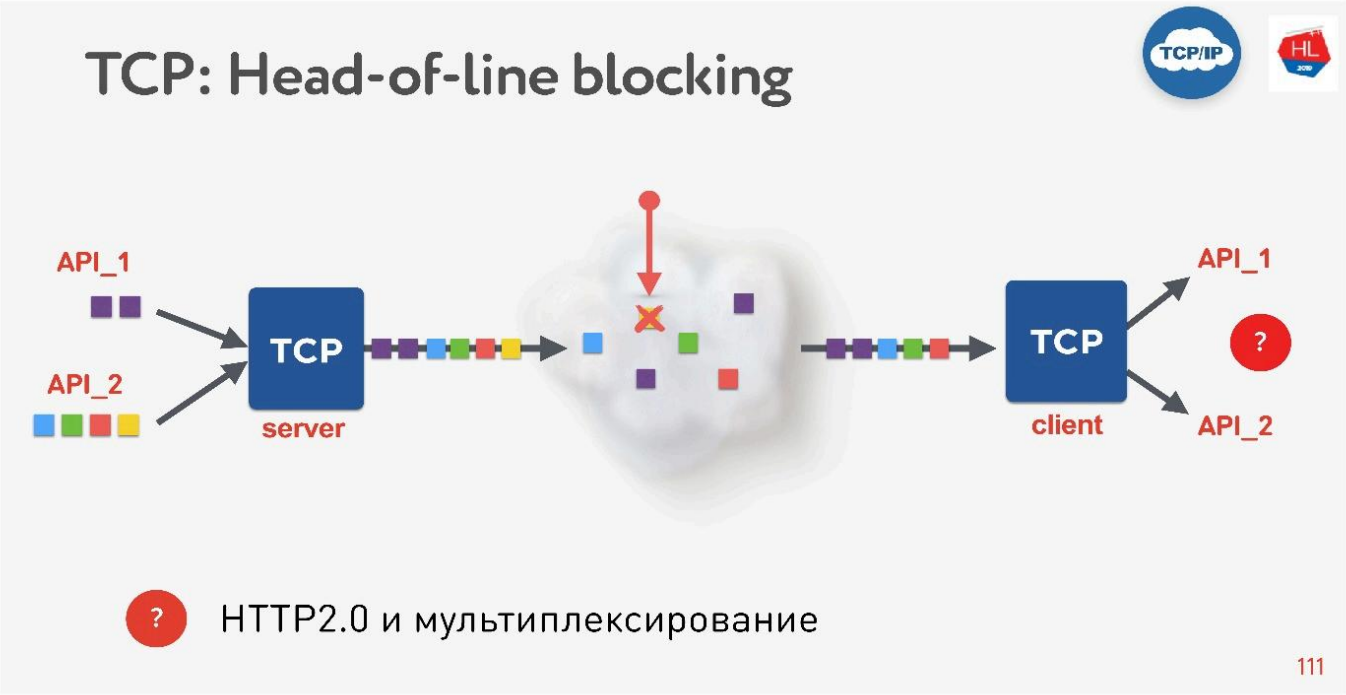
**Выводы про congestion control:**

- Для видео всегда хорош BBR.
- В остальных случаях, если мы используем свой UDP-протокол, можно взять congestion control с собой.
- С точки зрения TCP можно использовать только congestion control, который есть в ядре. Если хотите реализовать свой congestion control в ядро, нужно обязательно соответствовать спецификации TCP. Невозможно раздуть acknowledgement, сделать изменения, потому что просто их нет на клиенте.

Если вы делаете свой UDP-протокол, у вас гораздо больше свободы с точки зрения congestion control.

## Мультиплексирование и приоритизация

Это новый тренд, все сейчас этим занимаются. Какие здесь есть проблемы? Если мы используем TCP, наверняка все (или почти все) знают ситуацию head-of-line blocking.

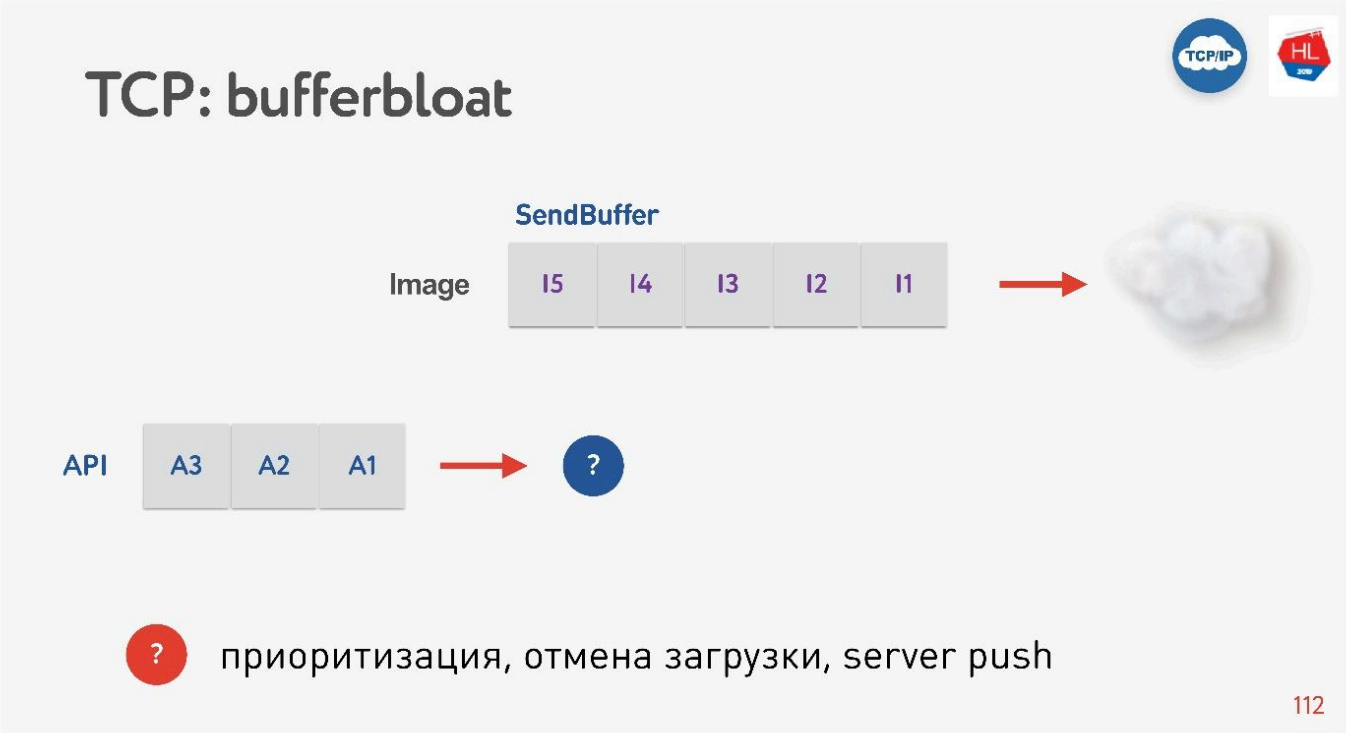



Есть несколько запросов, которые мультиплексируются через одно TCP-соединение. Мы их отправили в сеть, но какой-то пакет пропал. TCP-соединение будет этот пакет ретрансмитить, он заретрансмитится за время, близкое к RTT или больше. В это время мы ничего получить не сможем, хотя в TCP-буфере находятся данные от другого запроса,

полностью готовые к тому, чтобы их можно было забрать.

> Получается, что мультиплексирование поверх TCP, если вы используете HTTP 2.0, не всегда эффективно в плохих сетях.

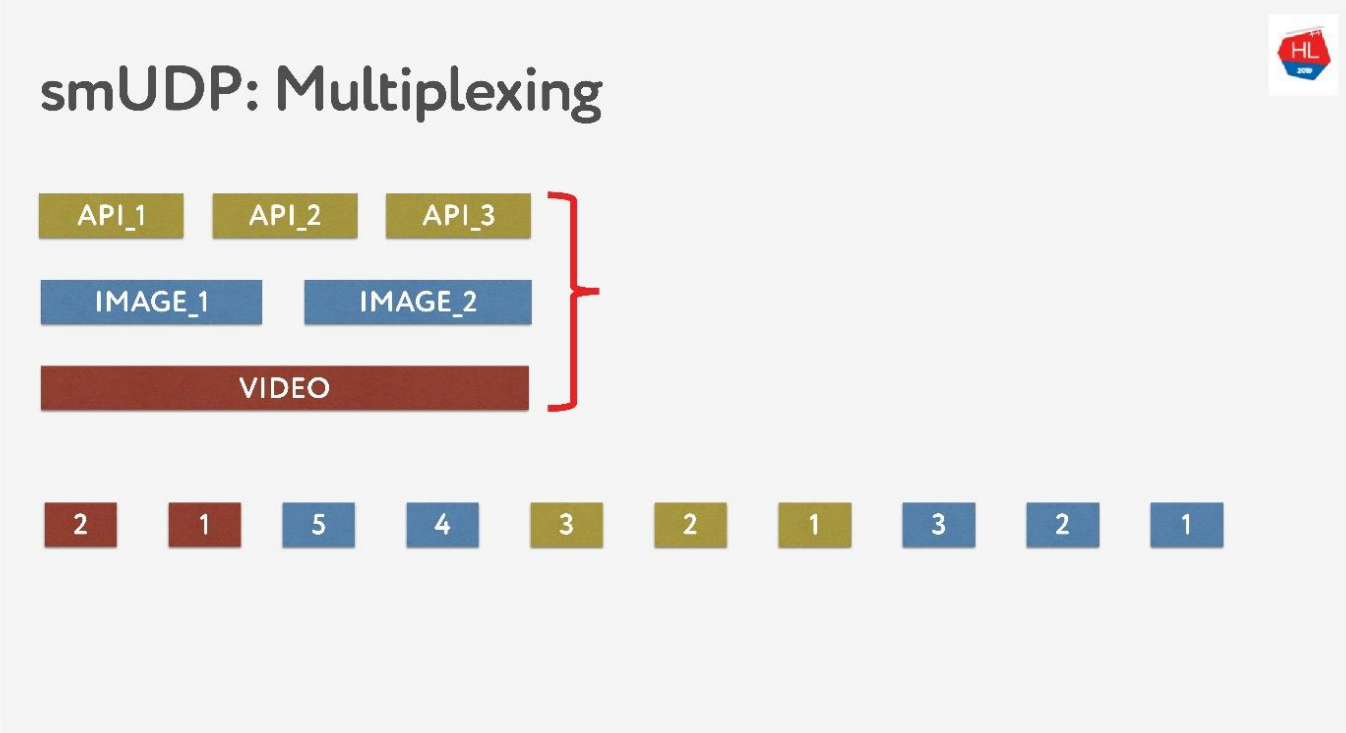
Следующая проблема — это распухание буфера.



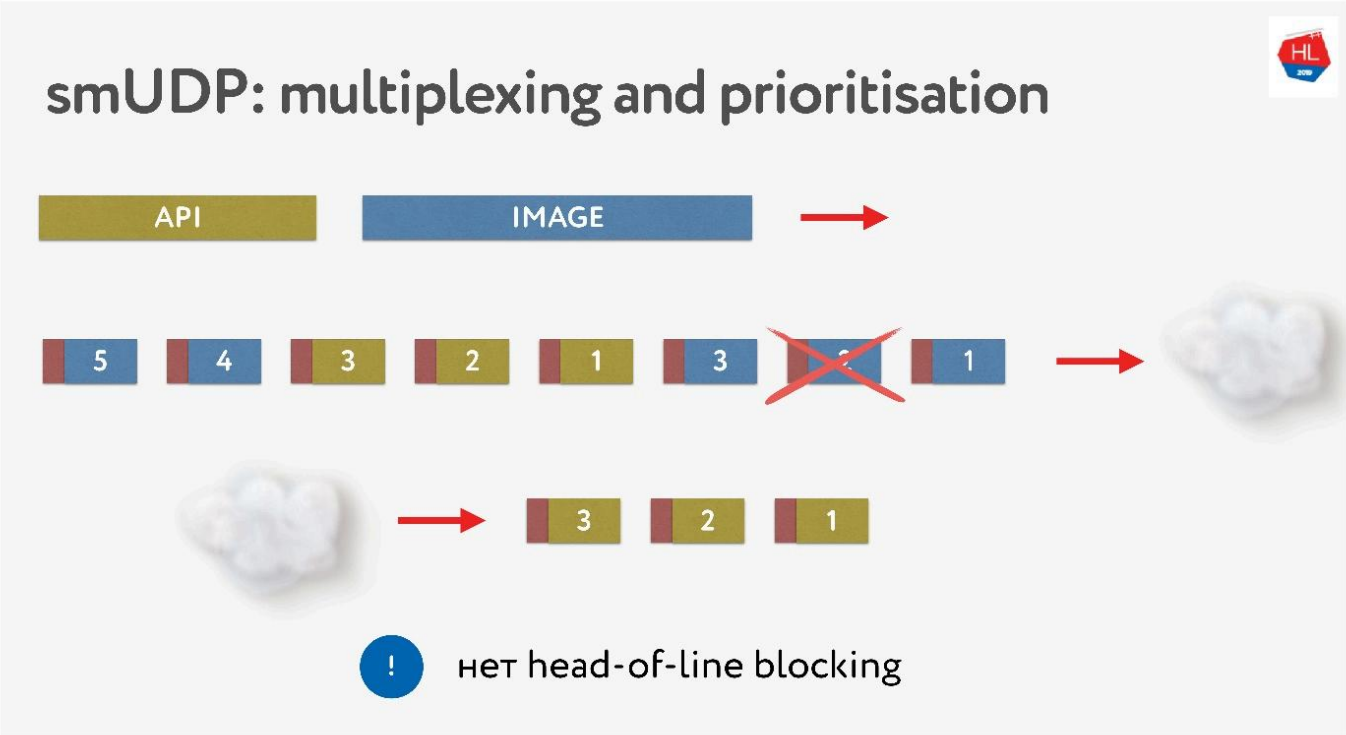

Когда картинка отправляется клиенту, увеличивается буфер. Мы его долго отправляем, а потом появляется API-запрос, и он никак не может быть приоритизирован. В таких случаях не работает TCP-приоритизация.

Таким образом, если случается потеря пакетов, есть head-of-Line blocking, а когда у клиента переменный битрейт (а у мобильных клиентов это бывает часто), то появляется эффект bufferbloat. В итоге не работает ни мультиплексирование, ни приоритизация, ни server push, ни все остальное, потому что у нас или забиты буферы, или клиент что-то ожидает.

Если мы делаем свое мультиплексирование, то можем поместить туда различные данные.



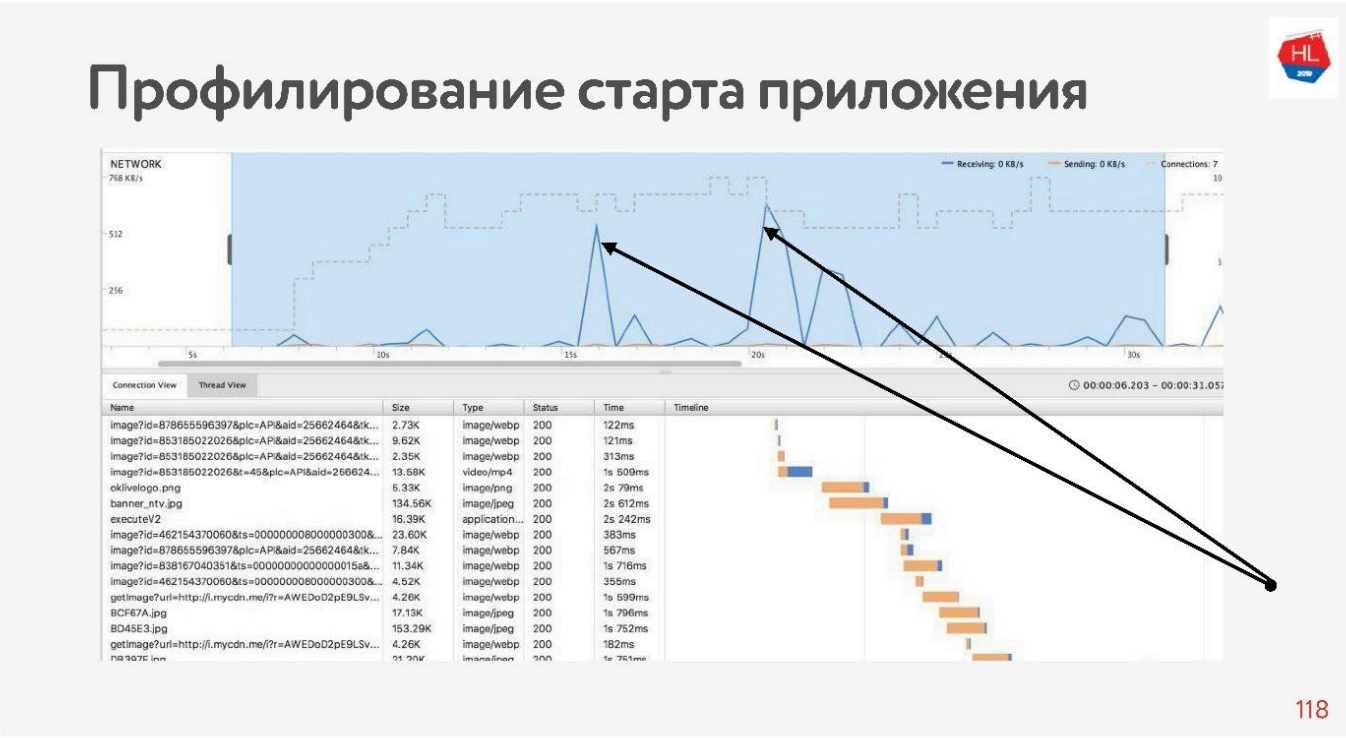

Это нетрудно, просто складываем в буфер пакеты с номерами. On-the-fly — то, что уже было отправлено, не трогаем, а то, что еще не отправлено, можно переставлять. Выглядит это так.



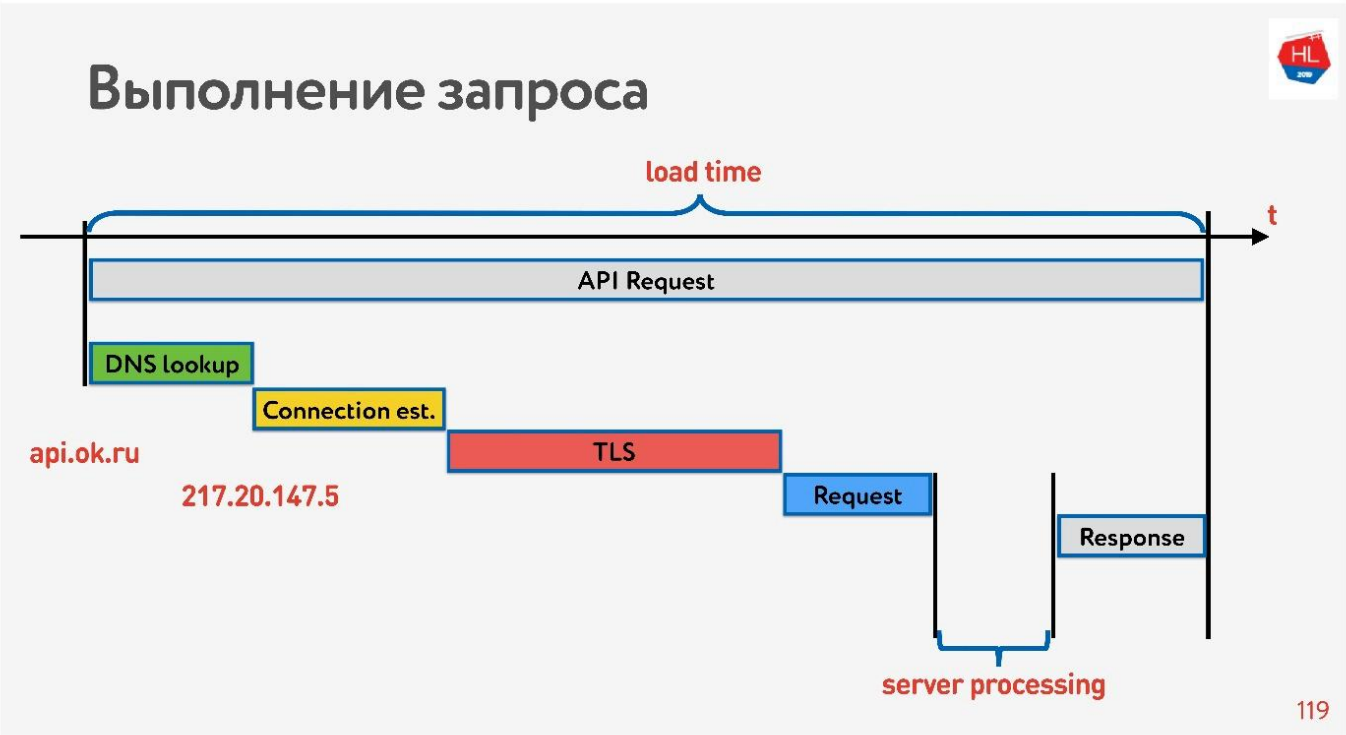

Отправили картинки, разбили на пакеты, пришел приоритетный API-запрос: его вставили, дослали картинку. Даже если пропал пакет, мы из буфера можем достать готовый API-запрос, он высокоприоритетный и быстро дойдет до клиента. В TCP по определению при стриминговой передаче данных такое невозможно.

## Установка соединения

Если попрофилировать наше приложение, то мы увидим, что большую часть времени на старте приложения сеть простаивает, потому что сначала устанавливается соединение до API, потом мы получаем данные, потом устанавливается соединение до картинок, скачиваются эти данные и т.д. Так всегда и происходит — сеть утилизируется пиками.

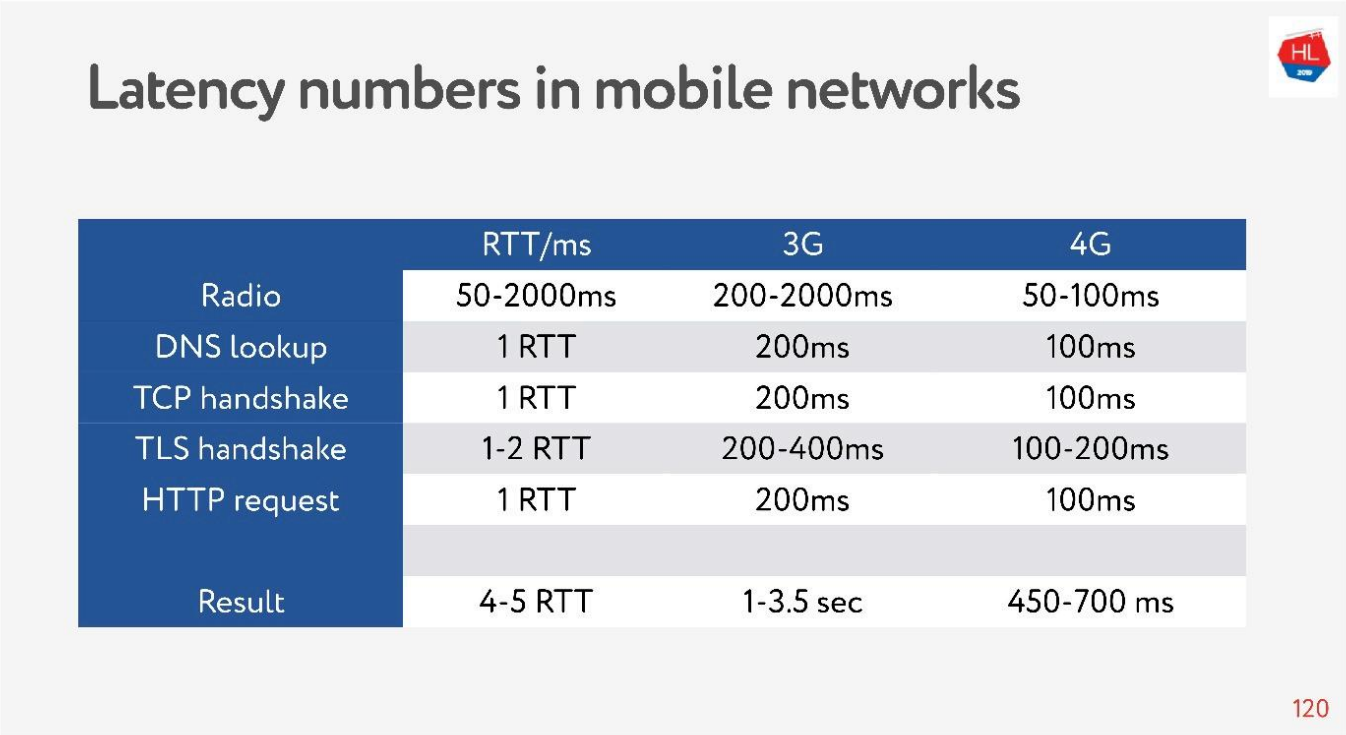



Чтобы с этим разобраться, посмотрим, как устанавливается соединение.



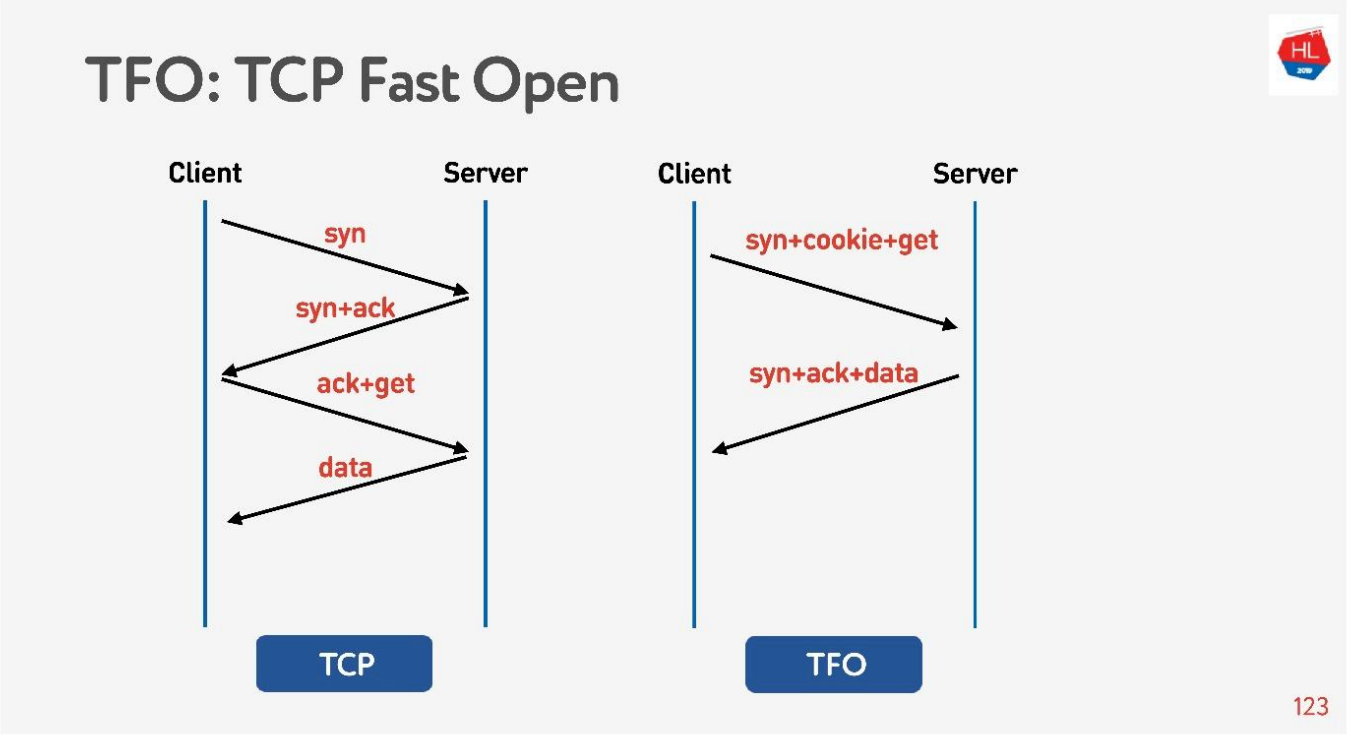
Первое — это resolve DNS — с этим мы ничего сделать не можем. Дальше установка TCP-соединения, установка безопасного соединения, потом выполнение запроса и получение ответа. Самое интересное, что часть работы, которую выполняет сервер, отвечая на запрос, обычно занимает меньше времени, чем установка соединения.

Сейчас очень модно измерять latency numbers для памяти, для дисков, еще для чего-то. Можно их для сети 3G, 4G измерить и увидеть, сколько займет в худшем случае установка соединения по TCP с TLS.

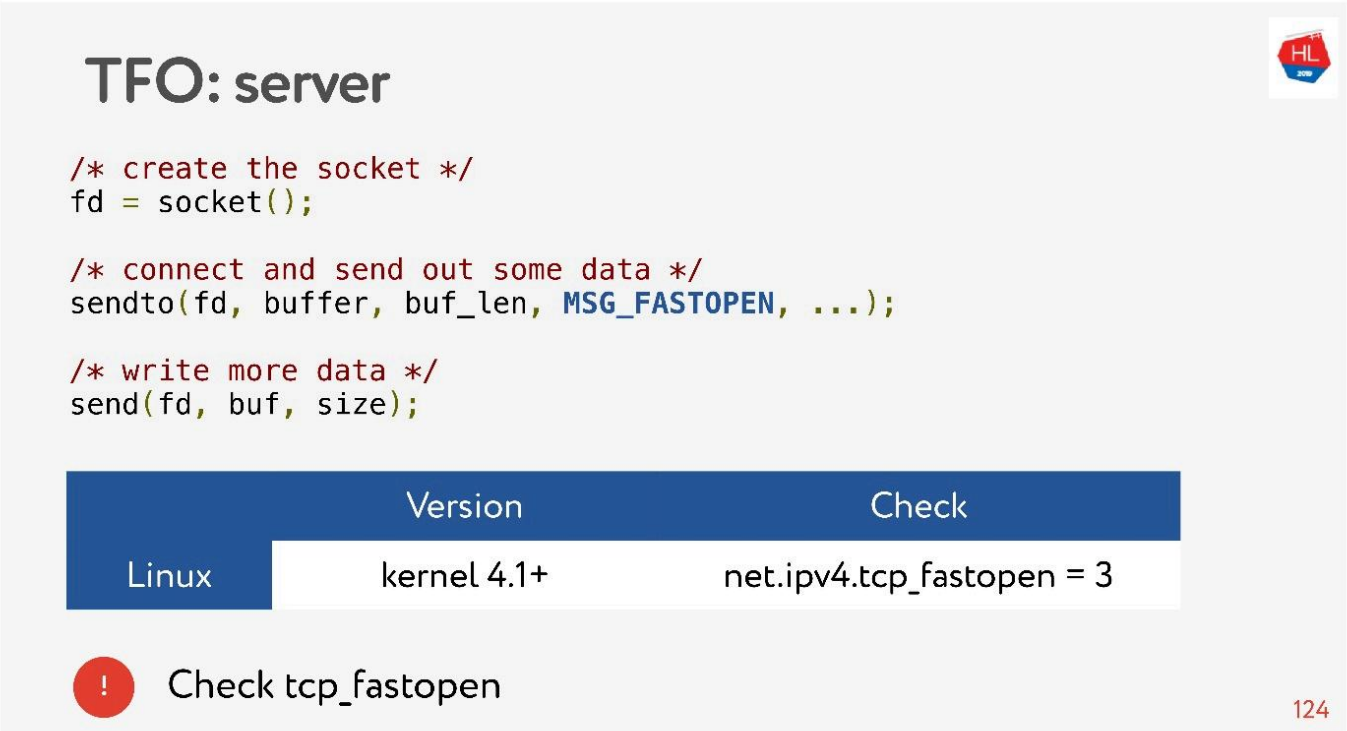



И это могут быть секунды! Даже на 4G до 700 мс –тоже существенно. Но TCP не мог так просто все это время жить.

В установке соединения базовый алгоритм **TCP 3-way handshake**. Делаете syn, syn + ack, подправляете уже потом запрос (слева на схеме).




Есть **TCP Fast Open** (справа). Если вы с этим сервером уже хэндшейкились, есть cookie, можно сразу за zero-RTT отправить свой запрос. Чтобы этим воспользоваться, нужно создать socket, сделать sendto() первых данных, сказать, что вы хотите FASTOPEN.

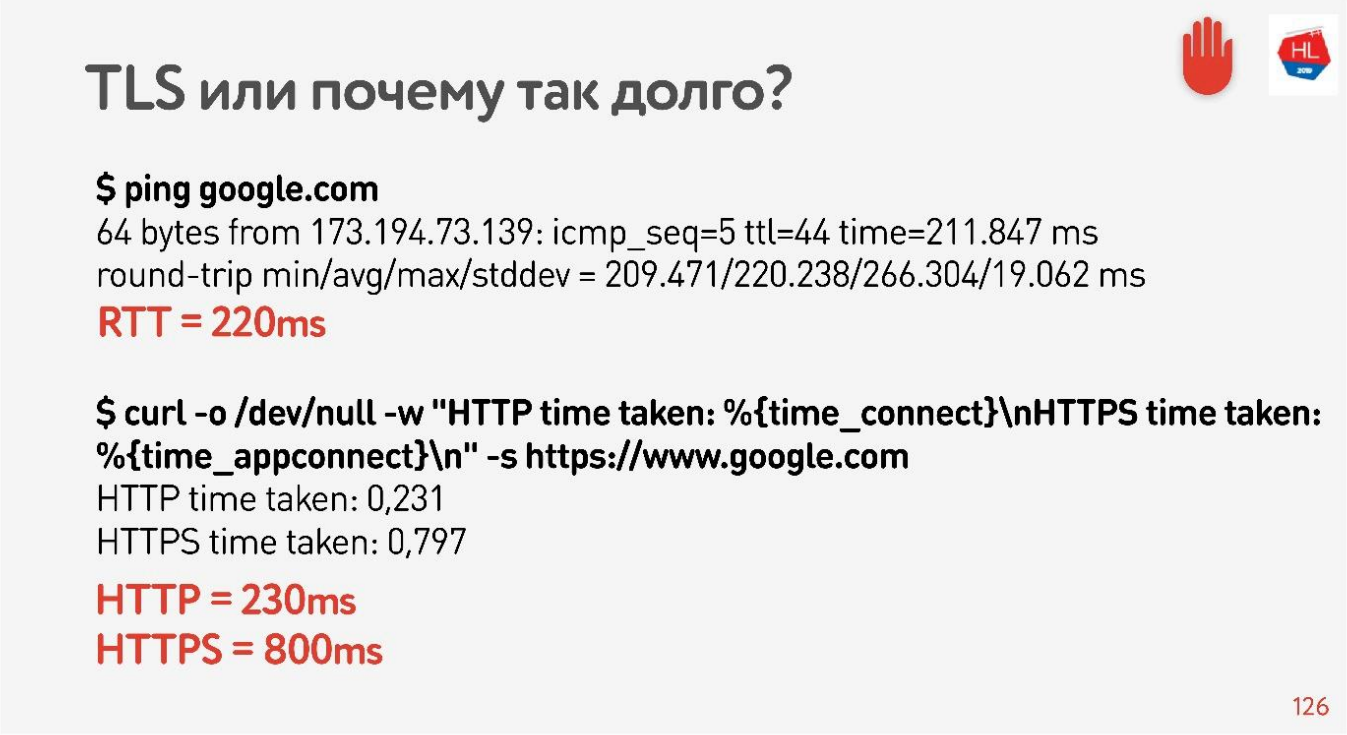



Nginx все это умеет — просто включите, все будет работать (или в ядре включите).

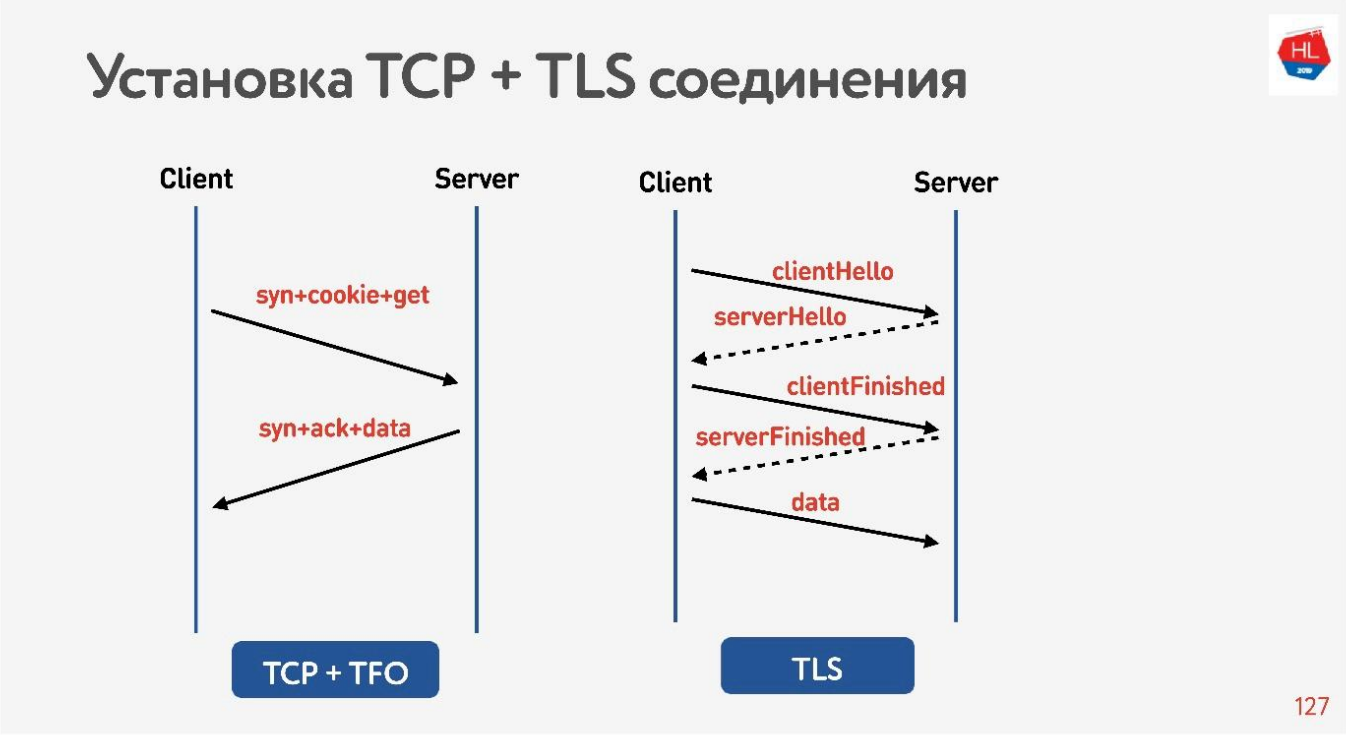
## TLS

Давайте проверим, что TLS — это плохо.

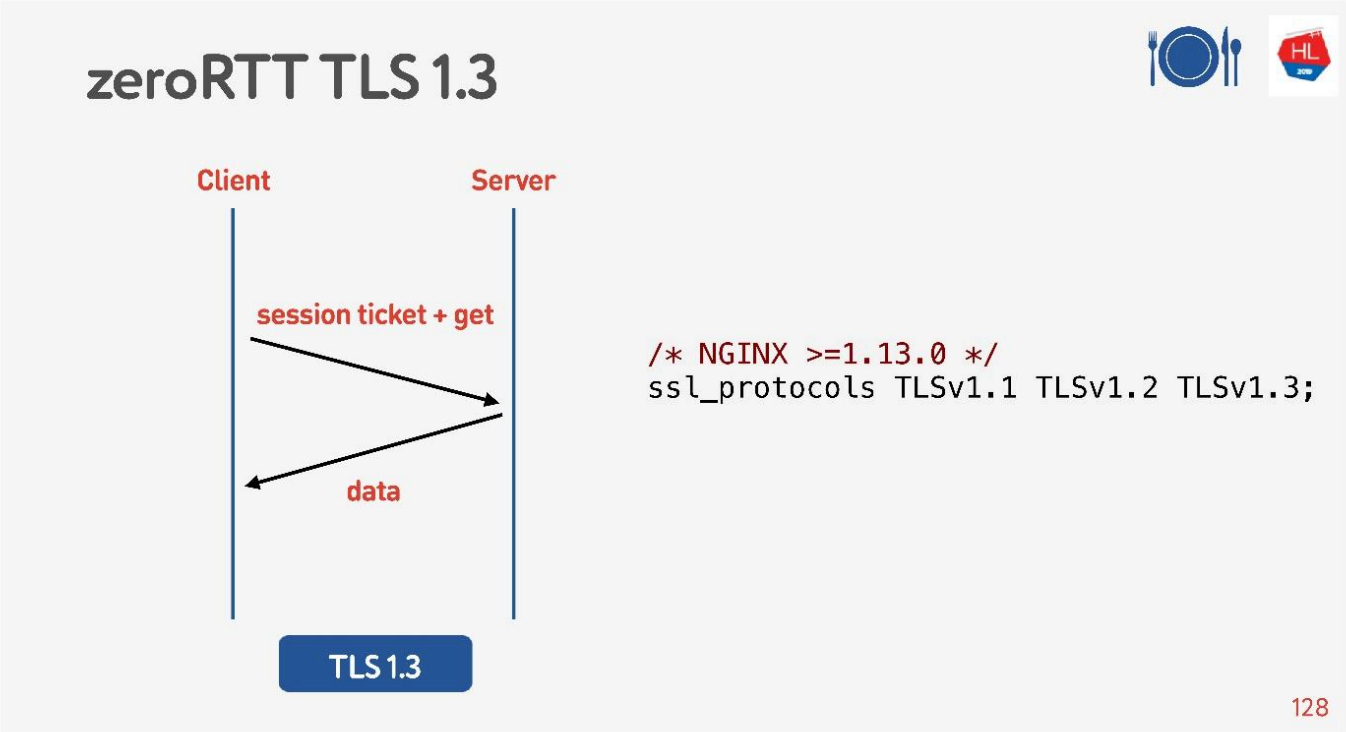
Я опять настроил net shaper на 200 мс, попинговал google.com и увидел, что RTT = 220 – мой RTT + RTT shaper. Потом сделал запрос по HTTP и HTTPS. Выяснил, что по HTTP можно за время RTT получить ответ, то есть TFO работает для Google с моего компьютера. Для HTTPS это заняло больше времени.




Это такие обычные накладные расходы TLS, который требует обмен сообщениями для того, чтобы установить безопасное соединение.



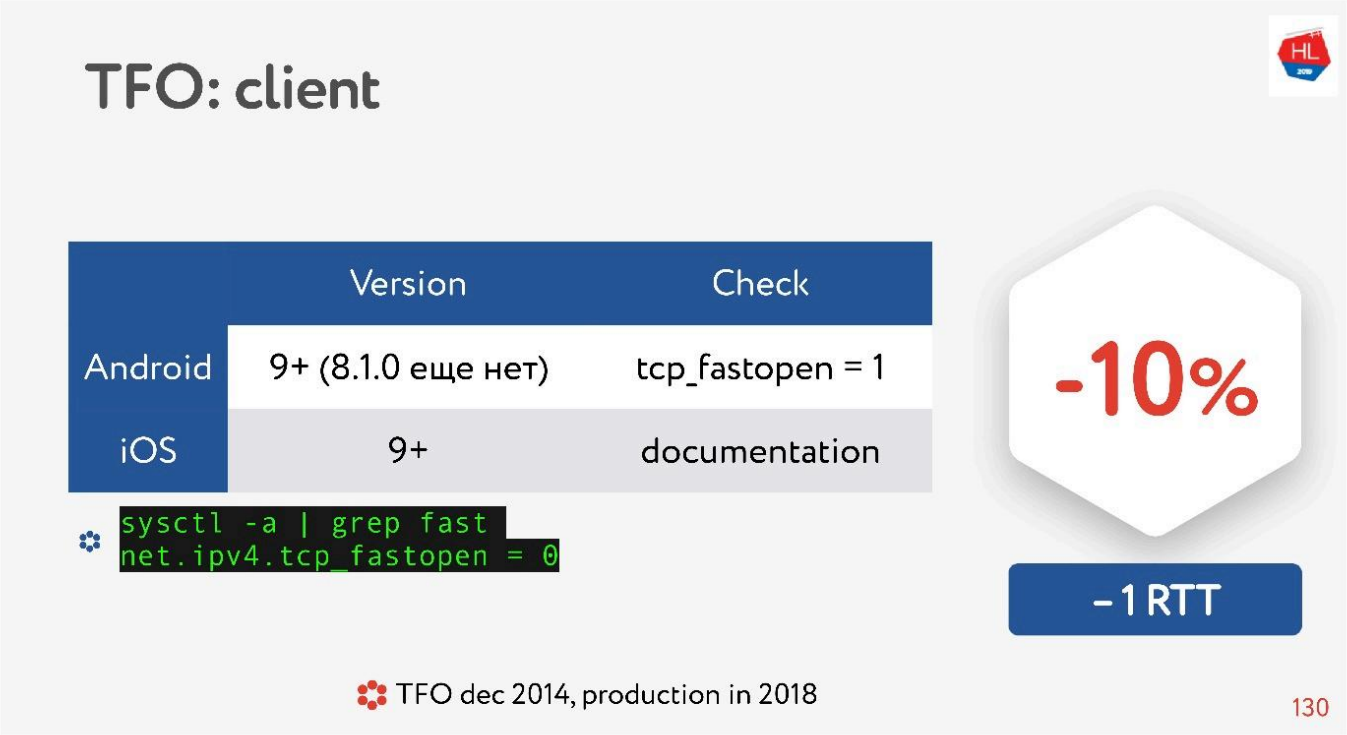

Для этого за нас подумали, добавили TLS 1.3. Его тоже легко включить в nginx.




Кажется, что все работает. Но давайте посмотрим, что там на наших мобильных клиентах, которые всем этим пользуются.

Что там у клиентов

TCP Fast Open — классная штука. По статистике.



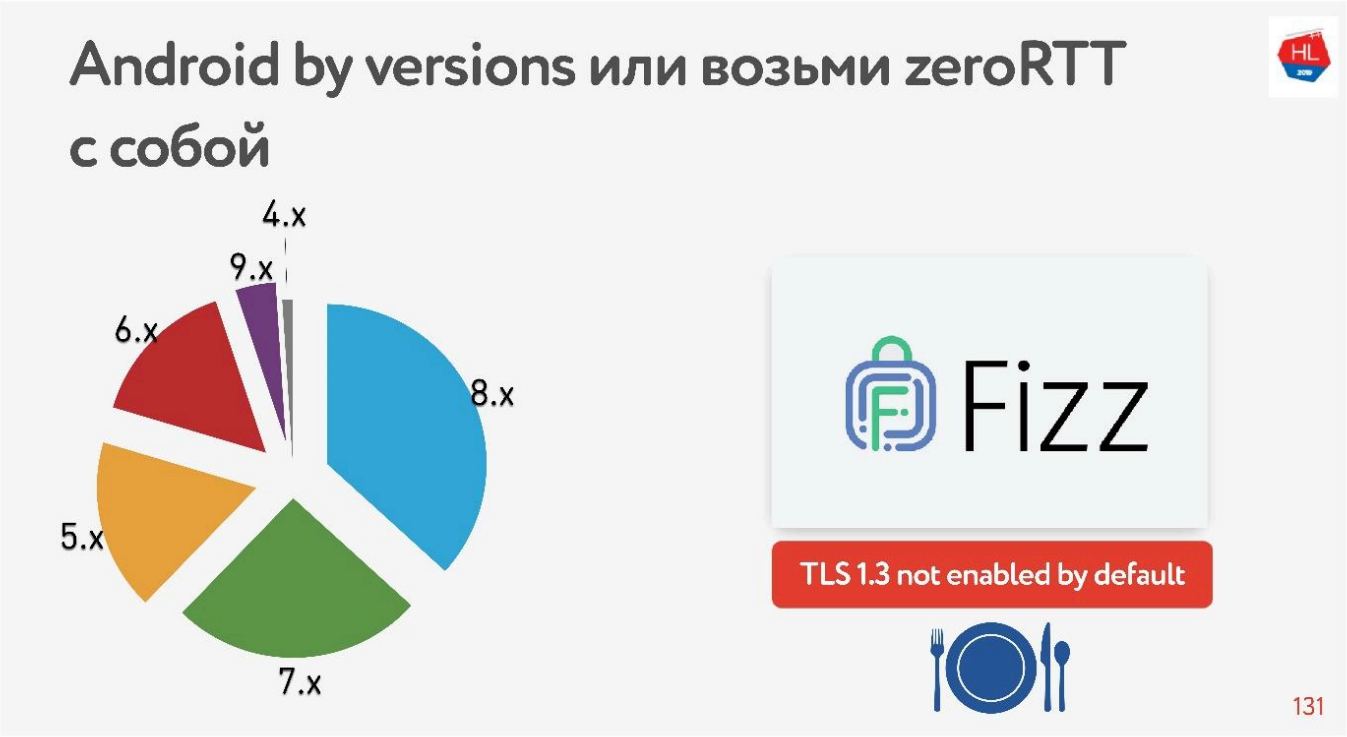

Есть много статей, которые говорят, что установка соединения гарантированно пройдет быстрее на 10%. Но на Android 8.1.0 (я смотрел различные устройства) ни у кого нет TFO. На Android 9 я видел TFO на эмуляторе, но не не реальных устройствах. С iOS чуть получше. Вот так это можно посмотреть:

```
sysctl -a | grep fast

net.ipv4.tcp_fastopen = 0
```

Почему так произошло? TCP Fast Open предложили еще в 2014 году, теперь он уже стандарт, поддерживается в Linux и все здорово. Но есть такая проблема, что TFO handshake стали в некоторых сетях разваливаться. Это происходит потому, что некоторые провайдеры (или какие-то устройства) привыкли инспектировать TCP, делать свои оптимизации, и не ожидали, что там будет TFO handshake. Поэтому его внедрение заняло так много времени, и до сих пор мобильные клиенты его не включают по умолчанию, по крайне мере, Android.

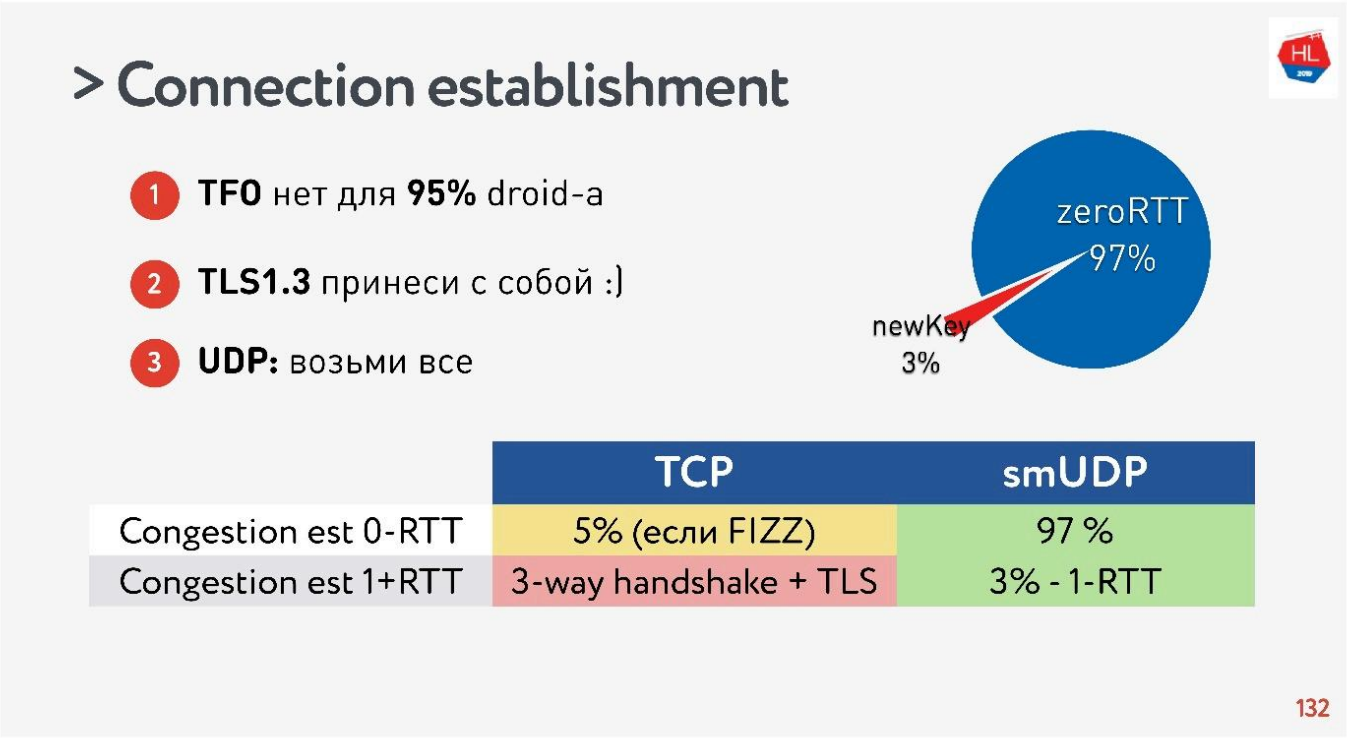
С TLS 1.3, который нам обещает zero-RTT установки соединений еще лучше. Я не нашел устройств на Android, на котором бы он работал. Поэтому Facebook сделал библиотеку Fizz. Пару месяцев назад она стала доступна в опенсорсе, ее можно притащить с собой и использовать TLS 1.3. Получается, что даже безопасность нужно тащить с собой, в ядре этого ничего не появляется.




На диаграмме представлено использование нашими мобильными клиентами различных версий Android. V 9.x совсем немного — там, где TFO может появиться, а TLS1.3 пока нет нигде.

**Выводы про установку соединения:**

- TFO недоступно для 95% устройств.
- TLS1.3 нужно притащить с собой.
- Если нужно это повторить в UDP, то переносив все это на UDP и повторяем.



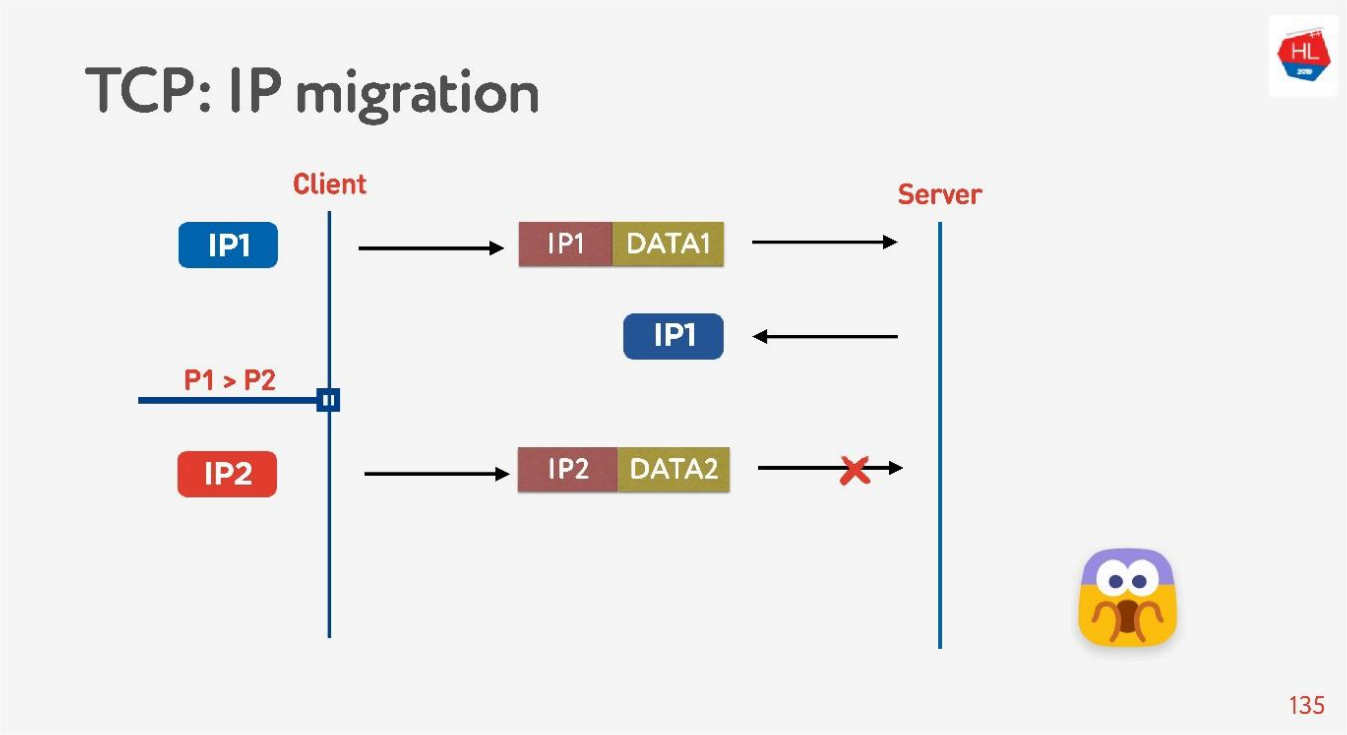

Выяснилось, что 97% создаваемых соединений используют уже имеющийся ключ, то есть 97% создается за zero RTT, и только 3% новых. Ключ какое-то время хранится на устройстве.

TCP этим похвастаться не может. Максимум в 5% случаев, если вы все сделаете правильно, вам удастся получить настоящий zero-RTT, о котором сейчас все разговаривают.

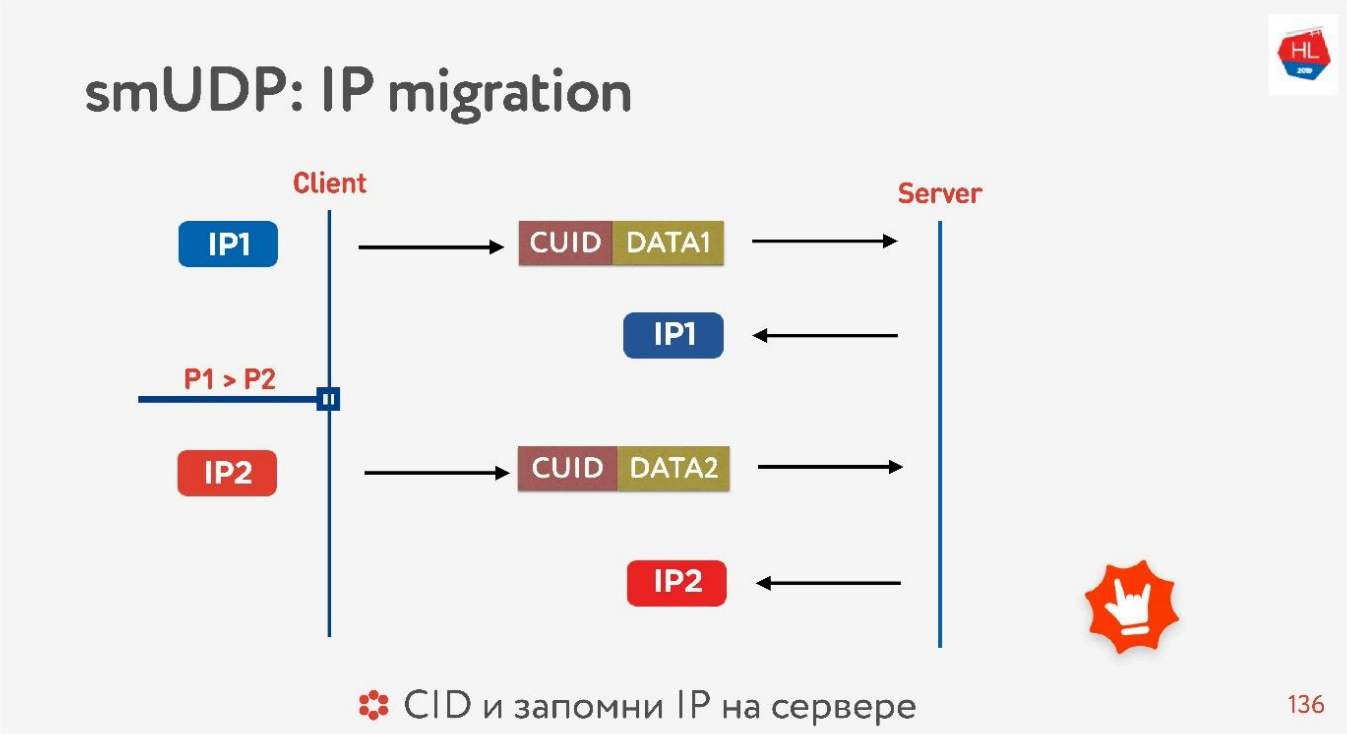
## Смена IP-адреса

Часто, когда вы уходите из дома, ваш телефон переключается с Wi-Fi на 4G.

TCP работает так: сменился IP-адрес — соединение развалилось.




Если вы пишите свой UDP протокол, то очень просто, внедряя в каждый пакет connection ID (CUID), вы сможете его идентифицировать, даже если он пришел с другого IP-адреса.




Понятно, что надо удостовериться в безопасности, что у него правильный ключ, все расшифровывается, и т.д. Но в принципе вы можете начать отвечать на этот адрес, проблем с этим не будет.

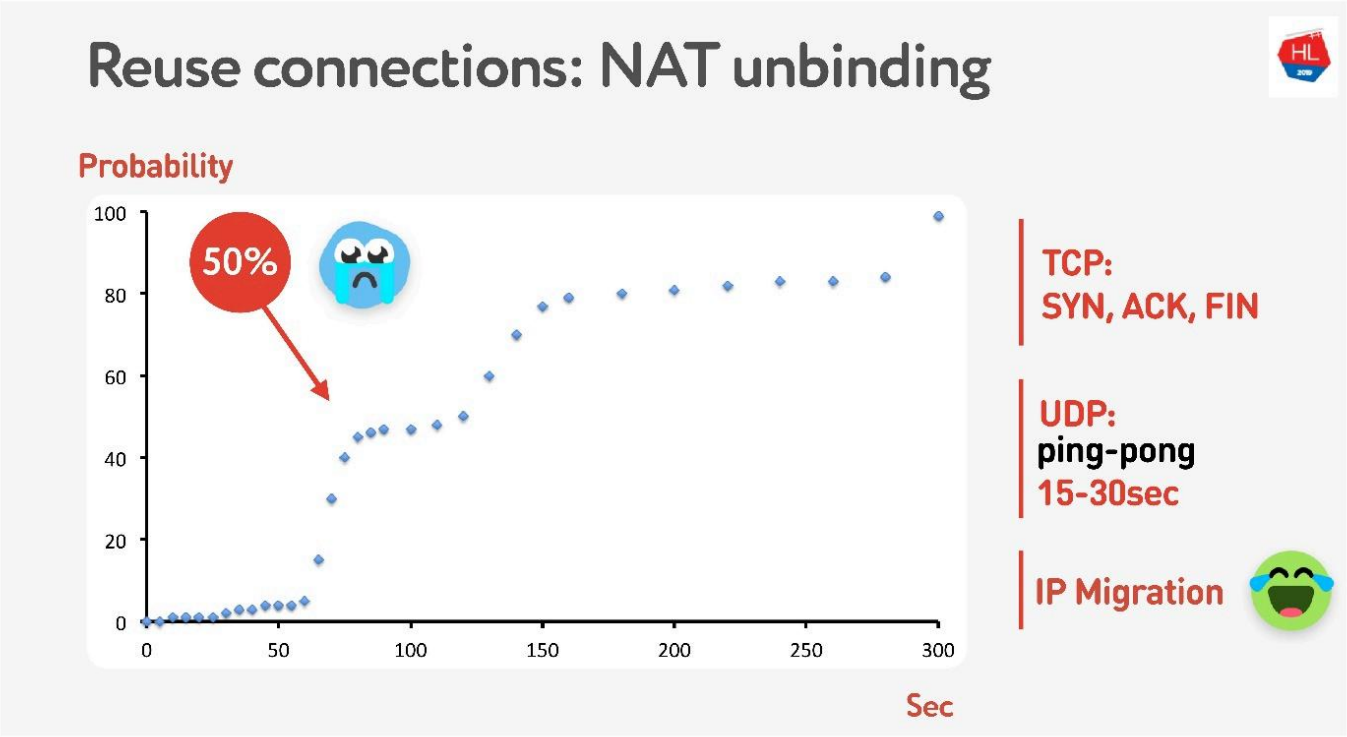
В TCP IP Migration — это невозможная вещь.

Если вы делаете свой UDP, и пришли на тот же самый сервер, нужно немножко поколдовать, включить CID в каждый пакет, и вам удастся использовать установленное соединение при смене IP адреса.

## Connection reuse

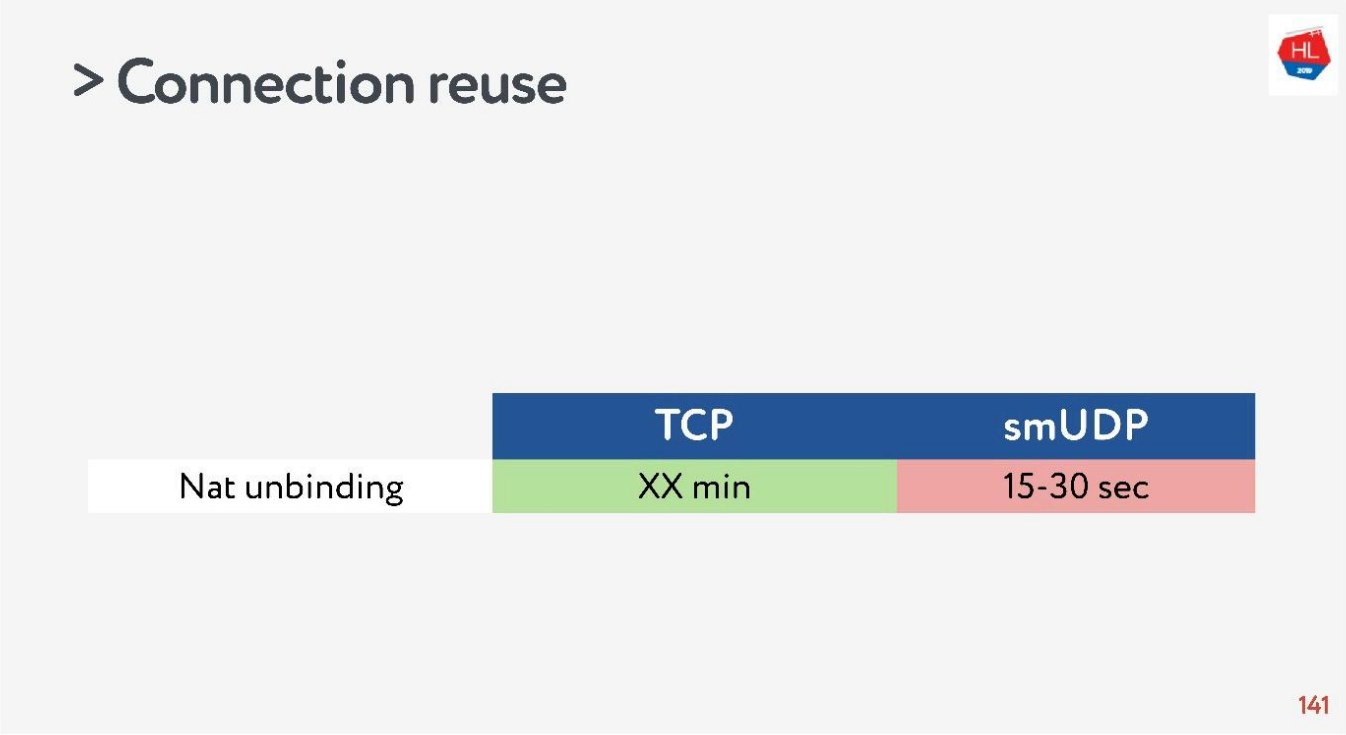
Все говорят, что нужно переиспользовать соединения, потому что соединения — очень дорогая вещь.


Но в переиспользовании соединения есть подводные камни.



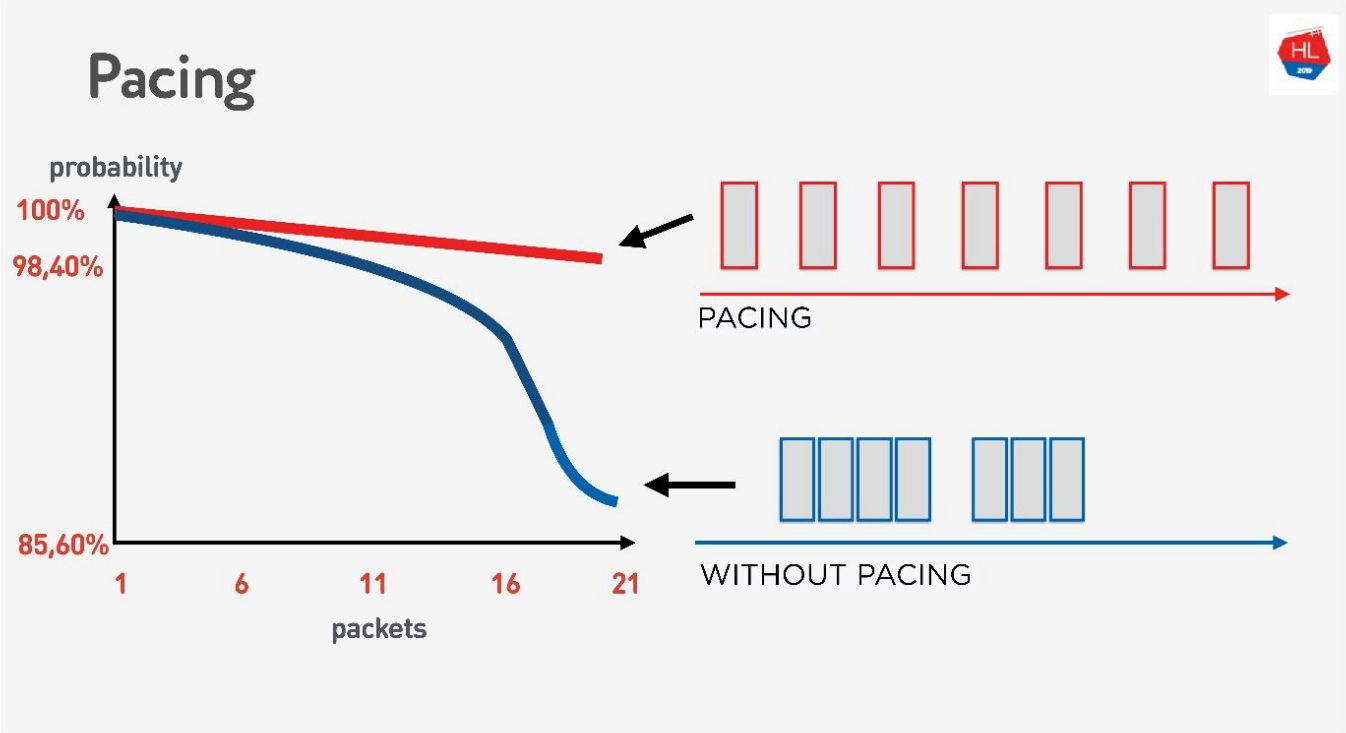

Наверное, многие помнят (если нет, то см. сюда), что не у всех публичные адреса, а есть NAT, который обычно на домашнем роутере хранит какое-то время mapping. Для TCP понятно, сколько хранить, а для UDP — непонятно. NAT оперирует timeout, если аккуратно измерить этот timeout, то получим, что примерно за 15-30 секунд более 50% соединений начнут разрушаться.

Ничего страшного — сделаем ping-pong пакета по 15 с. Для случаев, когда соединение таки разрушилось, есть IP Migration, который недорого позволит сменить порт на маршрутизаторе.



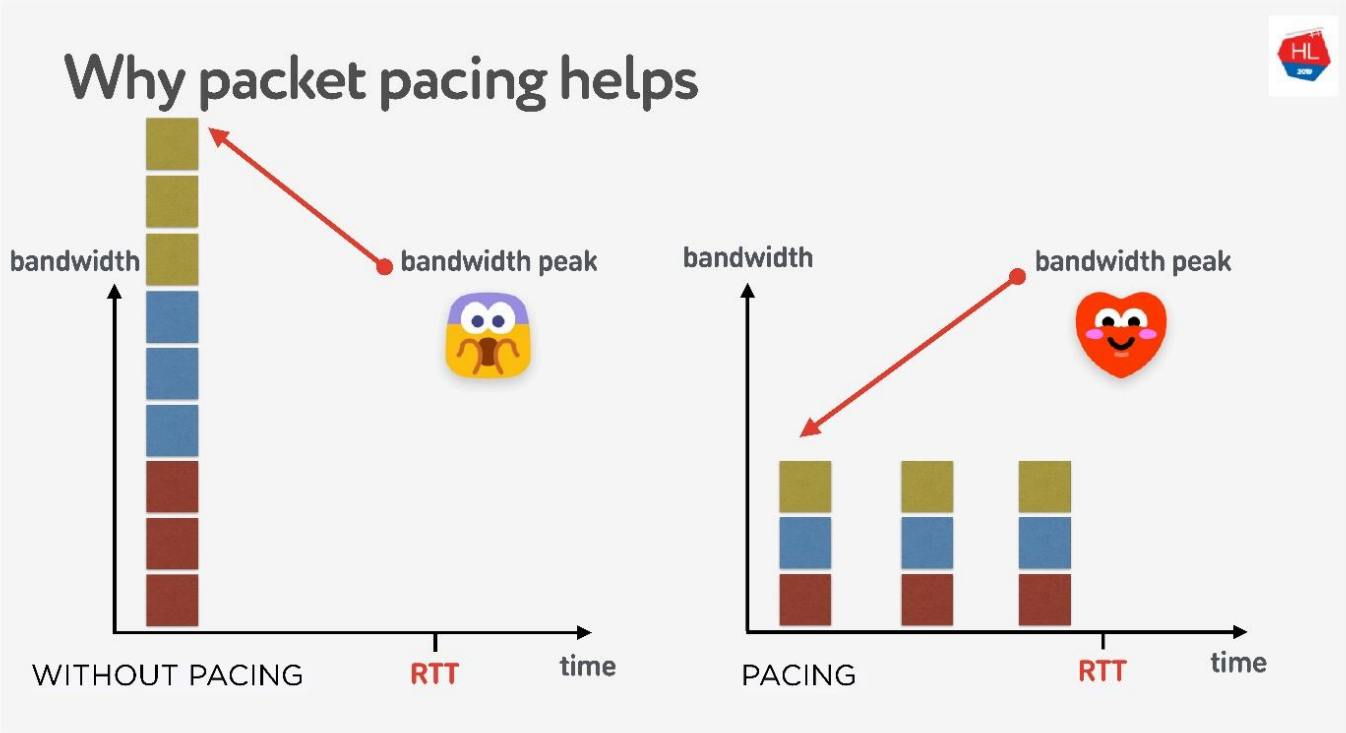

## Packet pacing

Это очень важная вещь, если вы делаете свой UDP-протокол.



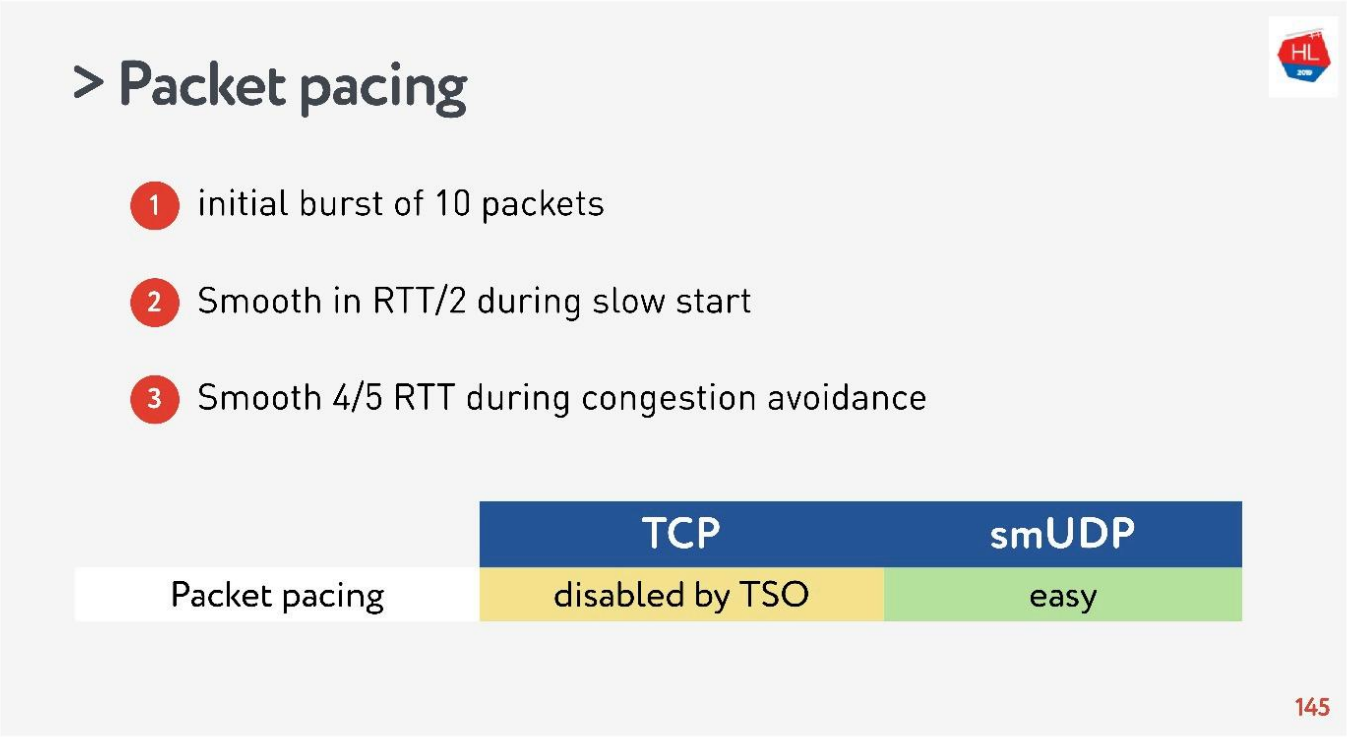

Если очень просто, то чем дольше вы непрерывно посылаете пакеты в сеть, тем больше вероятность packet loss. Если пакеты проредить, то packet loss будет ниже.

Есть много разных теорий, как это работает, но мне нравится эта.




Есть 3 соединения, которые создаются в один момент времени. У вас есть так называемый initial window — 10 пакетов, создаваемых одновременно. Конечно, в этот момент может не хватить bandwidth. Но если их аккуратно распределить, разделить, то все будет отлично, как на правом рисунке.

Таким образом, если задавать равномерный темп отправки пакетов, прореживать их, то вероятность того, что будет единомоментное переполнение буферов, станет ниже. Это не доказано, но теоретически получается так.

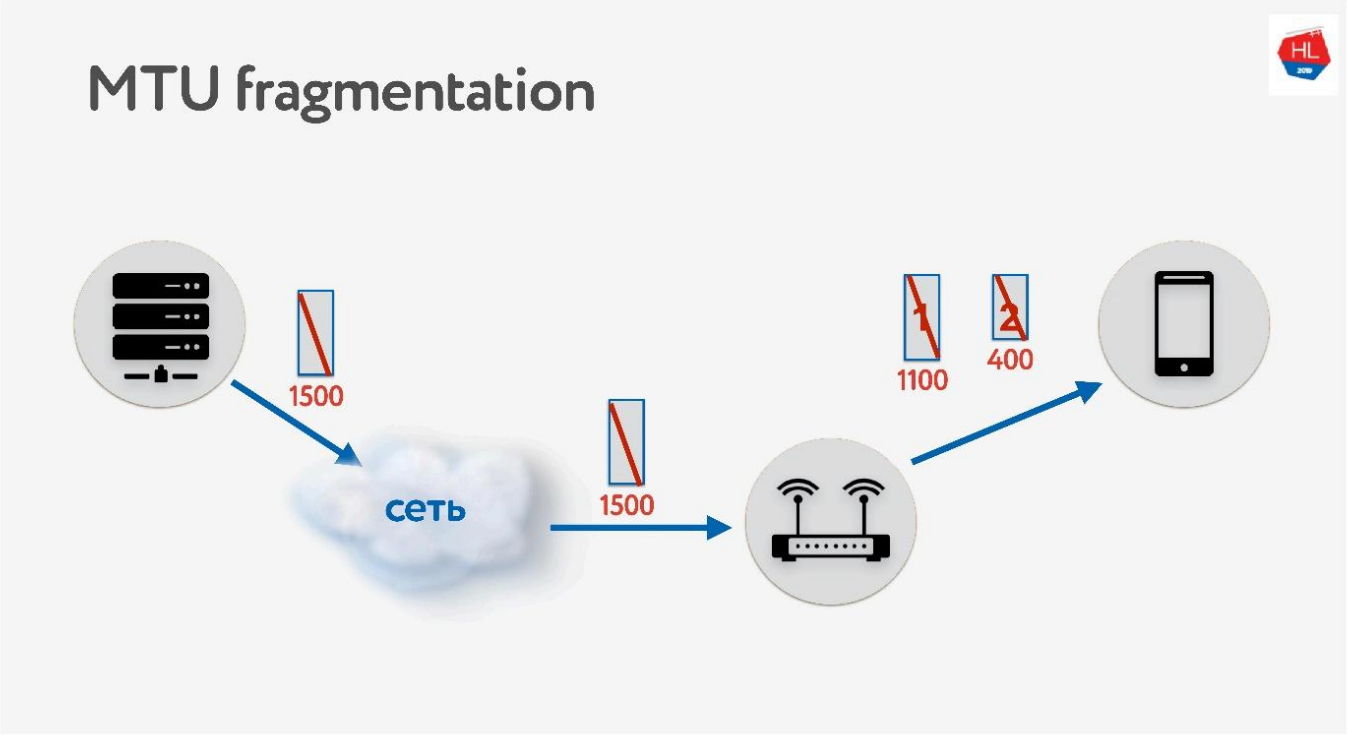



Когда нужно прорежать пакеты (делать pacing):

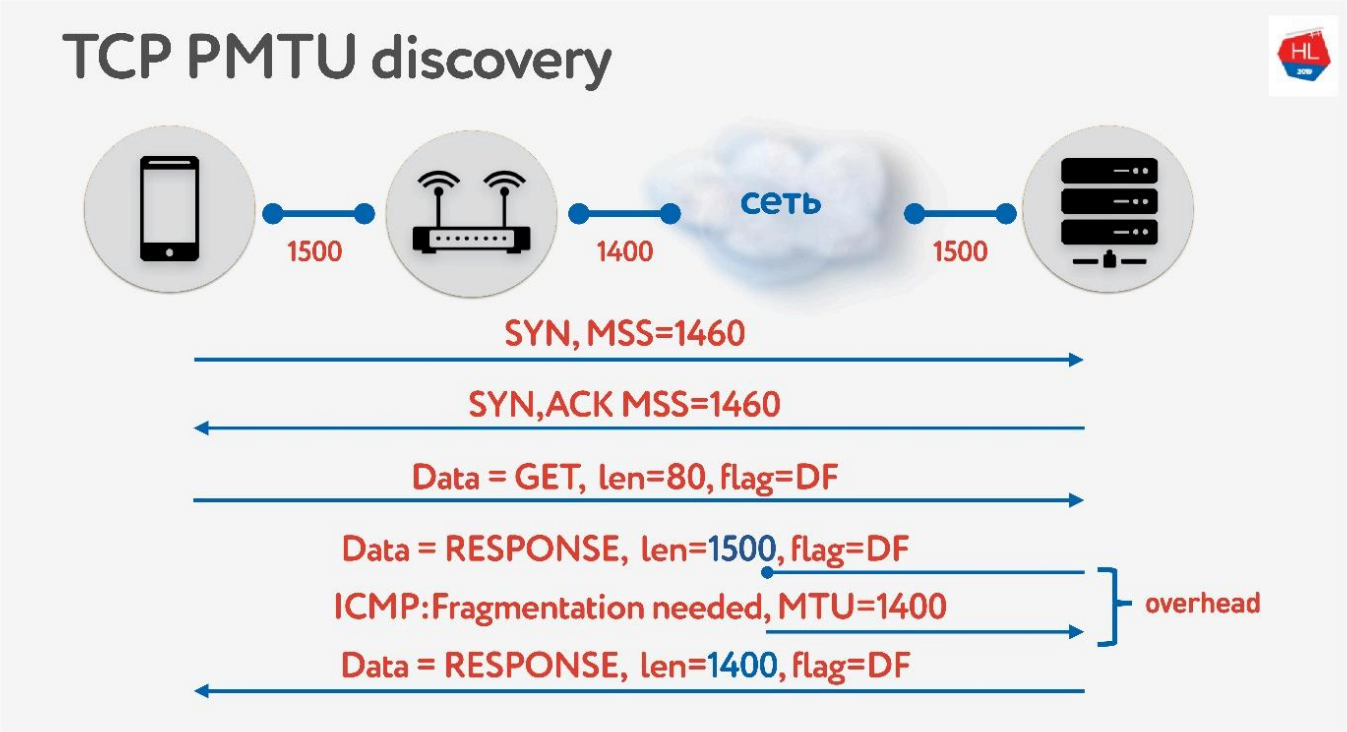
- Когда создаете окно.
- Когда увеличиваете окно, например, рекомендуется добавлять столько пакетов, сколько можно отправить за RTT/2. Это не ухудшит время доставки, но снизит packet loss.
- В случае congestion loss для уменьшения окна нужно еще больше размазать пакеты. 4/5 RTT — эмпирически подобранная цифра.

## MTU

При написании своего UDP-протокола обязательно нужно помнить про MTU. MTU — это размер данных, которые вы можете переправить.

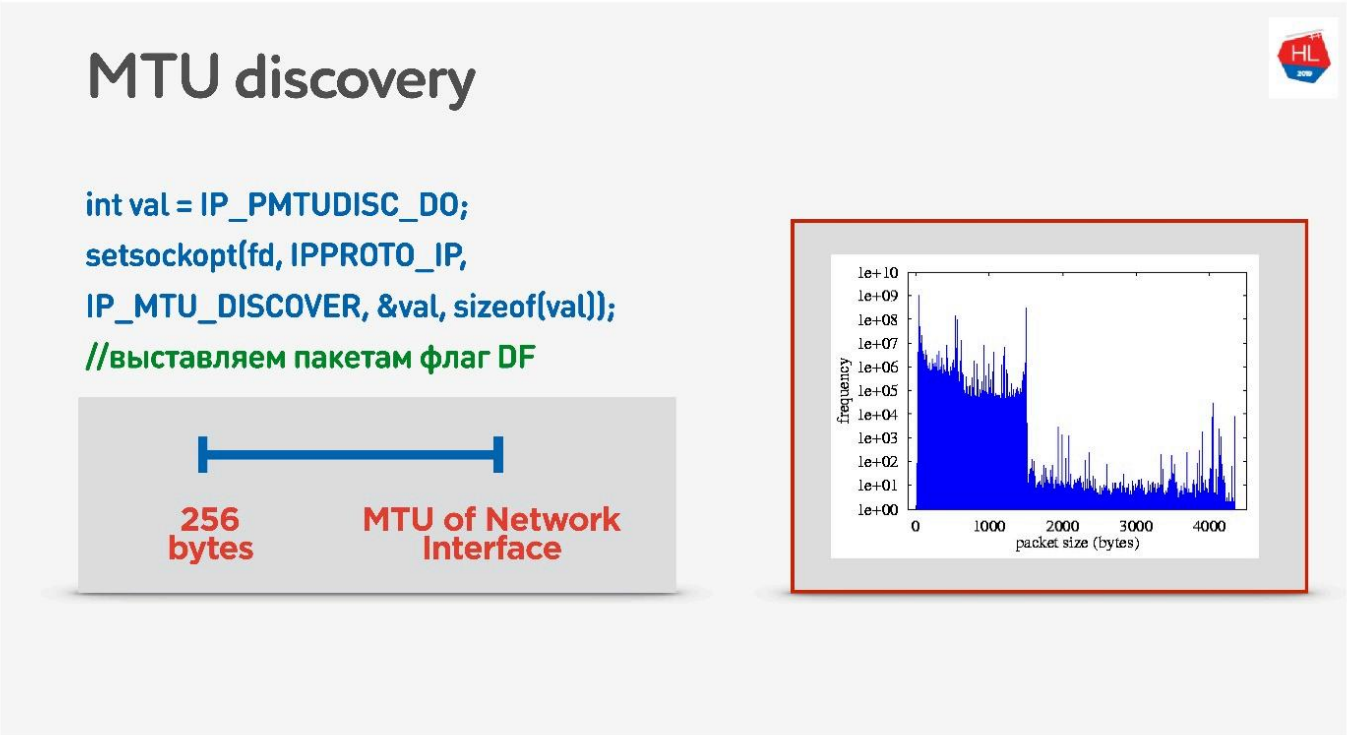



Отправляем пакеты с сервера на клиент, например, размером 1500. Если на пути встречается маршрутизатор, который не поддерживает этот размер MTU, он его фрагментирует. Единственная проблема фрагментации в том, что если потеряется один пакет, потеряются оба, и придется все это ретрасмитить. Поэтому в TCP есть алгоритм определения MTU — PMTU.



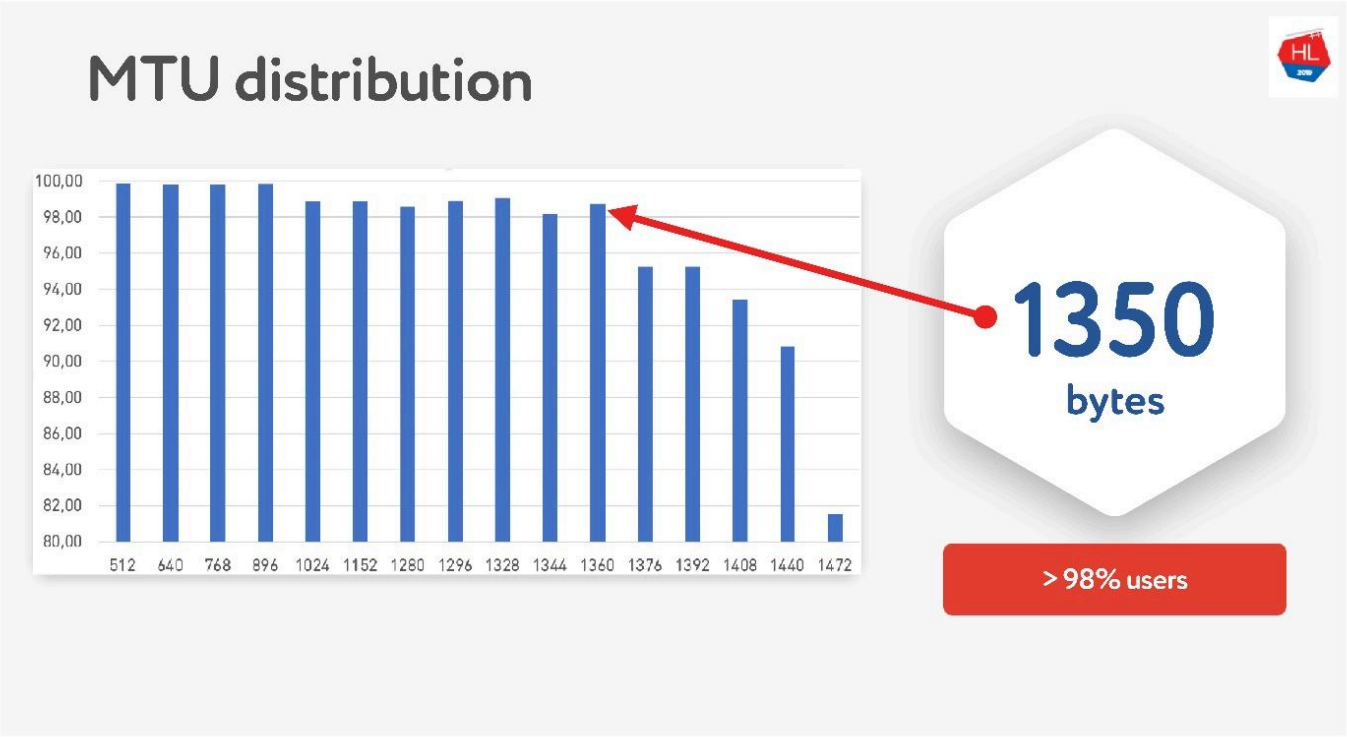

Каждый маршрутизатор смотрит MTU своего интерфейса, отправляет его одному клиенту, другой отправляет своему клиенту, все знают, сколько у них MTU на клиенте. Потом флагом

запрещается фрагментация и отправляются пакеты размером MTU. Если в этот момент кто-то внутри сети поймет, что у него MTU меньше, то по ICMP сообщит: «Извините, пакет пропал, потому что нужна фрагментация» и укажет размер MTU. Мы поменяем этот размер и продолжим отправку. В худшем случае наш небольшой overhead — это RTT/2. Это в TCP.




Если в UDP вам не охота заморачиваться с ICMP, то можно сделать следующее: при отправке обычных данных разрешить фрагментацию. То есть посылать фрагментированные пакеты — пусть они работают. А параллельно запустить процесс, который запретит фрагментацию, бинарным поиском подберет оптимальное MTU, на которое мы потом выйдем. Это не совсем эффективно, потому что вначале MTU будет как бы прогреваться.

Более хитрый вариант — посмотреть распределение MTU по мобильным клиентам.




Со всех клиентов мы отправили пакеты различного размера с запретом фрагментации. То есть если пакет не дойдет, он дропнется, а самый маленький MTU должен доходить стопроцентно. Но есть небольшой packet loss, поэтому на графике есть две горки:

1. 1350 байт — у нас получается вместо 98% доставка сразу 95%.
2. 1500 байт — MTU, после которого уже 80% клиентов такие пакеты не получит.

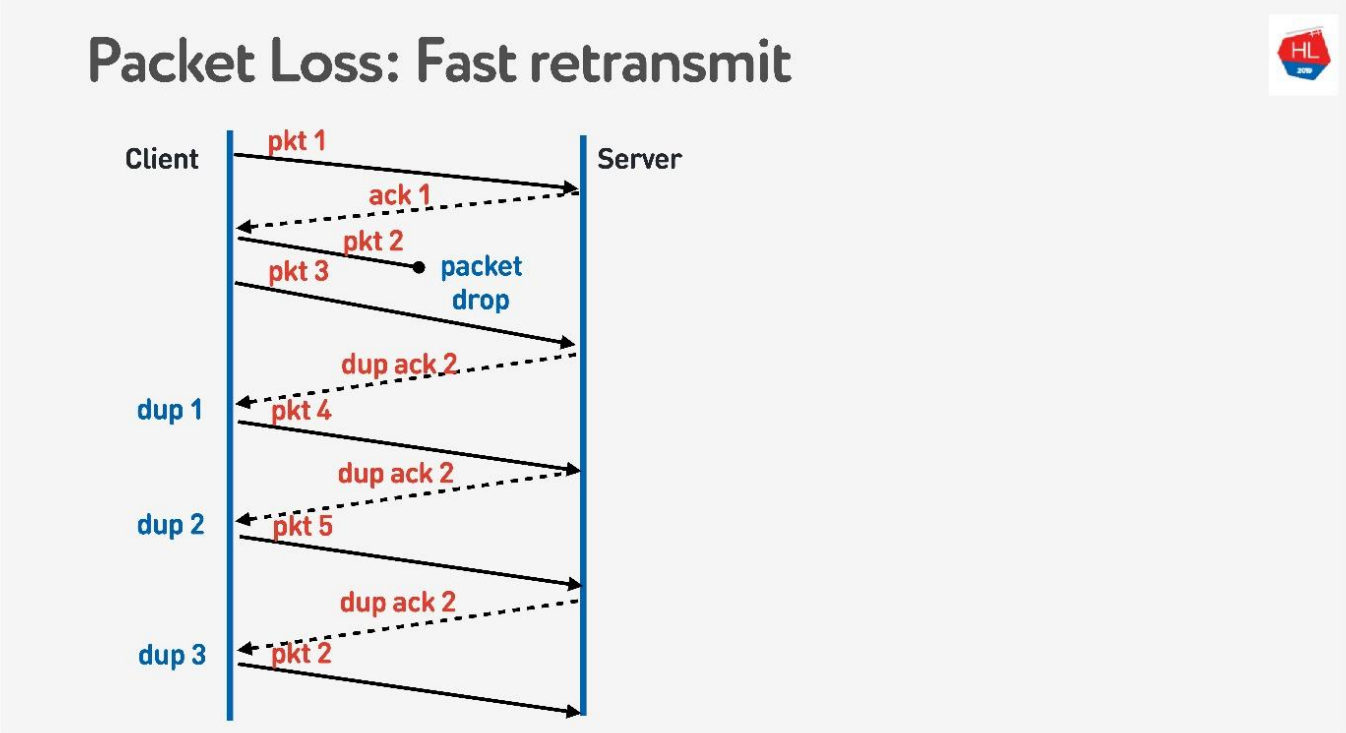
На самом деле можно сказать так: пренебрежем 1-2% наших клиентов, пусть они живут на фрагментированных пакетах. Зато мы сразу будем стартовать с того, с чего надо — это с 1350.

## Исправление ошибок (SACK, NACK, FEC)

Если вы делаете свой протокол, вам нужно исправлять ошибки. Если пакет пропал (для беспроводных сетей это нормально), его нужно восстановить.

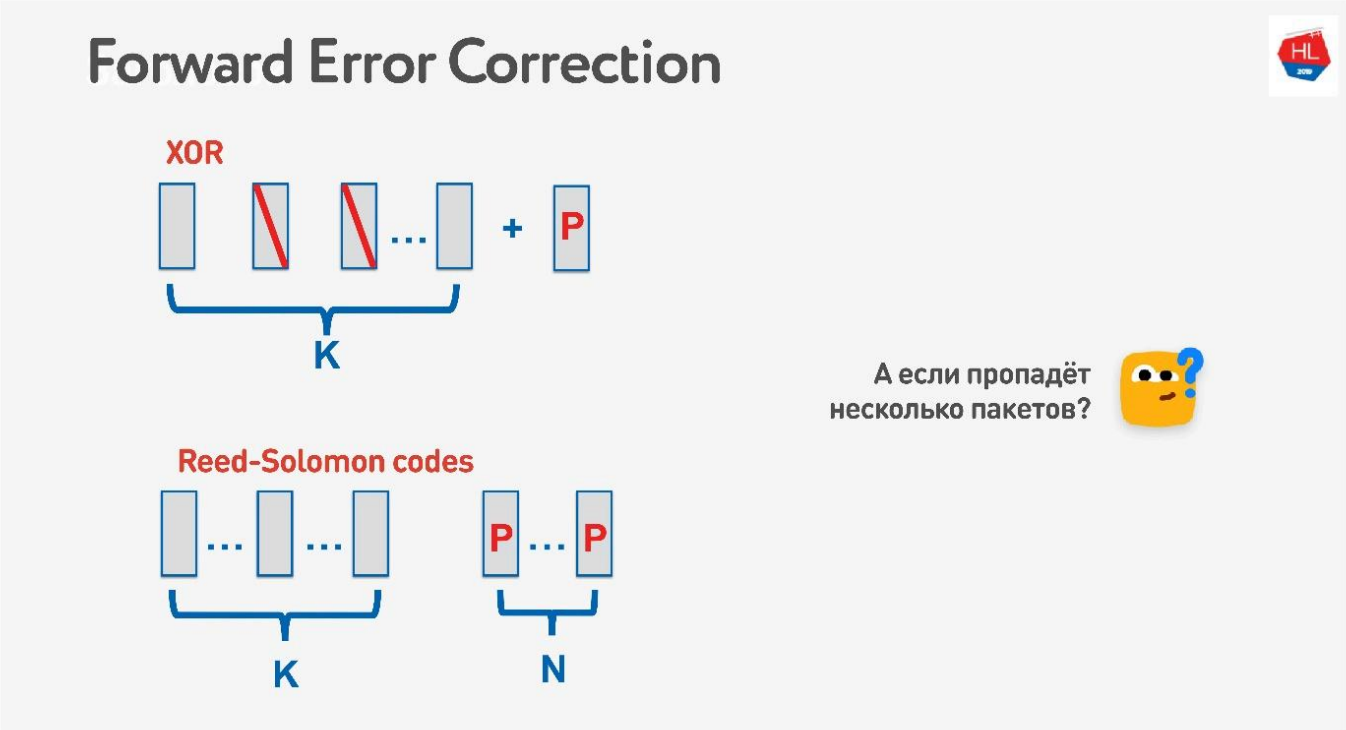
В самом простом случае (подробнее тут), есть ретрансмит через Retransmit Time Out (RTO). Если пакет пропал, ждем время ретрансмита и отправляем его заново.

Следующий алгоритм — это **Fast retransmit**. Это все алгоритмы TCP, но их можно легко перенести в UDP.



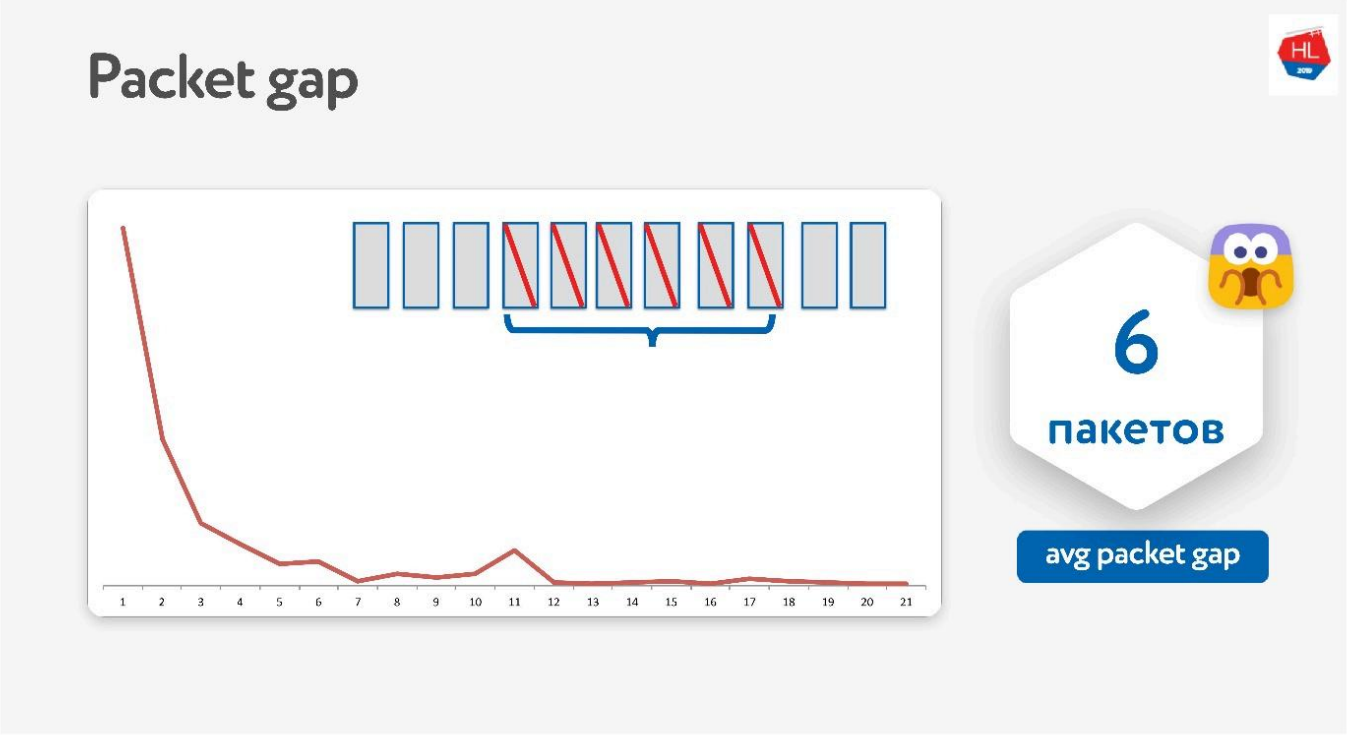

Когда пакет пропал, мы продолжаем посылать — есть передача других пакетов. В это время сервер говорит, что он получил следующий пакет, но предыдущего не было. Для этого он делает хитрый acknowledgement, который равен номеру пакета + 1, и выставляет флаг duplicate ack. Он так эти dup ack посылает, и на третьем мы обычно понимаем, что пакет пропал и посылаем его заново.

Что еще хочется классного сделать, чего нет в TCP и что предлагают делать в UDP — это **Forward Error Correction**.



Кажется, что если мы знаем, что пакеты могут пропасть, мы можем взять набор пакетов, добавить к нему XOR-пакет и починить проблему без дополнительных ретрансмитов сразу на клиенте при получении данных. Но есть проблема, если пропадет несколько пакетов. Кажется, что ее можно решить через parity protection, Reed-Solomon и т.д.

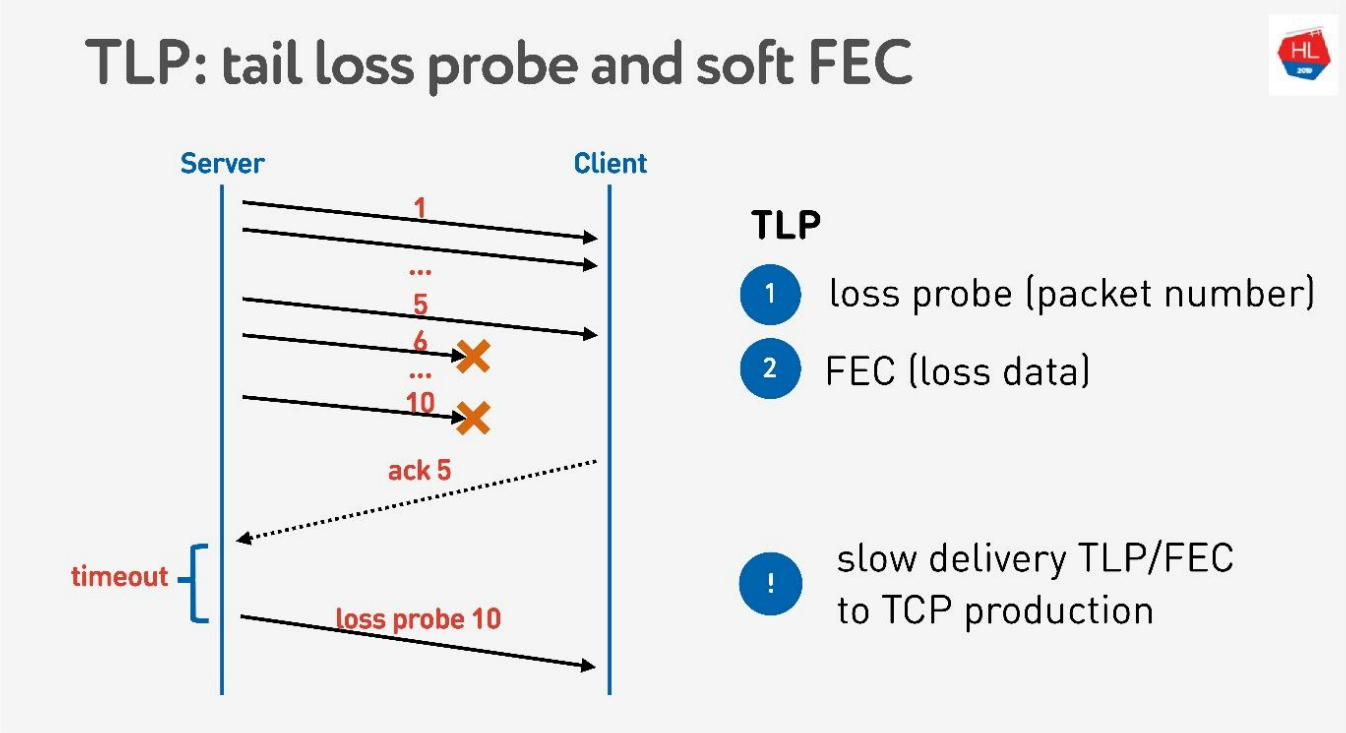
Мы так пробовали, у нас получилось, что на самом деле пакеты пропадают пачками.




Средний packet gap получился 6. Это очень неудобный packet gap — нужно очень много кодов исправления ошибок. При этом есть какой-то пик на 11 — не знаю почему, но пакеты иногда пачками по 11 пропадают. Из-за этого packet gap это не работает.

Google такое тоже пробовал, все грезят FEC, но пока ни у кого не заработало.

Есть еще следующий вариант, когда FEC может помочь.

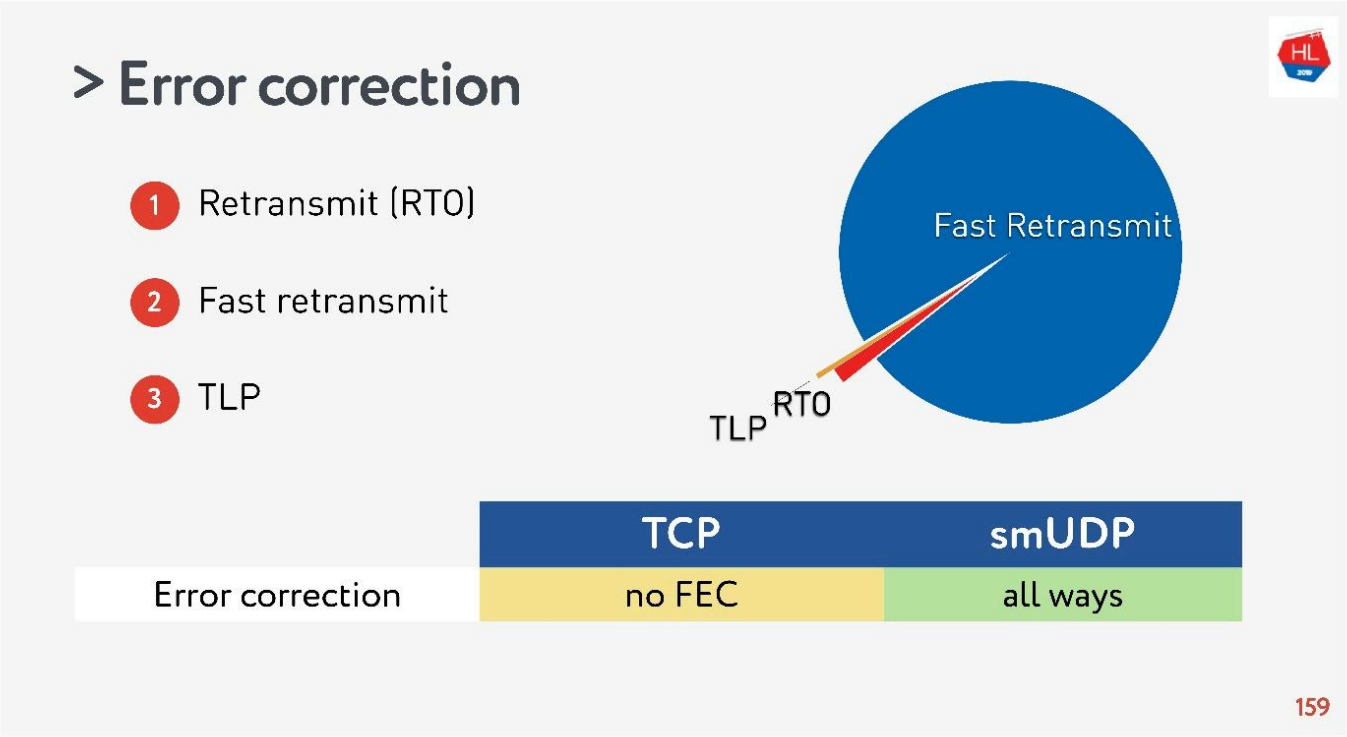



Кроме ретрансмита через Retransmit Time Out, Fast Retransmit, есть еще **tail loss probe**. Это такая штука, когда вы шлете данные, и хвостик пропал. То есть вы послали часть данных, послали пятый пакет — он дошел. Потом начали пропадать пакеты, например, потому что сеть провалилась. Пакеты пропадают, пропадают, и вы получили acknowledgement только на пятый пакет.

Чтобы понять, дошли ли эти данные, вы через какое-то время начинаете делать TLP (tail loss probe), спрашивать, а получен ли конец. Дело в том, что пересылка данных закончилась, и вы ничего не шлете, то Fast Retransmit не сработает. Чтобы это починить, делайте TLP.

К TLP можно добавить FEC. Вы можете посмотреть все пакеты, которые не пришли, посчитать по ним parity и делать отправку TLP с некоторым parity-пакетом.

Это все классно, кажется, должно работать. Но есть такая проблема.




Мы собрали статистику, и получилось, что 98% ошибок чинится через Fast Retransmit. Остальное чинится через Retransmit Time Out, и меньше 1% — через TLP. Если вы еще что-то почините FEC, это будет меньше, чем 0,5%.

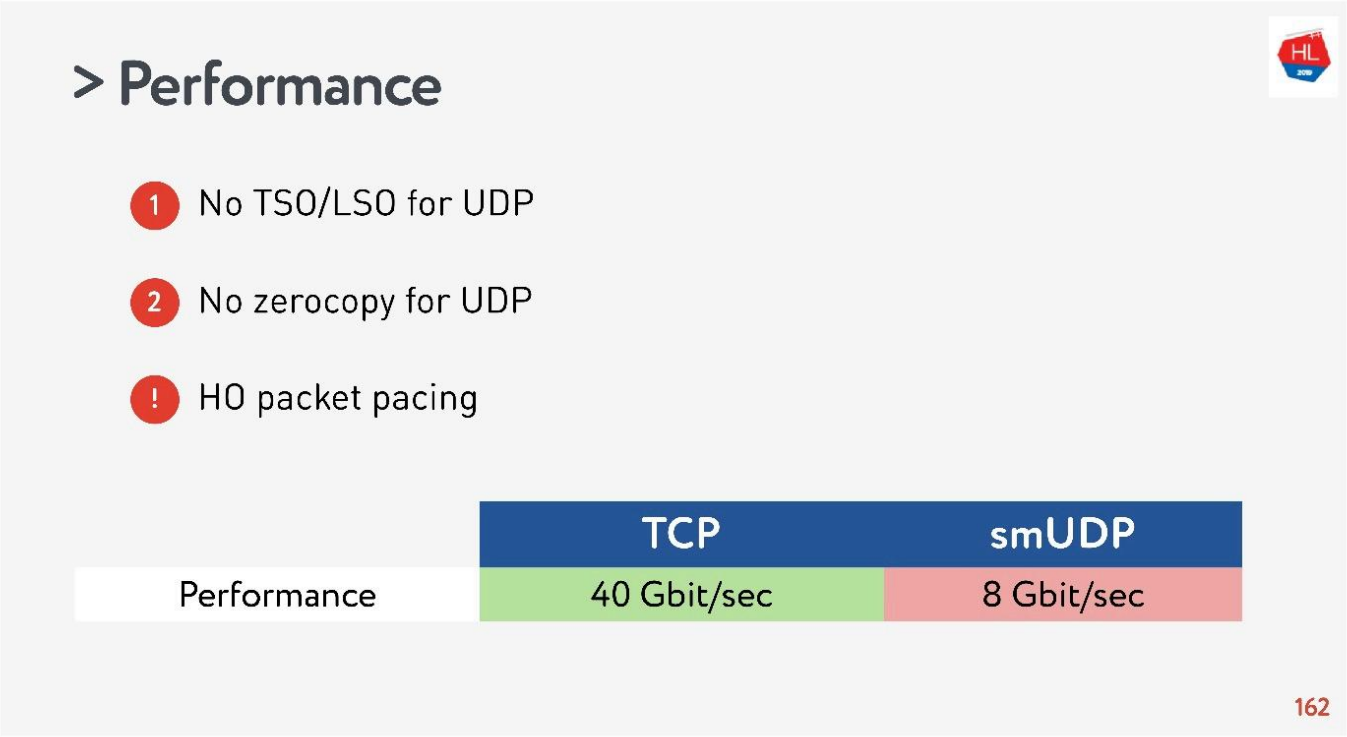
> TCP не поддерживает FEC. В UDP не трудно это сделать, но в общем случае стандартных алгоритмов восстановления TCP хватает.

## Performance

Нельзя было бы не задеть performance, сравнивая TCP с UDP.

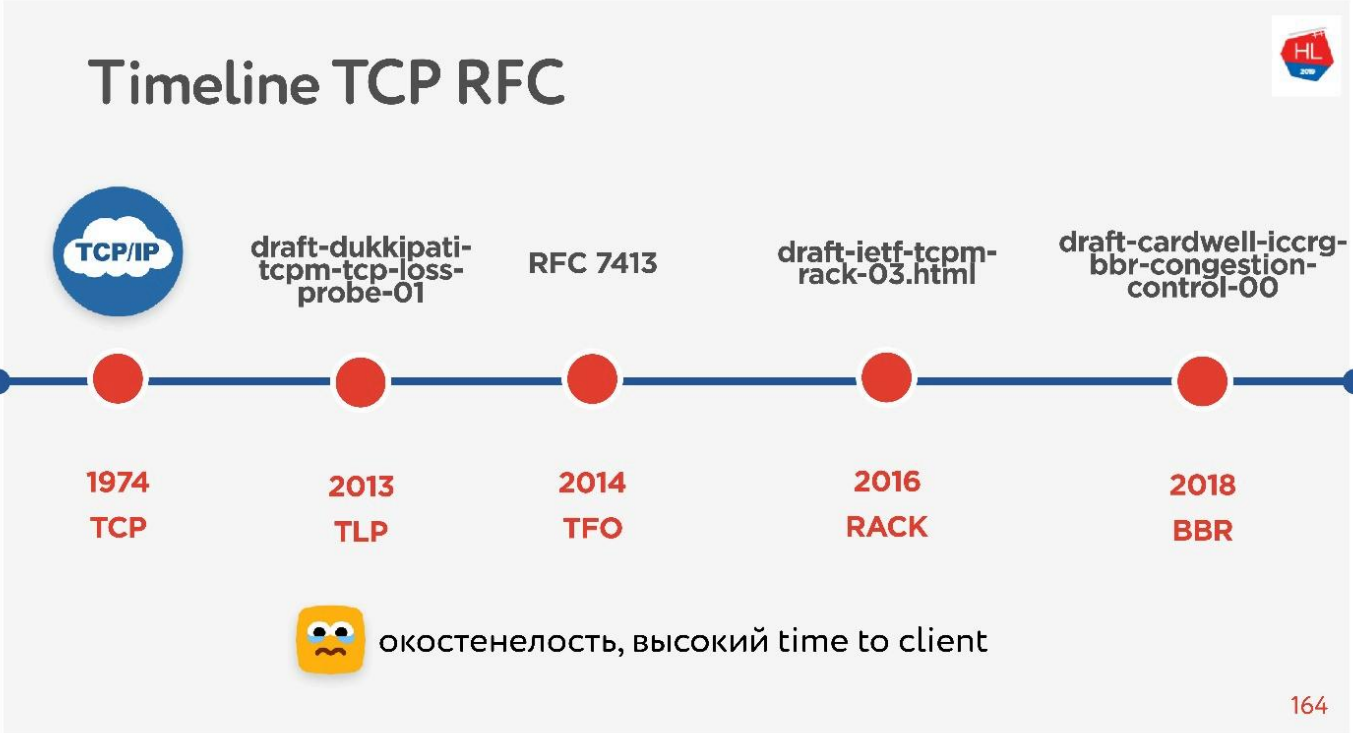
TCP — очень старый протокол с большим количеством различных оптимизаций, например, LSO (large segment offload) и zerocopy. Сейчас для UDP это все недоступно. Поэтому производительность UDP всего 20% относительно TCP с тех же серверов. Но уже есть готовые решения (UDP GSO, zerocopy), которые позволяют в Linux поддержать это.

Основная проблема поддержки оптимизации по zerocopy и LSO в том, что теряется pacing.




## Time to market или что убило TCP

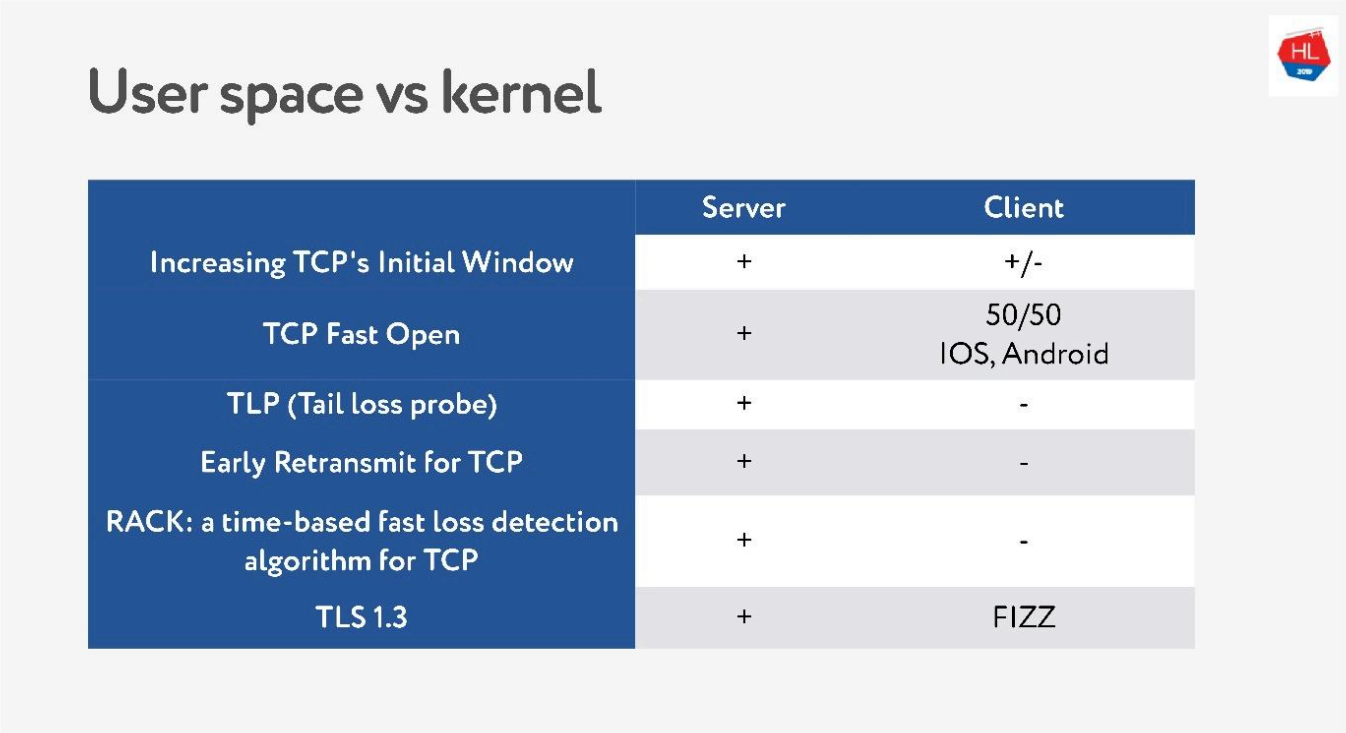
В последнее время, когда стали популярны мобильные беспроводные сети, появилось много различных стандартов TCP: TLP, TFO, новые Congestion control, RACK, BBR и прочее.



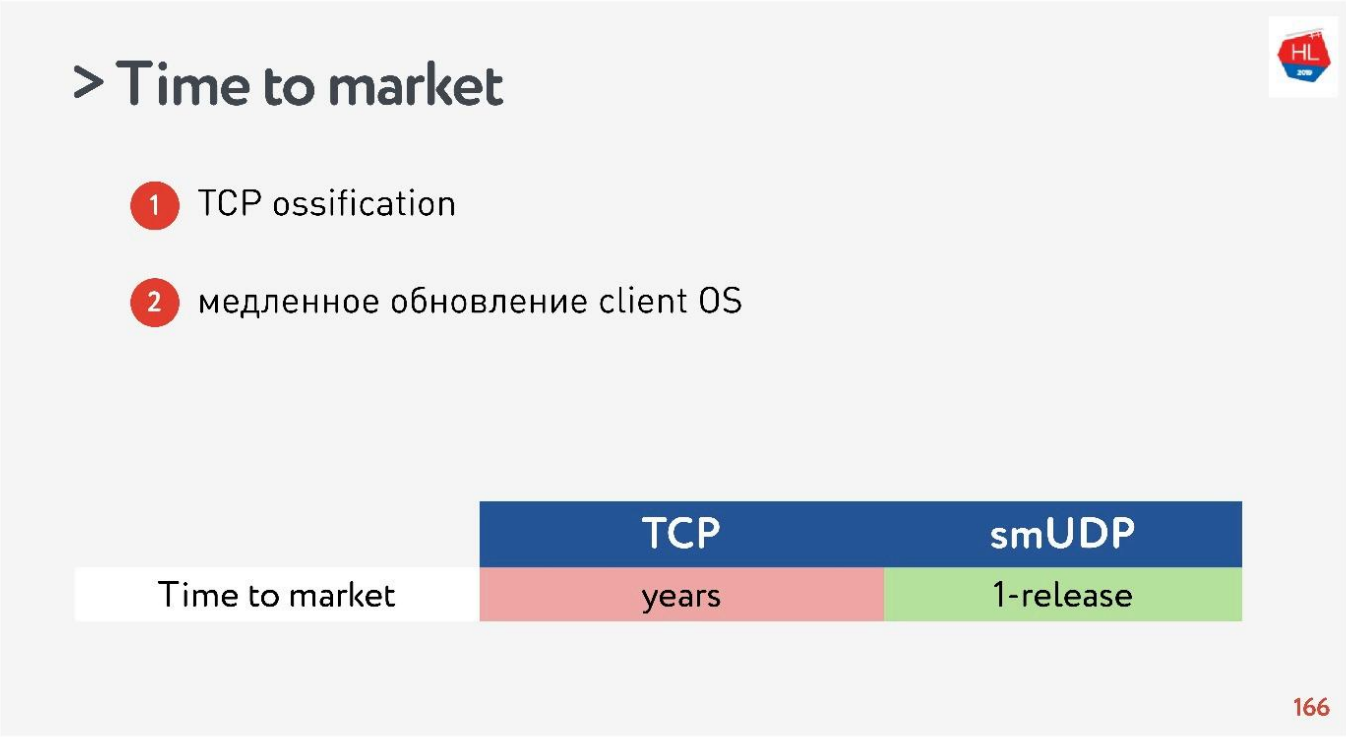

Но основная проблема в том, что многие из них не внедряются, потому что TCP, как говорят, окостенел. Во многих случаях операторы заглядывают в TCP пакеты и ожидают

увидеть то, что они ожидают. Поэтому его очень трудно менять.

К тому же мобильные клиенты обновляются долго, и мы не можем доставить эти обновления. Если посмотреть, что из последних свежих обновлений доступно на клиенте, а что на сервере, можно сказать, что на клиенте почти ничего.

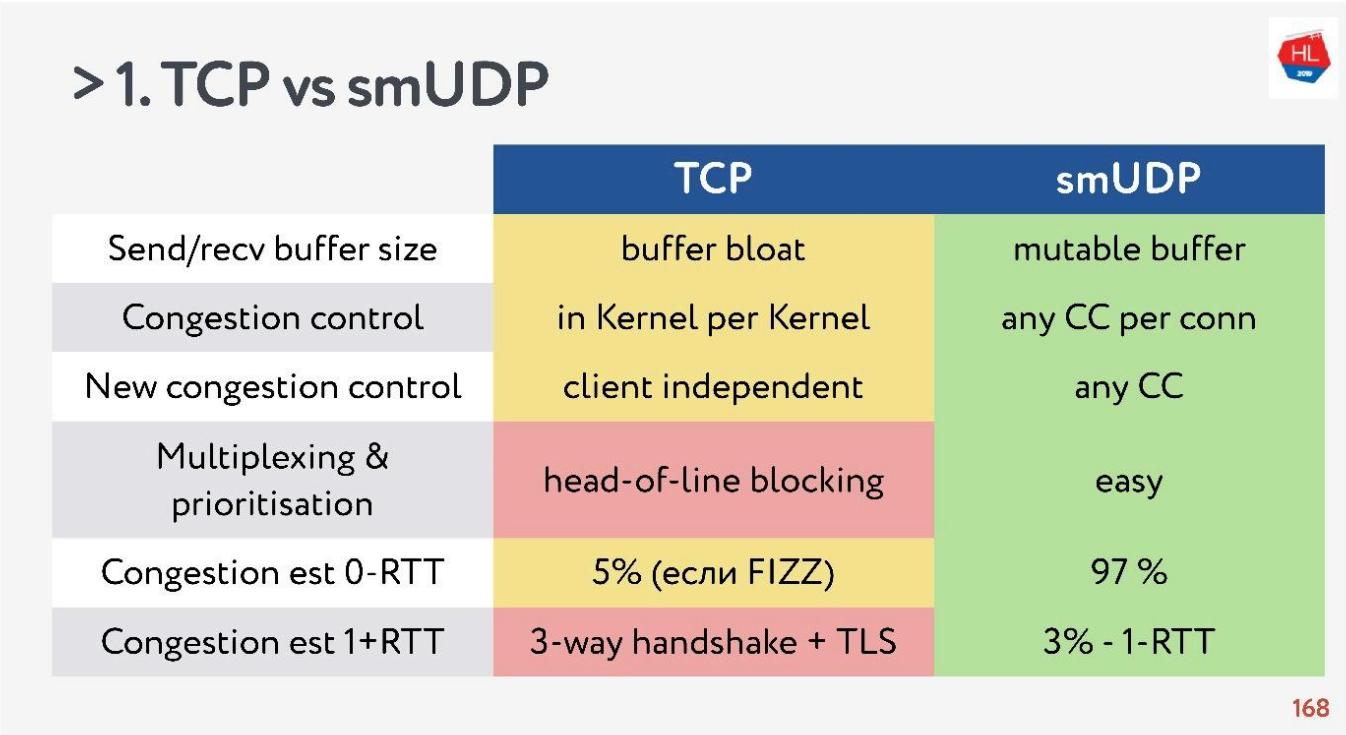



Поэтому решение написать протокол в user space, по крайней мере пока вы все эти фичи накапливаете, кажется не таким плохим.



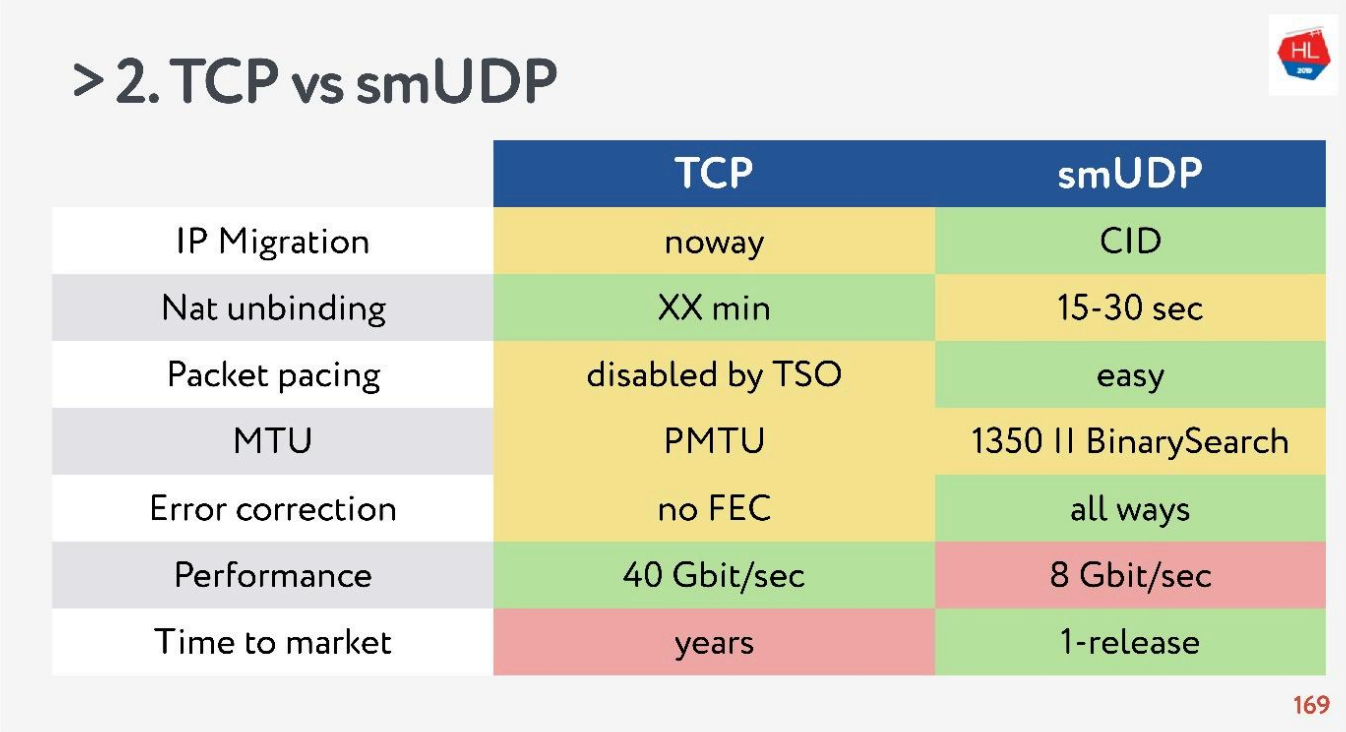

С TCP фичи раскатываются годами. Для своего UDP-протокола, вы можете обновить версию буквально за один апдейт клиента и сервера. Но надо будет добавить version negotiation.

## TCP vs self-made UDP. Final fighting



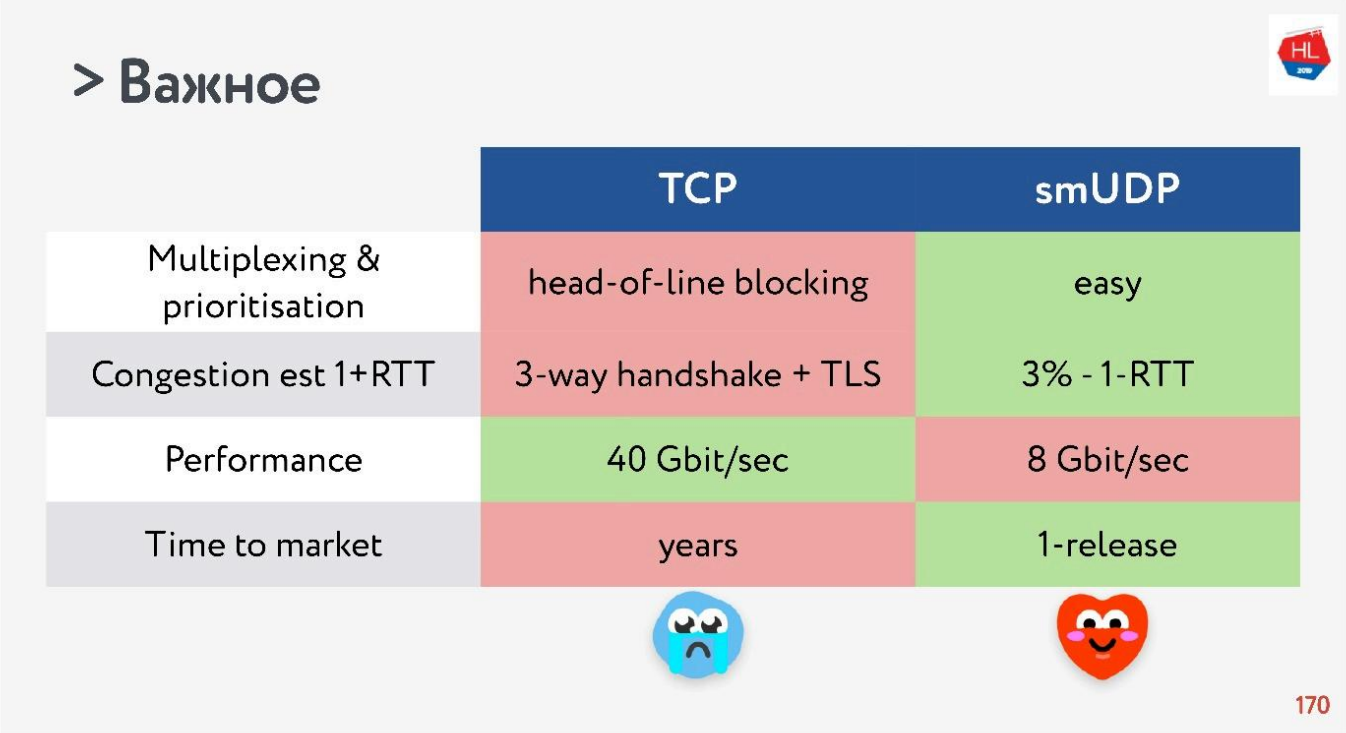

- Send/recv buffer: для своего протокола можно делать mutable buffer, с TCP будут проблемы с buffer bloat.

- Congestion control вы можете использовать существующие. У UDP они любые.
- Новый Congestion control трудно добавить в TCP, потому что нужно модифицировать acknowledgement, вы не можете это сделать на клиенте.
- Мультиплексирование — критичная проблема. Случается head-of-line blocking, при потере пакета вы не можете мультиплексировать в TCP. Поэтому HTTP2.0 по TCP не должен давать серьезного прироста.
- Случаи, когда вы можете получить установку соединения за 0-RTT в TCP крайне редки, порядка 5 %, и порядка 97 % для self-made UDP.




- IP Migration — не такая важная фича, но в случае сложных подписок и хранения состояния на сервере она однозначно нужна, но в TCP никак не реализована.
- Nat unbinding не в пользу UDP. В этом случае в UDP надо часто делать ping-pong пакеты.
- Packet pacing в UDP простой, пока нет оптимизации, в TCP эта опция не работает.
- MTU и исправление ошибок и там, и там примерно сравнимы.
- По скорости TCP, конечно, быстрее, чем UDP сейчас, если вы раздаете тонну трафика. Но зато какие-то оптимизации очень долго доставляются.

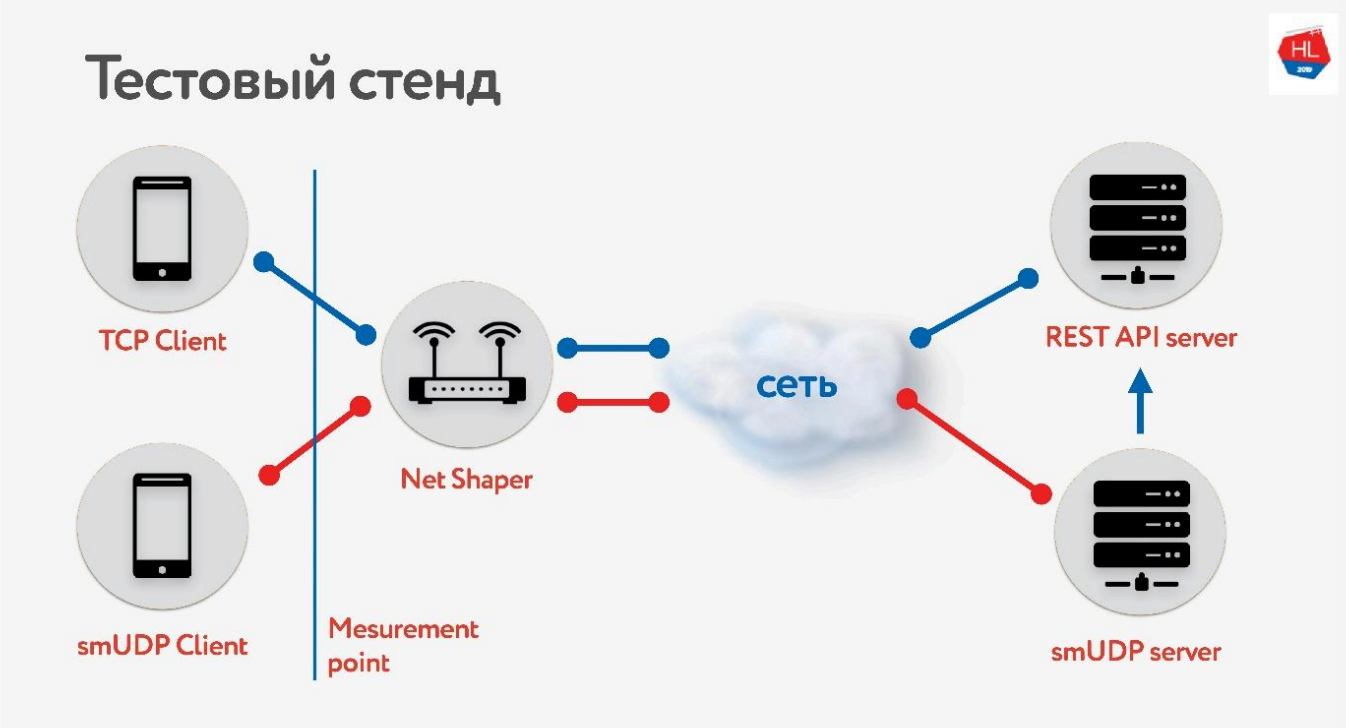
Если собрать все самое важное, то у UDP, скорее, больше плюсов, чем минусов.




**Выбираем UDP!**

## Тестирование self-made UDP на пользователях

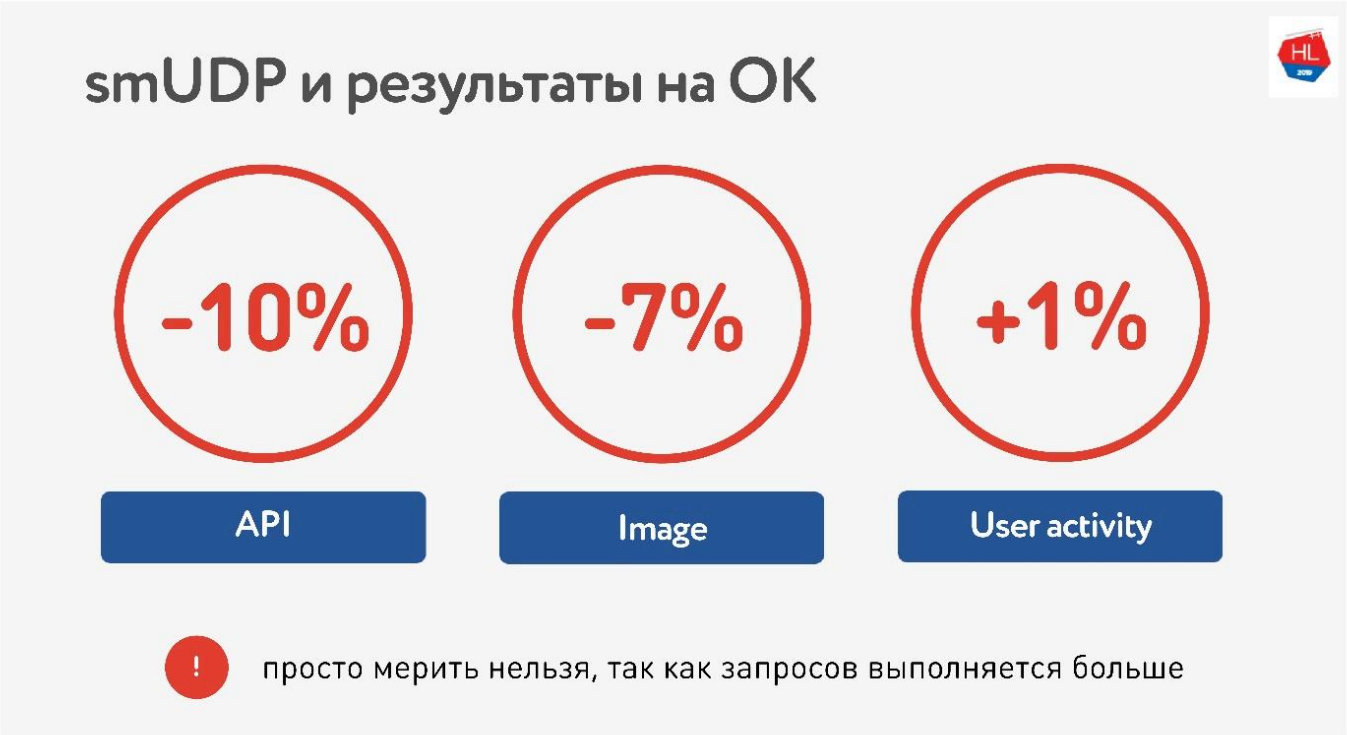
Мы собрали тестовый стенд.




Есть клиент на TCP и на UDP. Нормировали трафик через net shaper, отправили в интернет и на сервер. Один сервис REST API, второй с UDP. Причём UDP ходит на тот же REST API

внутри одного дата-центра, чтобы проверить данные. Собрали разные профили наших мобильных клиентов и запустили тест.


Измерив среднее по порталу, мы увидели, что мы смогли уменьшить время вызова API на 10%, картинки на 7%. User activity выросла всего-навсего на 1 %, но мы не сдаемся, думаем, что будет лучше.




По нагрузкам у нас сейчас порядка 10 млн пользователей на нашем self-made UDP, трафик

до 80 Гбит/с, 6 млн пакетов в секунду и 20 серверов все это обслуживают.

## UDP checklist

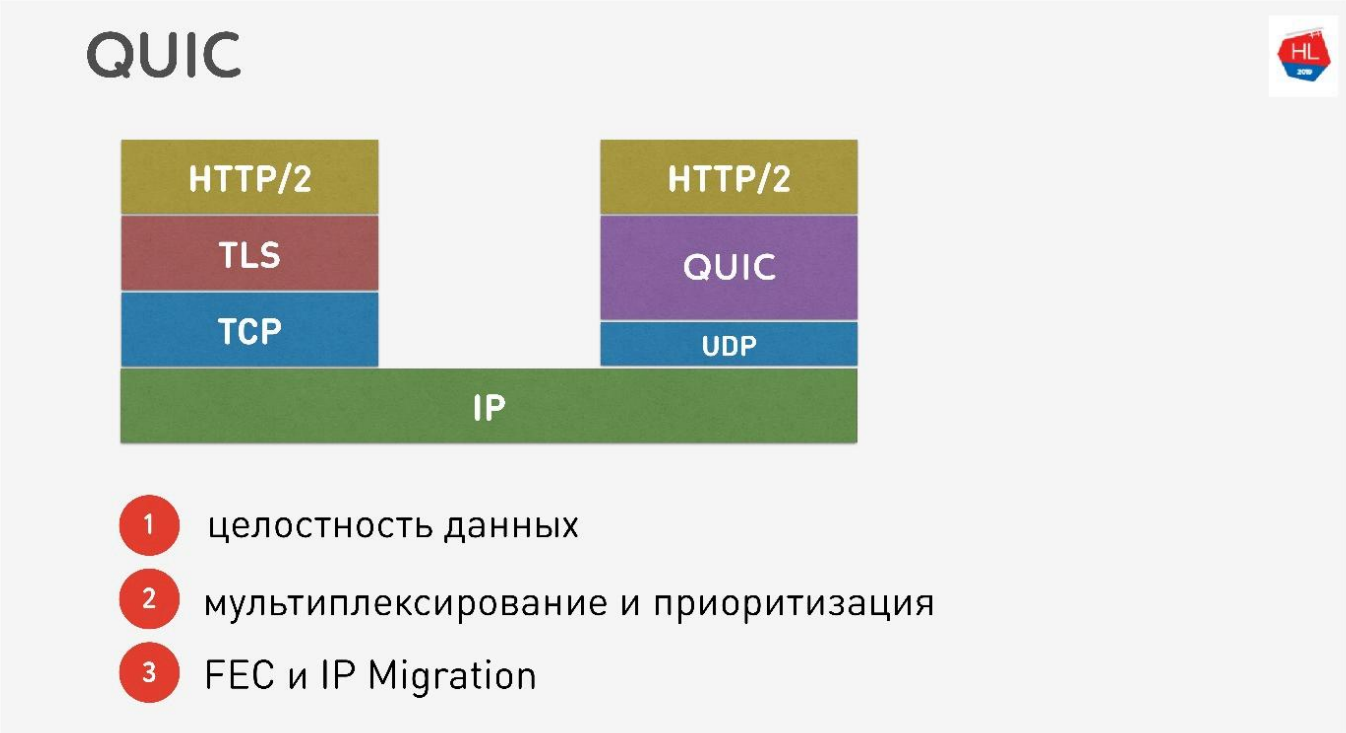
Если вы будете писать свой протокол, вам нужен чек лист:

- Pacing.
- MTU discovery.
- **Исправление ошибок обязательно**.
- Flow control и Congestion control.
- Опционально можете поддержать IP Migration, TLP — это легко.

Помните, что каналы асимметричны, и пока вы получаете данные от сервера, ваш upload может простаивать, пробуйте его использовать.

## QUIC

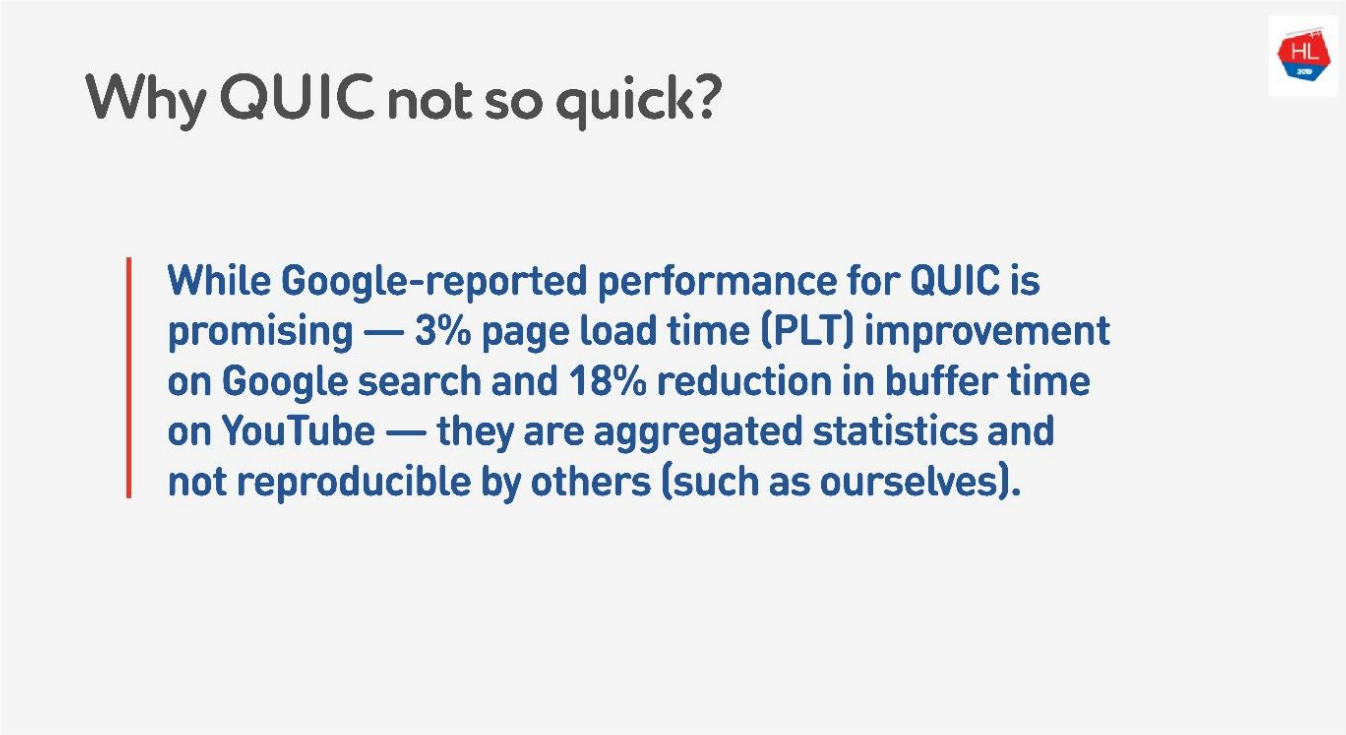
Было бы нечестно говорить, что Google такого не делал.




Есть протокол QUIC, который реализовал Google под интерфейсом HTTP 2.0, который поддерживает примерно то же самое.

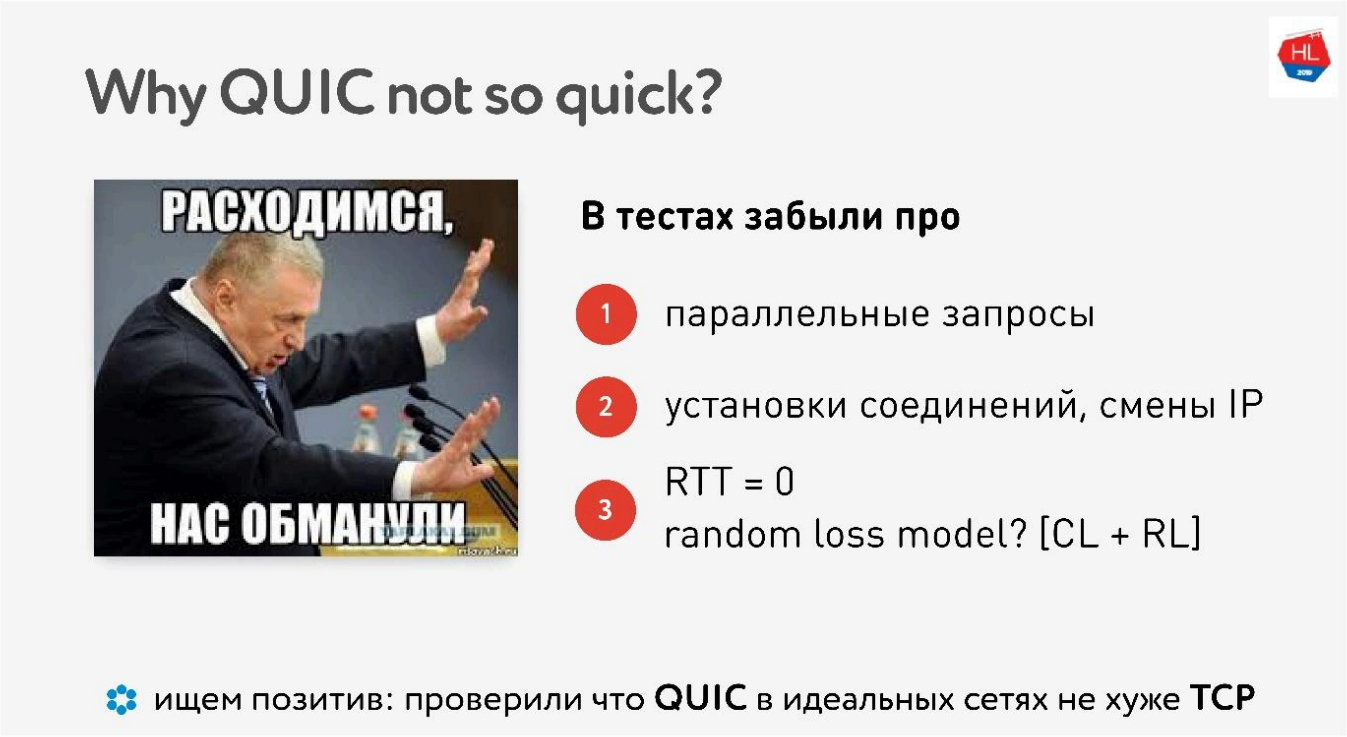
## Почему QUIC не так quick

Когда вышел QUIC, появилось очень много хейтинга по поводу того, что Google говорит, что все работает быстрее, а «я померял у себя дома на компьютере — работает медленнее».




В этой статье куча картинок и измерений.

Что же, получается, мы все это зря делали, люди померили за нас? Есть реальные домашние измерения, даже с примерами кода.

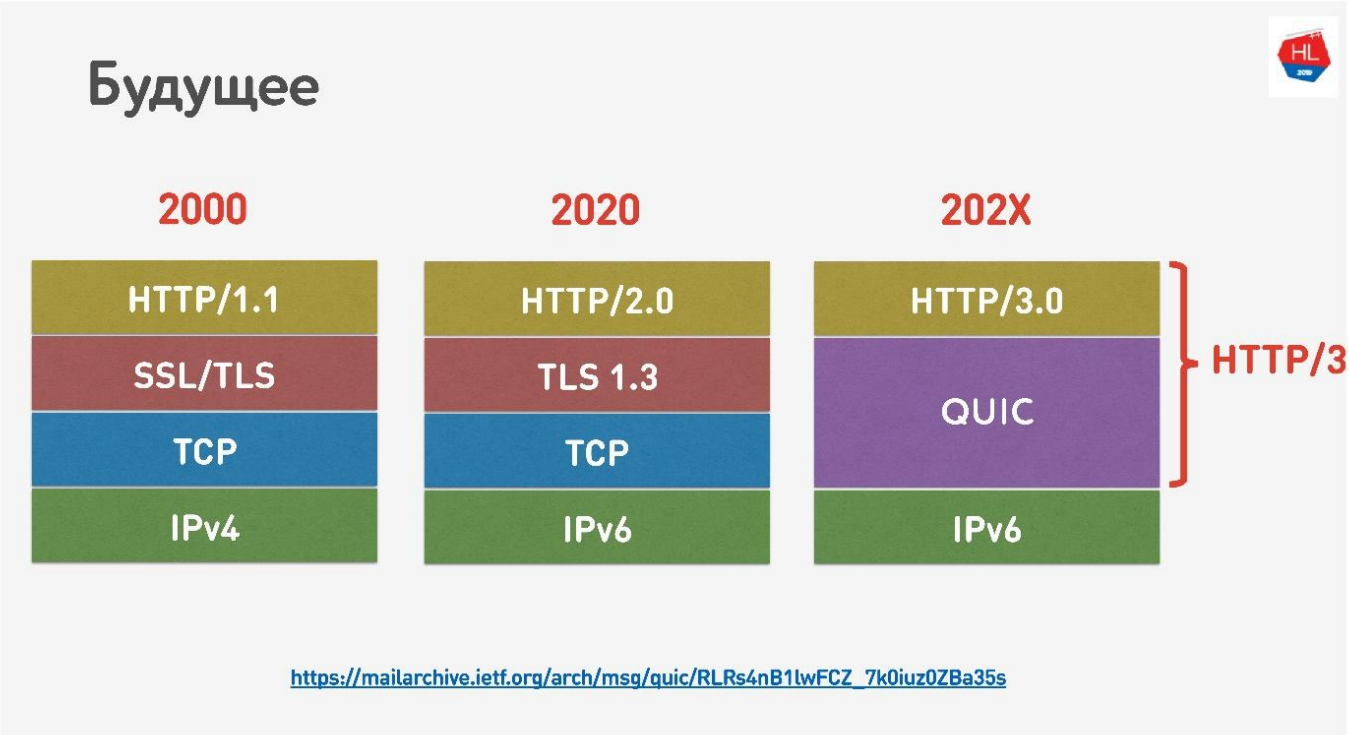



На самом деле улучшений не будет до тех пор, пока вы не будете параллелить запросы, работать в реальных сетях, и пока потери пакетов не будут делиться на congestion loss и random loss. Нужна реальная эмуляция реальной сети.

Но есть и позитив, говорят, QUIC не лучше и не хуже. Таким образом в идеальных сетях QUIC работает хорошо.

## Будущее

Недавно Google назвал версию HTTP 2.0 поверх QUIC HTTP 3, чтобы не путаться, потому что HTTP 2.0 мог быть поверх TCP и поверх QUIC. Теперь он HTTP 3.




Был еще Google QUIC — это QUIC, который реализован в Chrome, и iQUIC — стандартизованный QUIC. Стандартизованный QUIC по факту нигде не имплементировался, стандартные серверы iQUIC не хэндшейкались с Google QUIC. Сейчас они обещают эту проблему решить, и скоро это будет доступно.

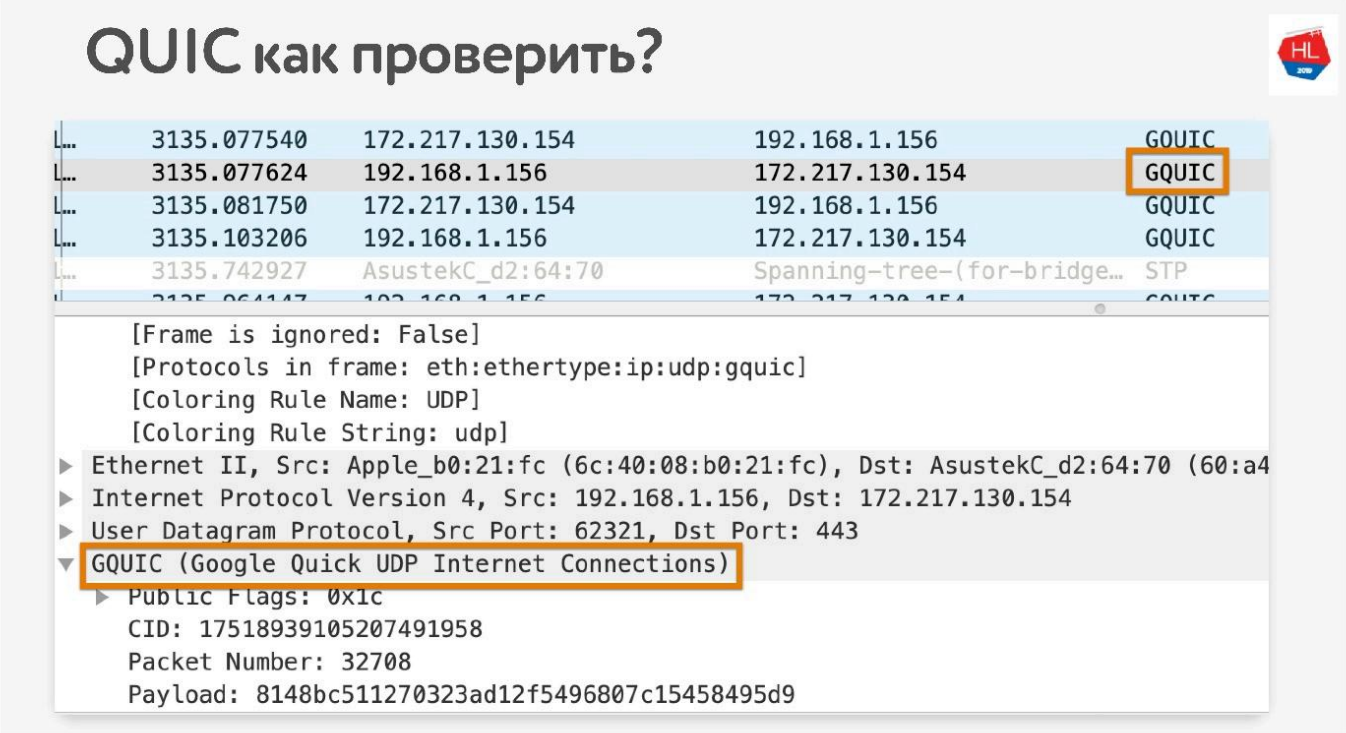
## QUIC повсюду

Если вы еще не верите, что TCP умер, то я вам скажу, что когда вы используете Chrome, Android, а скоро и iOS, и ходите в google, youtube и прочее, то используете QUIC и UDP (пруфлинк).

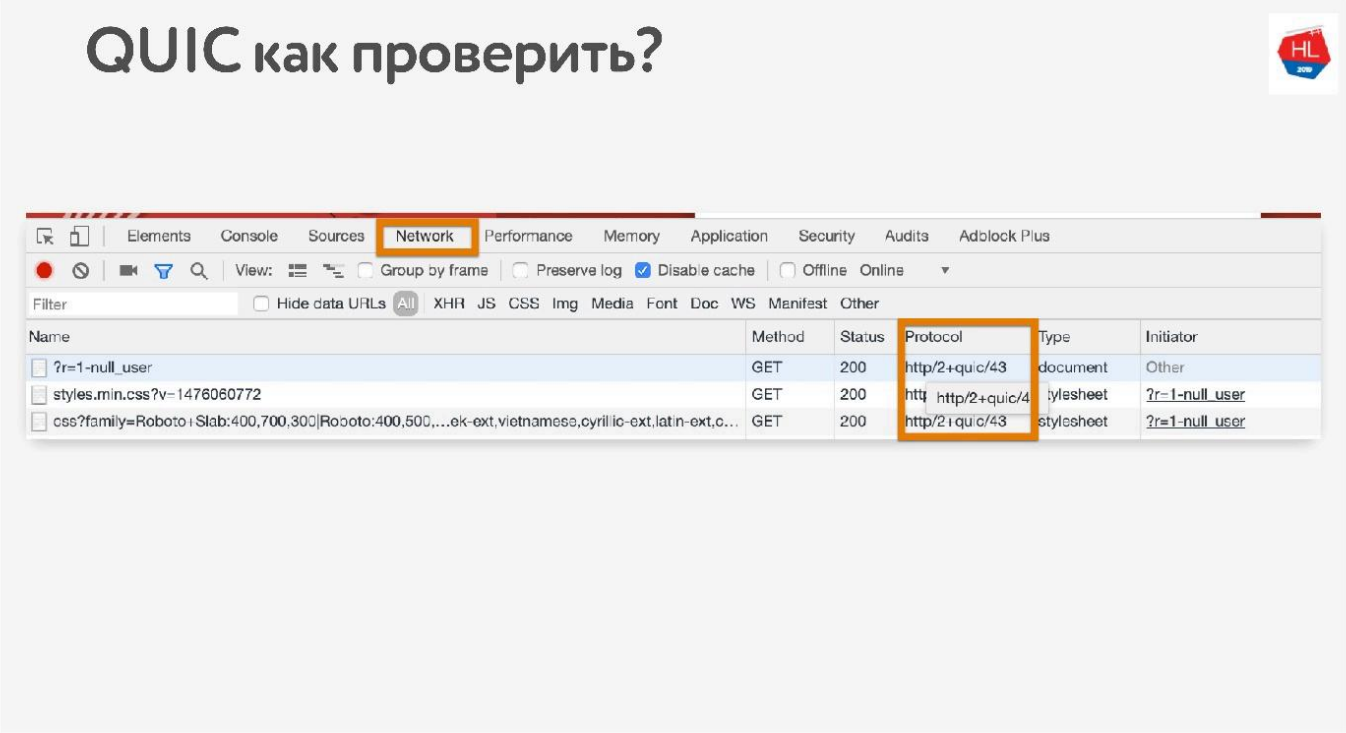
QUIC сейчас — это:

- 1,9 % всех вебсайтов;
- 12 % всего трафика;
- 30 % видео трафика в мобильных сетях.

Как проверить, что вы используете QUIC, если не верите? Откройте в Chrome Wireshark. Я искал iQUIC, нигде не нашел, но GQUIC бывает.

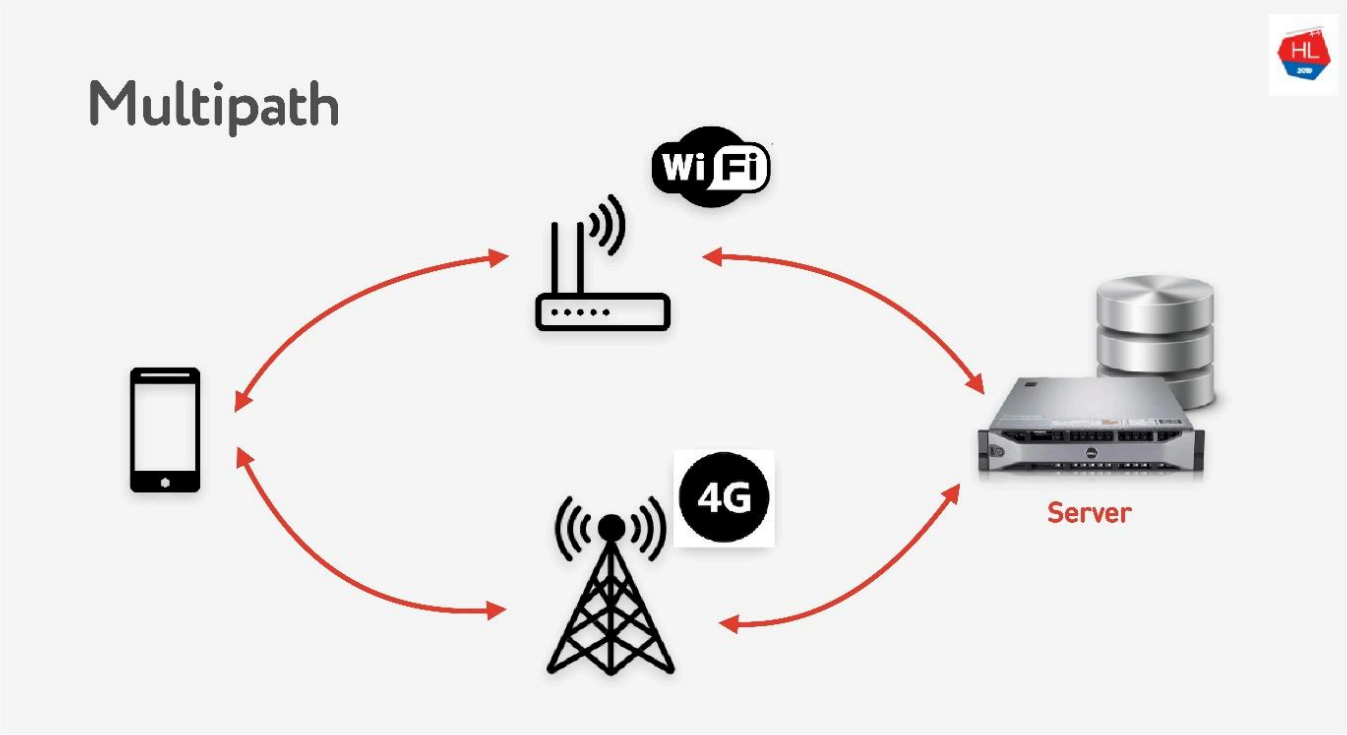



Также можно зайти в сеть в браузере и тоже увидеть, что там есть GQUIC.

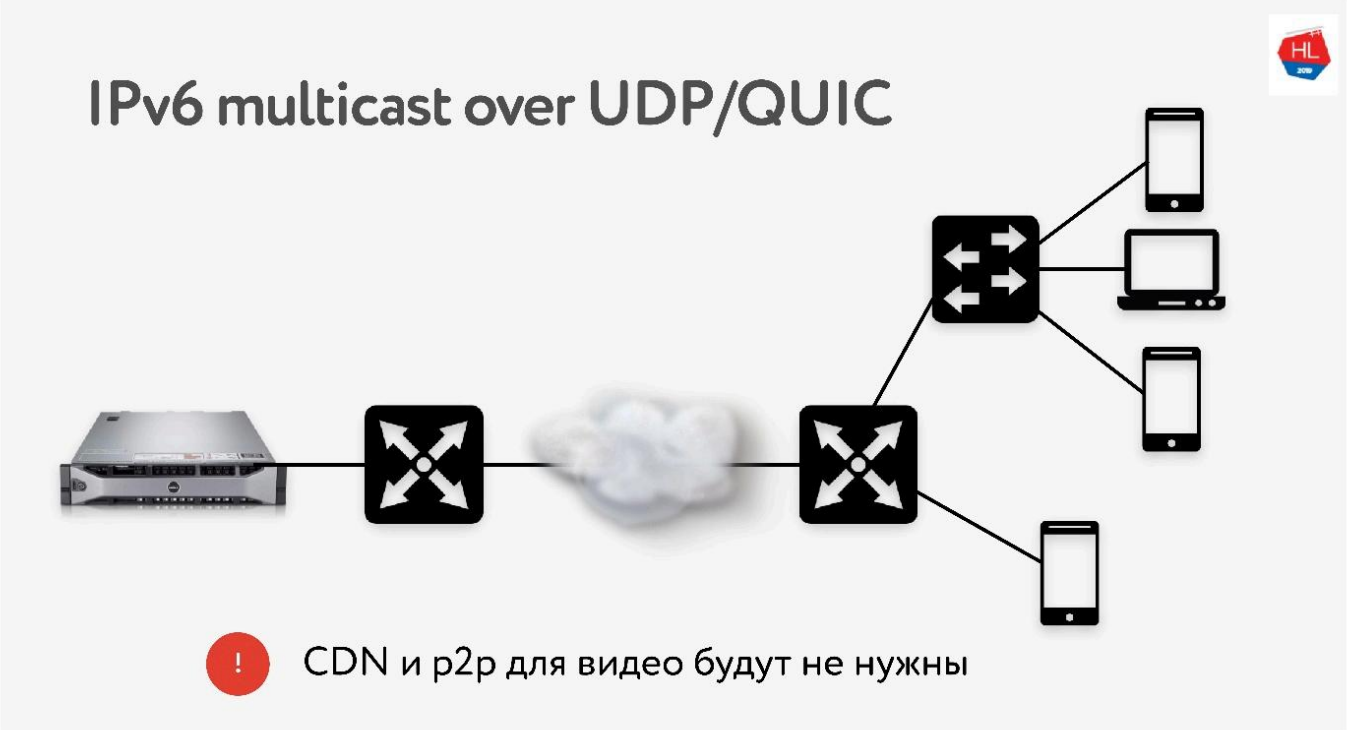



Ещё немного будущего

Скоро нас ждёт multipath.




Когда у вас есть мобильный клиент, у которого есть и Wi-Fi, и 3G, вы можете использовать оба канала. Multipath TCP сейчас в разработке, скоро будет доступен в ядре Linux. Очевидно, что до клиентов он дойдет нескоро, думаю, на UDP его можно сделать гораздо быстрее.




Так как мы проводим массу трансляций объемом по 3 Тб, мы очень часто используем такие технологии как CDN и p2p раздача, когда один и тот же контент нужно доставить многим пользователям по всему миру.

В IPv6 есть multicast с UDP, который позволит доставлять пакеты сразу нескольким подписавшимся пользователям. Поэтому я думаю, что технологии CDN и p2p в скором будущем будут не нужны, если мы будем доставлять весь контент с использованием multicast на IPv6.

## Выводы

Надеюсь, что вам стало понятнее:

- Как реально работает сеть, и что TCP можно повторить поверх UDP и сделать лучше.
- Что TCP не так плох, если его правильно настроить, но он реально сдался и больше уже почти не развивается.
- Не верьте хейтерам UDP, которые говорят, что в user space работать не будет. Все эти проблемы можно решить. Пробуйте — это ближайшее будущее.
- Если не верите, то сеть можно и нужно трогать руками. Я показывал, как почти все можно проверить.

Вы всё прочитали и разобрались, что дальше?

- Настраивайте протокол (TCP, UDP — неважно) под ситуацию (профиль сети + профиль нагрузки).
- Используйте рецепты TCP, которые я вам рассказал: TFO, send/recv buffer, TLS1.3, CC...
- Делайте свои UDP-протоколы, если есть ресурсы.
- Если сделали свой UDP, проверьте UDP check list, что вы сделали все, что надо. Забудете какую-нибудь ерунду типа pacing, не будет работать.

Если у вас нет ресурсов, готовьте свою инфраструктуру под QUIC. Он рано или поздно к вам придет.

> Мы с вами определяем будущее. То, какими протоколами пользоваться, решаем мы сами. Хотите использовать QUIC — используйте, хотите свое UDP или остаться на TCP — определяйте будущее сами.

## Полезные ссылки

- Миллион видеозвонков в сутки или «Позвони маме!».

- Пишем свой протокол поверх UDP.
- Подкаст про сетевую оптимизацию.
- Увеличение скорости передачи данных в плохих сетях.

До 7 сентября на московский HighLoad++ еще можно подать заявку и поделиться, а как вы готовите свои сервисы для высоких нагрузок. Но программа уже постепенно наполняется, от Одноклассников приняты доклады о новой архитектуре графа друзей, об оптимизации сервиса подарочков под высокие нагрузки и о том, что делать, если вы все оптимизировали, а данные до пользователя доходят недостаточно быстро.

Только зарегистрированные пользователи могут участвовать в опросе. Войдите, пожалуйста.

**Как вы думаете — будущее за UDP?**