92,23
Рейтинг

Одноклассники
Делимся экспертизой

alatobol 17 сентября 2019 в 09:43

Смотри меня полностью: выжимаем максимум из live video на мобильных платформах

Блог компании Одноклассники, Разработка под iOS, Разработка мобильных приложений, Разработка под Android

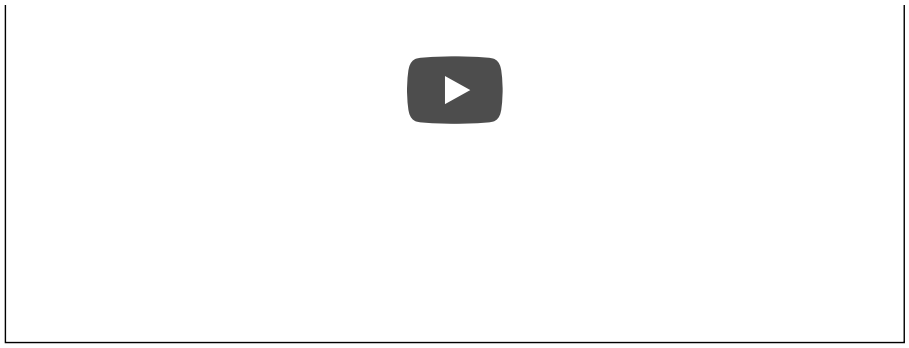
Самый простой способ воспроизвести видео на мобильном устройстве — это открыть ссылку имеющимся в системе плеером, но это не всегда эффективно.

Можно взять ExoPlayer и оптимизировать его, а можно вообще написать свой видеоплеер, используя только кодеки и сокеты. В статье будет рассказано о работе стриминга и воспроизведения видео, и о том, как уменьшить задержку старта видео, снизить время отклика между стримером и зрителем, оптимизировать энергопотребление и нагрузку на железо.

Разберём это на примере конкретных приложений: мобильного клиента «Одноклассников» (где видео воспроизводят) и OK Live (где трансляции стримят с телефона в 1080p). Здесь не будет мастер-классов о том, как по ссылке проиграть видео, с примерами кода. Рассказ пойдёт о том, как видео выглядит изнутри, и как, зная общую архитектуру видеоплееров и видеостриминга, можно разобраться в любой системе и сделать её лучше.

В основе материала — расшифровка доклада **Александра Тоболя**(@alatobol) и **Ивана Григорьева**(@ivan_a) с конференции **Mobius**.

Александр Тоболь, Иван Григорьев — Стриминг и воспроизведе...

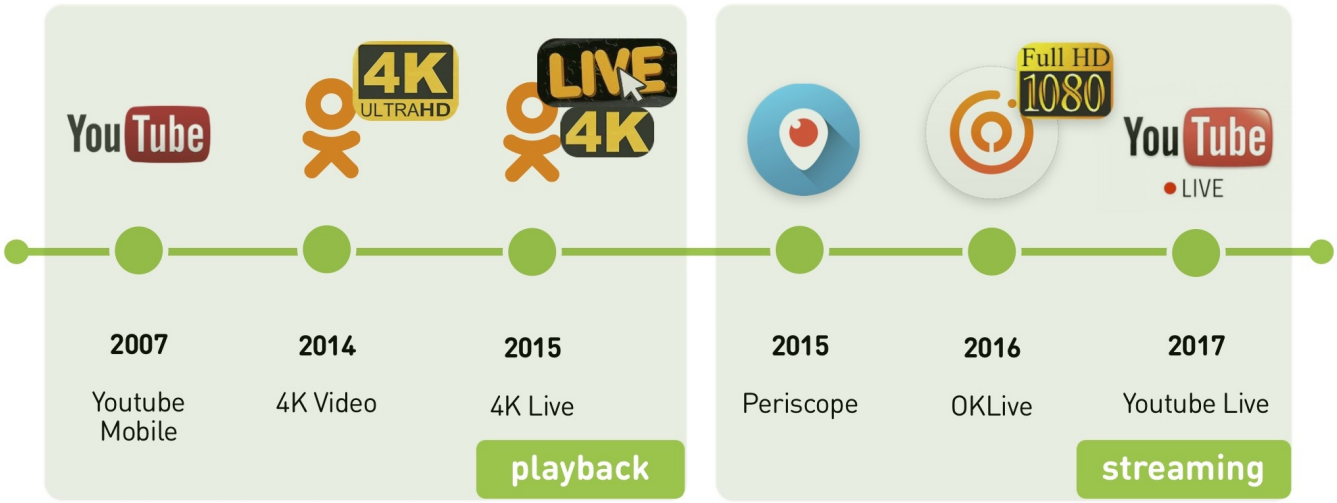


Вступление

Для начала — немного цифр про видео в Одноклассниках.

Пиковый среднедневной трафик VOD (видео по запросу) больше полутора терабит в секунду, а у Live-трансляций — больше 3 терабит в секунду.

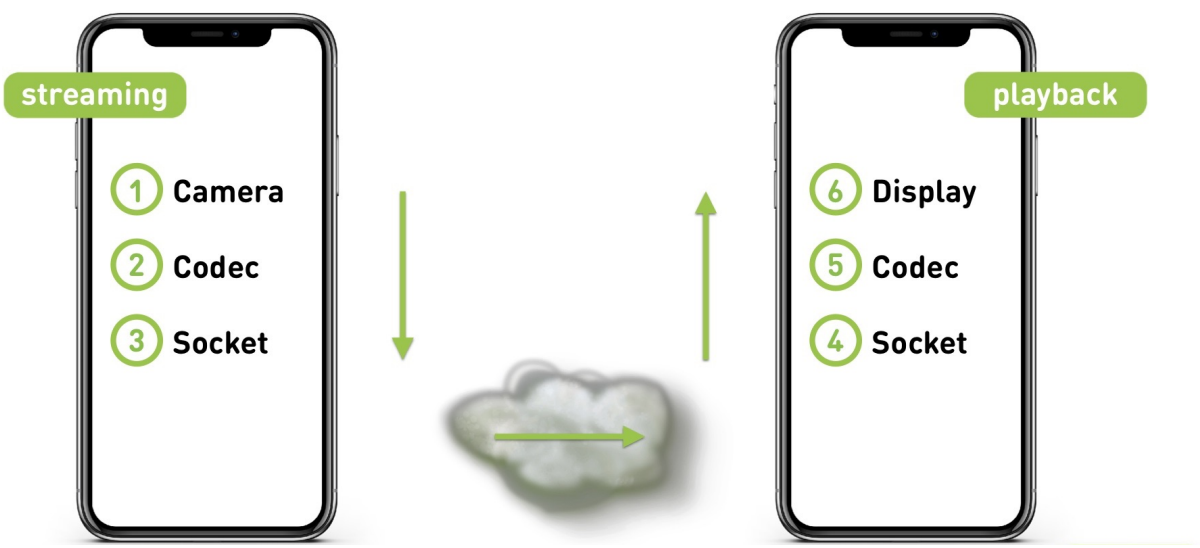
Сейчас в OK больше 870 миллионов просмотров видео в сутки, больше половины из которых — с мобильных устройств.



Если посмотреть на историю стриминга, то мобильное видео появилось у YouTube в 2007 году. Мы в этот поезд запрыгивали позже, но в 2014-2015 годах у нас уже было воспроизведение 4K-видео на мобильных устройствах, и в последние годы мы активно развивали наши плееры. Про это и пойдёт разговор.

Второй тренд, возникший с Periscope в 2015-м — трансляции с телефонов. Мы запустили свое приложение OK Live, которое позволяет стримить даже Full HD-видео через мобильные сети. Во второй половине материала о стриминге тоже поговорим.

Не будем останавливаться на API для работы с видео, а сразу нырнём глубоко и попробуем узнать, что происходит внутри.



Когда вы снимаете видео на камеру, оно попадает в кодек, оттуда в сокет, затем на сервер (вне зависимости от того, VOD или Live). А дальше сервер в обратном порядке раздаёт его зрителям.

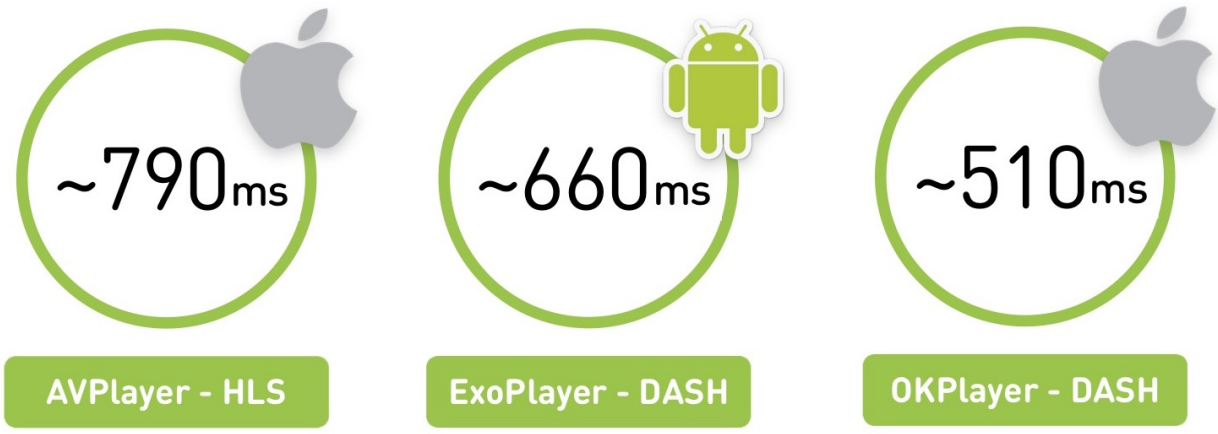
Начнём с KPI плеера. Что мы от него хотим?

- Быстрый первый кадр. Пользователи не хотят ждать начала воспроизведения.
- Отсутствие буферизаций. Никто не любит сталкиваться с «крутилкой».
- Высокое качество. Когда 4K-контента ещё почти не было, мы уже сделали поддержку 4K «на вырост»: если отдебажить плеер для неё и разобраться с производительностью, то 1080p даже на слабых устройствах будет играть прекрасно.
- Требования по UX. Нам нужно, чтобы в ленте видео играло во время скроллинга, и ещё для ленты нужен префетчинг видео.

На пути этого встаёт много проблем. Поток для 4K-видео большой, а мы работаем на мобильных устройствах, где есть проблемы с сетью, есть различные особенности форматов и контейнеров видео на разных устройствах, а сами устройства тоже могут стать проблемной штукой.

Как думаете, где быстрее стартует видео, на iOS или Android?

На самом деле, любой ответ — верный: смотря что, где и как играть. Если взять регион России с не очень хорошей сетью, то увидим, что AVPlayer стартует на примерно за 800 миллисекунд. Но при такой же сети ExoPlayer на Android, играющий другой формат, запустит его за 660 мс. А если на iOS сделать свой плеер, то он сможет запускаться ещё быстрее.



Есть нюанс в том, что мы измеряем среднее по пользователям, а средняя мощность iOS-устройств выше, чем на Android.

Первая часть материала будет теоретической: мы узнаем, что такое видео и как выглядит архитектура любого Live-плеера.

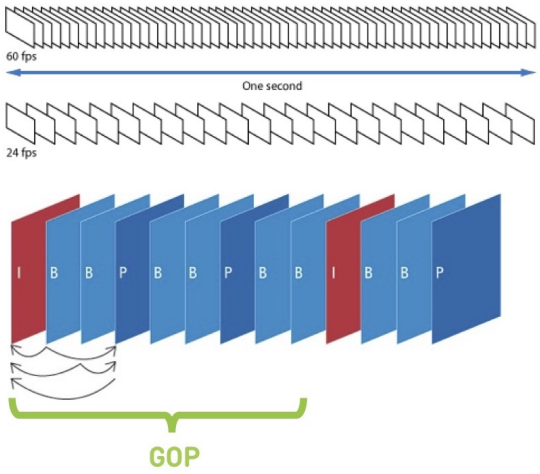
А во второй части сравним плееры и поговорим о том, когда стоит писать свой собственный.

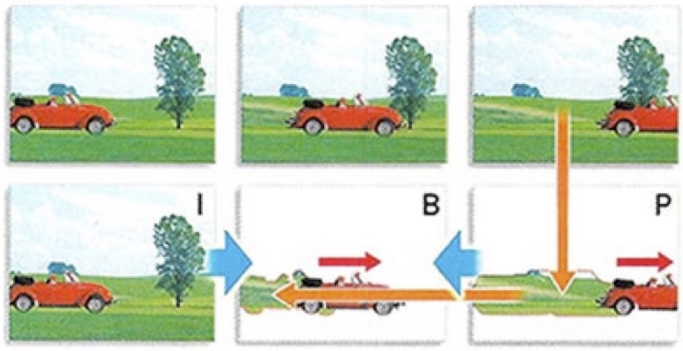
Часть первая

Что такое видео

Начнём с самого базового. Видео — это 60 или 24 картинки в секунду.

Очевидно, что хранить это полным набором картинок довольно дорого. Поэтому они хранятся таким образом: одни кадры называются опорными (I-frames), а другие (B-frames и P-frames) — это «диффы». Фактически, у вас есть JPG-файл и определённый набор изменений к нему.





Есть также понятие GOP (group of picture) — это независимый набор кадров, который начинается с опорного и продолжается набором диффов. Его можно независимо проиграть, распаковать и так далее. При этом, если вы в группе потеряли опорник, остальные кадры там уже не имеют значения.

Алгоритмов кодирования, матриц трансформации, поиска движения и тому подобного очень много — этим всем и отличаются кодеки.

Эффективность кодеков

Codec	Year	Efficiency	Server	Android		iOS	
				decoder	encoder	decoder	encoder
h264	2003	baseline	GPU, CPU	HW	HW 3.0+	HW	
vp8	2010	0 %	CPU	HW/SW	SW 4.3+	SW	
h265/HEVC	2013	50 %	GPU, CPU	5.0+	-	SoC, iOS11 (faceTime)	
vp9	2013		CPU, GPU dec	Snapd 835, Exynos 8		-	
AV1	2018	35 %	HW 2020	just Web: Chrome, Firefox			
H.266/JVEC	2021	50 %					

Классический H.264 известен с 2003 года и неплохо развивался. Его эффективность примем за базовую. Он работает и играет везде. У него есть аппаратная поддержка CPU/GPU (что на iOS, что на Android). Это означает, что есть либо какой-то специальный сопроцессор, который умеет его кодировать, либо встроенные наборы команд, позволяющие быстро это делать. В среднем аппаратная поддержка обеспечивает увеличение производительности до 10 раз и экономит батарейку.

В 2010 году появился VP8 от Google. По эффективности не отличается от H.264. Ну, на самом деле эффективность кодека —

очень спорная штука. В лоб она измеряется как соотношение исходного видео к сжатому, но понятно, что есть разные артефакты видео. Поэтому приводим ссылку на подробные сравнения кодеков от МГУ. А здесь ограничимся тем, что VP8 ориентирован на software-организацию, его можно притащить с собой куда угодно, и обычно он используется как запасной вариант, если нет нативной поддержки H.264.

В 2013 году появилось новое поколение кодеков — H.265 (HEVC) и VP9. Кодек H.265 даёт прирост эффективности в 50%, но на Android видео им нельзя закодировать, декодер появился только с Android 5.0+. А вот на iOS поддержка есть.

Существует альтернатива H.265 — VP9. Всё то же самое, но поддерживается компанией Google. Ну, V9 — это YouTube, а H.265 — это Netflix. Так что у каждого свои особенности: один не будет работать на iOS, у другого будут проблемы на Android. А в итоге многие остаются на H.264.

В будущем нам обещают кодек AV1, у него уже есть софтверная реализация, и его эффективность на 35% выше, чем у кодеков 2013 года. Сейчас доступен в Chrome и Firefox, а в 2020 году Google обещает hardware-поддержку — думаю, скорее всего, все мы переедем на него.

Наконец, недавно анонсировали кодек H.266/JVEC, заявив, что всё будет лучше и быстрее.

Главная закономерность: чем выше эффективность кодека, тем больше вычислительных ресурсов он требует от устройств.

В общем, по умолчанию все берут H.264, а дальше для конкретных устройств можно усложнять.

Качество, разрешение и битрейт

В 2019-м никого не удивишь адаптивным качеством: пользователи загружают или стримят видео в одном качестве, а мы нарезаем линейку разных качеств и отправляем на устройства самое подходящее.

При этом нужно, чтобы разрешение видео соотносилось с битрейтом. Если удваивается разрешение — должен удваиваться и битрейт:

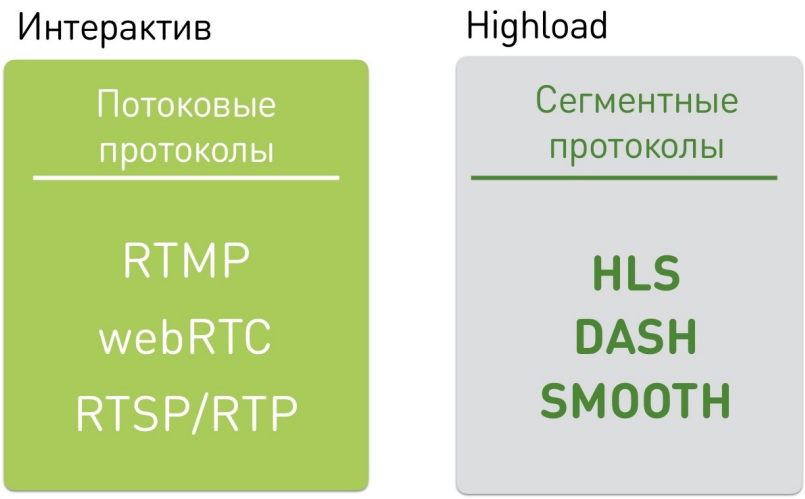
Quality	Resolution	Bitrate
4K	3840x2160	30-50 Mbit/sec
QUAD HD	2560x1440	10 Mbit/sec
FullHD	1920x1080	6 Mbit/sec
HD	1280x720	2.5 Mbit/sec
SD	854x480	1.2 Mbit/sec
medium	640x360	600 Kbit/sec
low	426x240	300 Kbit/sec

Очевидно, что если сжать большое разрешение низким битрейтом или наоборот, то будут либо артефакты, либо бесполезное прожигание битрейта.

Как соотносится битрейт закодированного видео с исходным объёмом информации? На 4K-экране мы можем проиграть почти 6 Гбит/с информации (если посчитать все пиксели и частоту их смены при 60 кадрах в секунду), при этом битрейт кодека может составлять 50 МБит/сек. То есть кодек сжимает видео до 100 раз.

Технологии доставки

У вас есть аудио и видео, запакованные какими-то кодеками. Если вы это просто храните у себя — можно сложить всё аудио и видео, добавив небольшой индекс, который сообщает, с какой секунды начинаются аудио и видео. Но так видео не доставить на телефон, и для стриминга зрителю в онлайне существуют два основных класса протоколов: потоковые и сегментные.



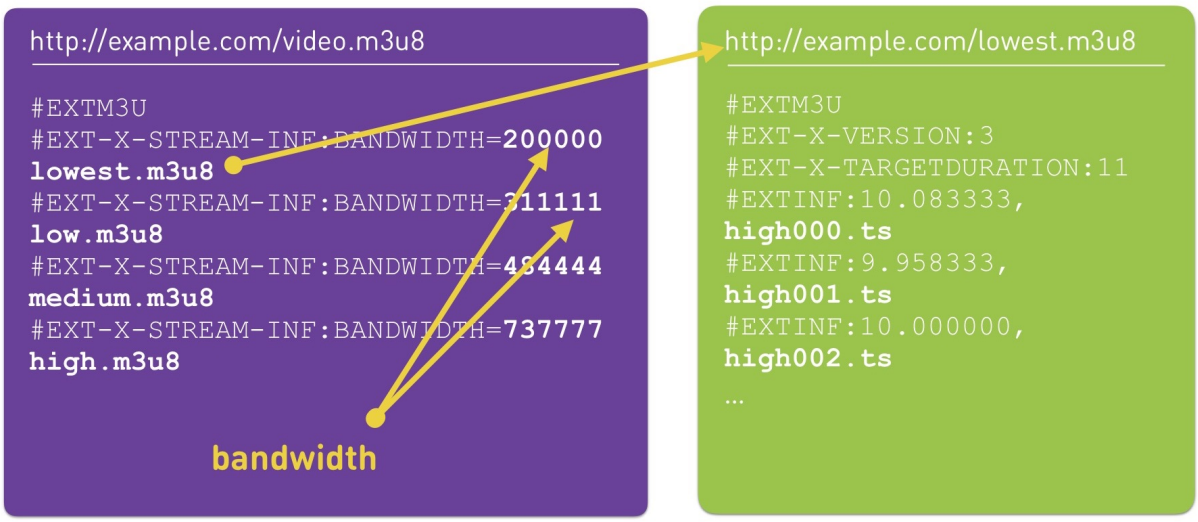
Потоковый протокол подразумевает, что у вас на сервере есть какой-то state, у клиента тоже, и он отправляет данные. Сервер может регулировать, например, качество. Очень часто это UDP-соединение.

Такие протоколы отличаются высокой сложностью для сервера и трудностью доставки. Для высоконагруженных трансляций мы используем сегментные протоколы, которые работают поверх HTTP, могут кэшироваться nginx и CDN, и их гораздо легче распространять. А сервер ни за что не отвечает и, в данном случае, stateless.

Как выглядит сегментная доставка: мы нарезаем имеющееся видео на сегменты, сопровождаем их заголовком для аудио и видео, в качестве примера транспорта могут быть MPEG-TS и MP4. На телефон отдаём манифест с информацией, где и для какого качества лежит сегмент, и этот манифест можно периодически обновлять.

Исторически сложилось, что Apple доставляет через HLS, а Android — через DASH. Посмотрим, чем они отличаются.

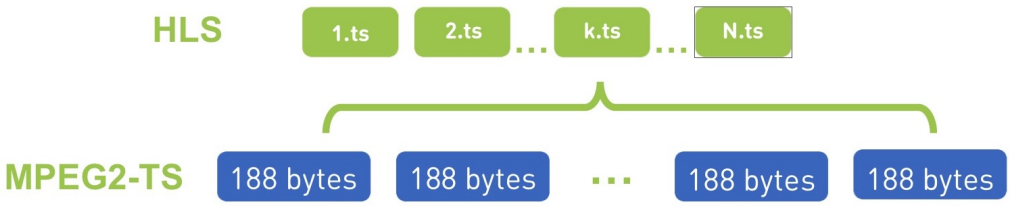
Начнем с более старого HLS, в нём есть манифест, где описаны все имеющиеся качества — low, medium, high и так далее. Есть битрейты этих качеств, чтобы плеер мог сразу выбрать подходящее. Он выбирает качество и получает вложенный манифест со списком ссылок на сегменты. Указана также продолжительность этих сегментов.



Здесь есть интересная особенность: чтобы начать играть первый кадр, придется сделать два дополнительных round trip-а. Первым запросом получаете основной манифест, вторым — вложенные манифесты, и только потом обращаетесь к самим данным, что не очень хорошо.

! лишний RTT(поход на сервер) на первом кадре

Вторая сложность: HLS проектировался для работы в интернете по HTTP, но в качестве контейнера для видео данных был выбран legacy MPEG-2 Transport Stream, который разрабатывался для совсем других целей: передачи сигнала со спутника в зашумленных каналах. В результате мы получаем дополнительные заголовки, которые в случае HLS совершенно бесполезны и только добавляют накладных расходов.

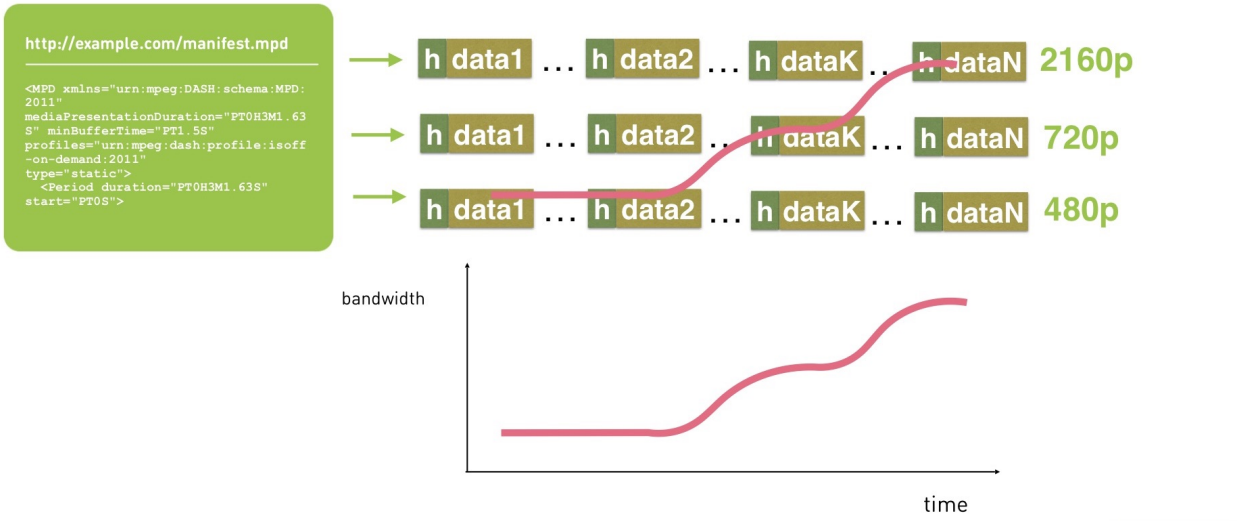


1 40Гбит/сек по 188байт пакет = 26 млн пакетов в сек

2 дорого парсить по CPU и до 15% network overhead

Добавьте сетевой overhead и сложность по парсингу: если вы пытаетесь играть 4K в DASH и HLS в Chrome, то почувствуете разницу, когда с HLS-пакетами компьютер будет «взлетать».

В Apple пытаются это решить. В 2016 году анонсировали возможность использования Fragmented MPEG-4, появилась некоторая поддержка DASH в HLS, но лишний RTT и его особенности никуда не делись.



DASH выглядит чуть проще: у вас есть один манифест со всеми качествами внутри, и каждое качество представляет собой набор сегментов. Можно воспроизводить один сегмент воспроизводить в одном качестве, потом понять, что скорость выросла, со следующего сегмента переключиться на другое. Все сегменты всегда начинаются с опорных кадров, позволяя переключаться.

Вот небольшая табличка о том, из чего необходимо выбирать:

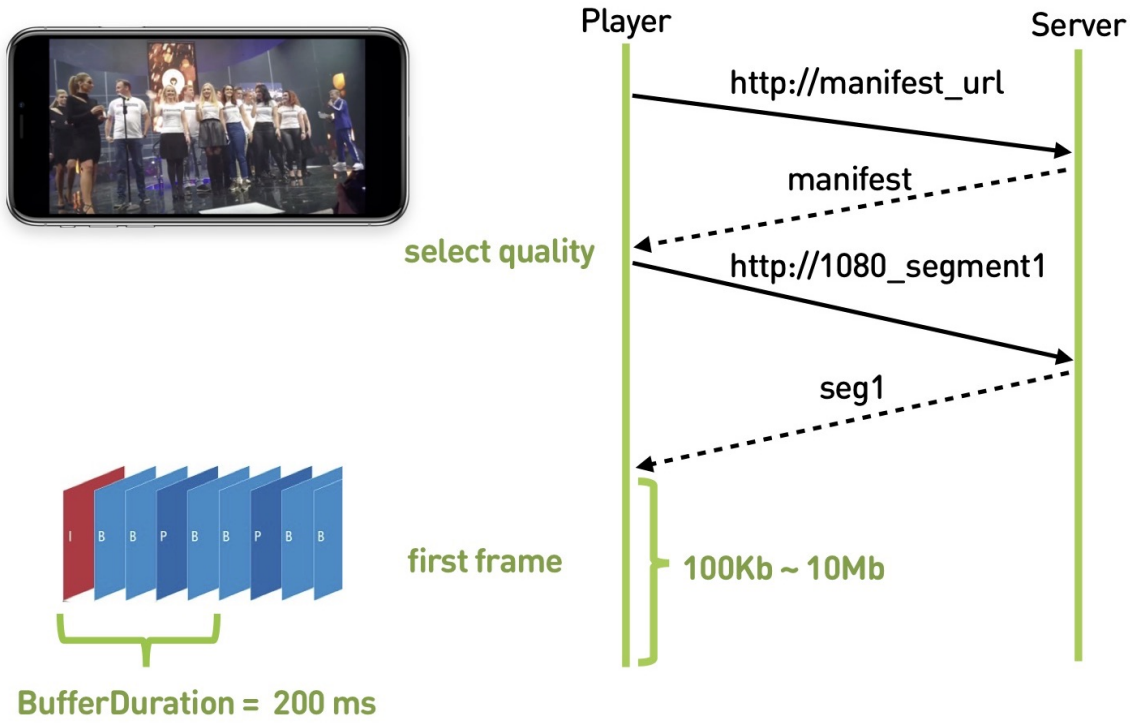
	HLS	MPEG-DASH
Кодек	h264,aac	любой
Транспорт	MPEG2-TS *	mp4, MPEG2-TS
Манифест	m3u8	mpd
RTT до данных	2	1
Размер сегмента	188 байт *	в зависимости от
OS	iOS, Android 4+	Android

* wwdc2016 HLS стал поддерживать fragmented mp4

В HLS исторически поддерживаемые видеокодеки — только H.264, в MPEG-DASH можно впихнуть любой. Основная проблема HLS — лишний round trip на старте, воспроизводится хорошо и на iOS, и на Android с 4.0. А DASH в основном поддерживается Google (Chrome и Android) и не воспроизводится на iOS.

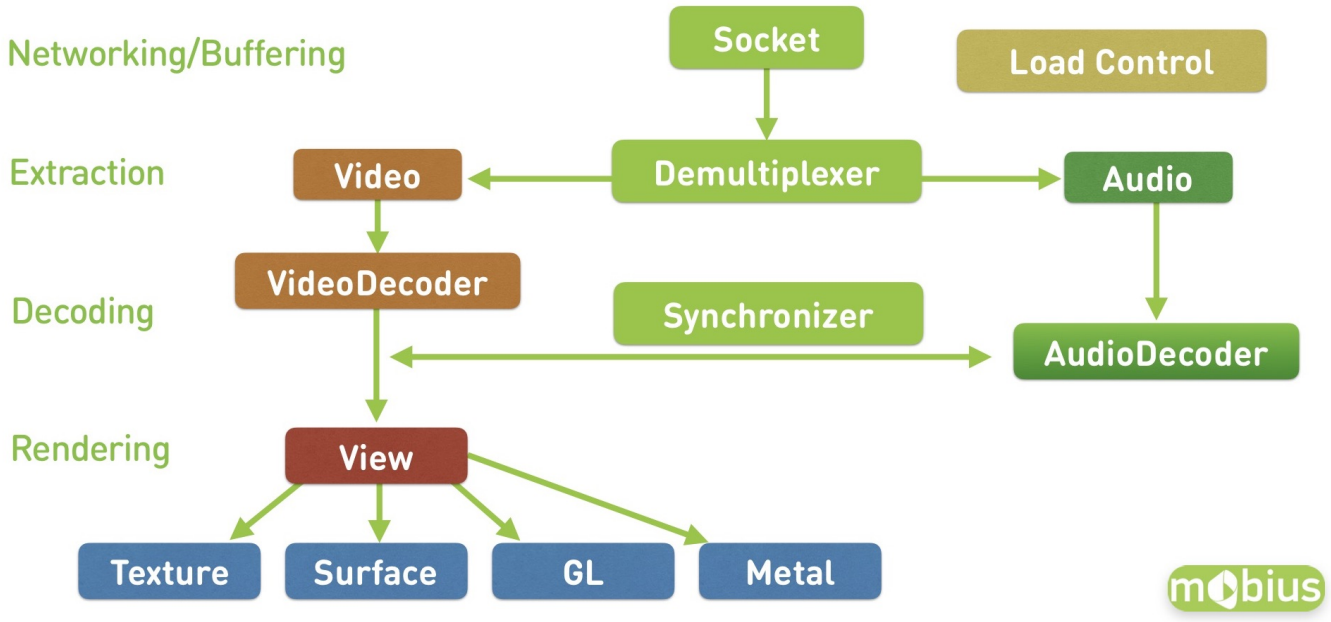
Архитектура плеера

С видео более-менее разобрались, теперь посмотрим, как выглядит любой плеер.



Начнём с сетевой части: при запуске видео плеер идёт за манифестом, как-то выбирает качество, затем идёт за сегментом, скачивает его, после этого должен раскодировать кадры, понять, что в буфере достаточно кадров для проигрывания, и затем начинает воспроизведение.

Общая архитектура плеера:



Есть сетевая часть, сокет, откуда приходят данные.

После этого — демультиплексер или какая-то штука, которая достаёт из транспорта (HLS/DASH) аудио- и видеопотоки. Она отправляет их соответствующим кодекам.

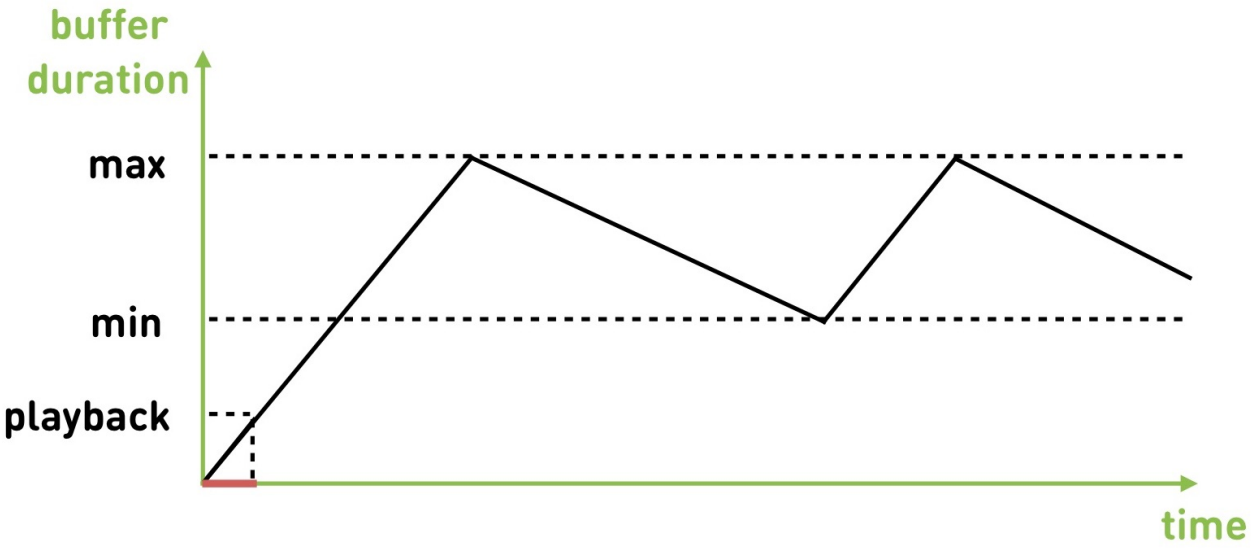
Кодеки раскодируют видео и аудио, а затем происходит самое интересное: их нужно синхронизировать, чтобы ваше видео и аудио играли одновременно. Для этого есть разные механизмы, основанные на timestamps.

Дальше нужно куда-то это зарендерить — в Texture, Surface, GL или Metal, куда угодно.

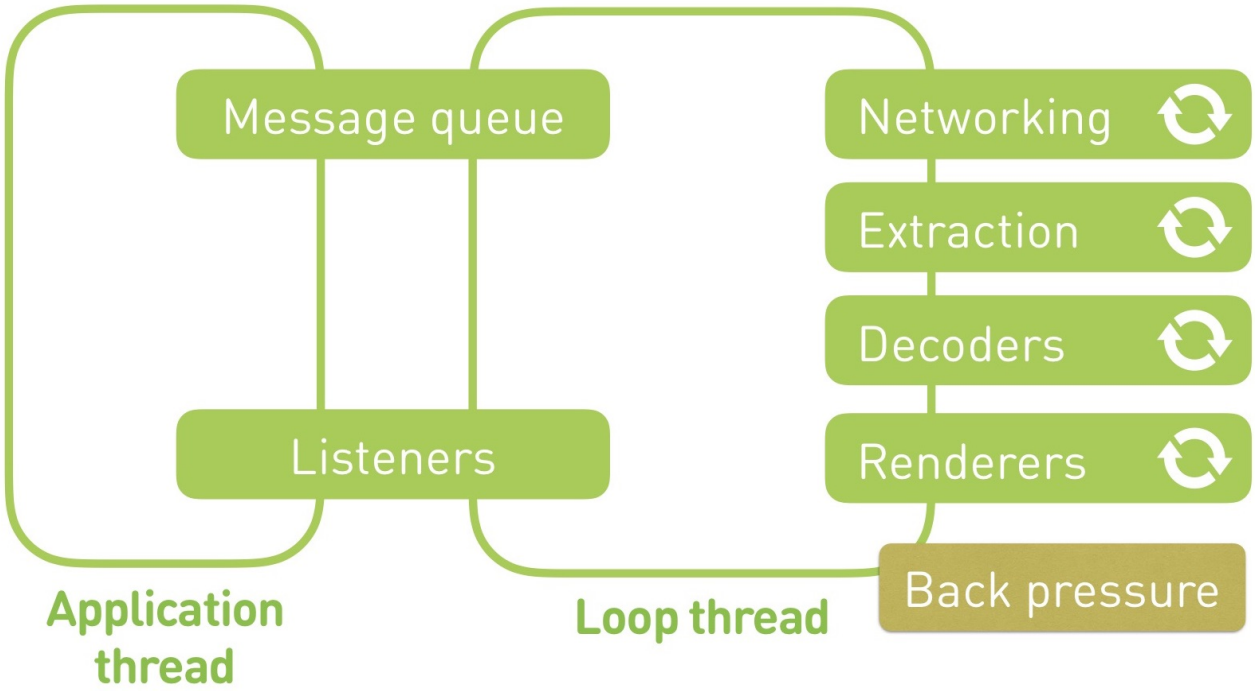
А на входе есть load control, который подгружает данные и управляет буфером.

Как выглядит load control во всех плеерах? Есть какой-то объём данных, которые нужно скачать. Плеер ждёт, пока их

скачает, затем начинает играть, а мы скачиваем дальше. У нас есть максимальная граница буфера, при достижении которой скачивание прекращается. После этого по мере воспроизведения объём данных в буфере падает — и есть минимальная граница, при которой начинает догрузку. Так всё это и живёт:



Как выглядит основной loop thread? Игровикам знакомо понятие “tick thread”, здесь похоже. Есть часть, отвечающая за сеть, которая складывает всё в один буфер. Есть экстрактор, который распаковывает и отправляет это в кодеки, где свой промежуточный буфер и дальше это отправится на рендеринг. И у вас есть tick, который перекладывает и контролирует их, занимается синхронизацией.

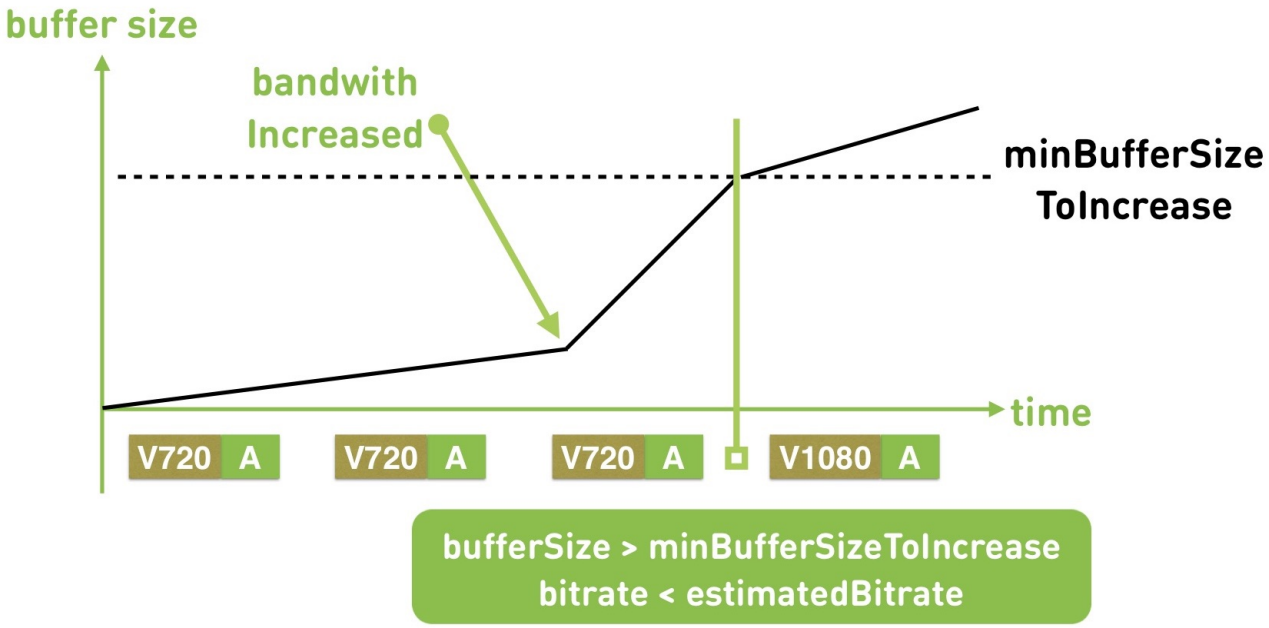


Снаружи у вас есть приложение, которое через message queue отправляет какие-то команды и получает какую-то информацию через listeners. И иногда может появляться back pressure, которое понижает качество — например, в ситуации, когда у вас заканчивается буфер или не справляется рендер (скажем, появляются дропфреймы).

Эстиматор

При адаптации плеер опирается на 2 основных параметра: скорость сети и запас данных в буфере.

Как это выглядит: сначала воспроизводится определённое качество, например, 720p. У вас вырастает буфер, кешируется все больше и больше. Потом вырастает скорость, вы понимаете, что можете качать ещё больше, растёт буфер. И в этот момент вы понимаете, что наступаете на некоторые границы минимального буфера, когда можно попробовать следующее качество.



Понятно, что его нужно пробовать аккуратно: есть ещё эстиматор, который говорит, можете ли вы уложиться в это качество по скорости сети. Если вы укладываетесь в эту оценку и запас буфера позволяет, то вы переключаетесь, например, на 1080p и продолжаете играть.

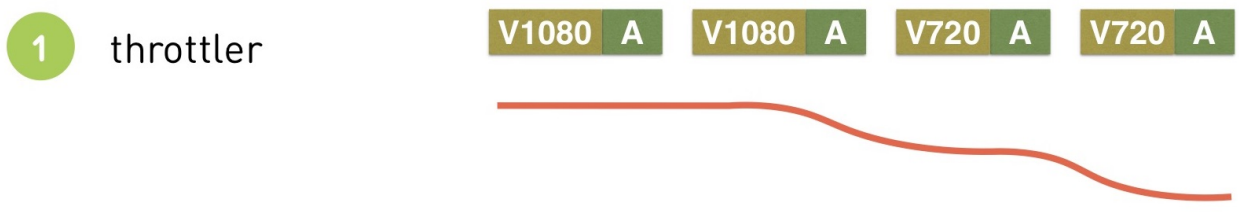
Защита от перегрузок (back pressure)

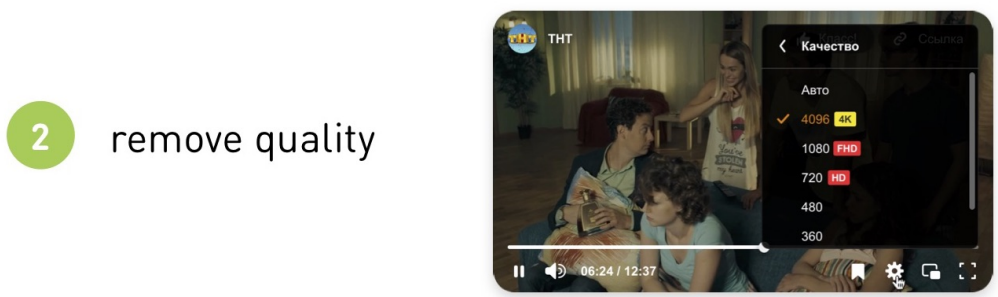
У нас она появилась со временем путём проб и ошибок. Необходимость в ней возникает, когда вы немного перегружаете своё оборудование.

Бывает ситуация, когда во время воспроизведения затупила сеть или на бэкенде кончились ресурсы. Когда плеер возобновляет воспроизведение, он начинает догонять.

В манифесте плеера к этому моменту скопился огромный набор сегментов, он быстро докачивает их все сразу, и получается некоторый «удар трафиком». Ситуация может усугубиться, если на клиентах происходит таймаут, и плеер начинает перезапрашивать данные. Поэтому нужно обязательно предусмотреть в системе back pressure.

Первый простой способ, который мы, конечно же, используем — throttler на сервере. Он понимает, что заканчивается трафик, снижает качество и специально затормаживает клиентов, чтобы не получился тот самый удар.





Но это не очень хорошо сказывается на эстиматорах. Они могут сгенерировать те самые «крутилки». Поэтому, если есть возможность, поддержите удаление качества из манифеста. Для этого нужно либо периодически обновлять манифест, либо

при наличии канала обратной связи дать команду убрать качество, и плеер автоматически переключится на другое, пониже.

Плееры

В iOS есть только нативный AVPlayer, а вот на Android возникает выбор. Есть нативный MediaPlayer, а есть опенсорсный ExoPlayer на Java, который приложения «приносят с собой». Какие у них плюсы и минусы?

Сравним все три:

	MediaPlayer	ExoPlayer	AVPlayer
framework/app level	Native	Java	Native
adaptive streaming	-	DASH/HLS	HLS
OS	API 1	API 16 (JB)	iOS 6
prefetching	-	doable	hardcore
bugs fixes	OS release	up Exo version	OS release
frame freeze	-	freeze, scroll	freeze, scroll
power efficient on audio	+	-	+

! на iOS нет аналога Exo

mobius

В случае с адаптивным стримингом ExoPlayer играет DASH/HLS и имеет множество расширяемых модулей под другие протоколы, а у AVPlayer всё хуже.

Поддержка версий операционных систем, в принципе, везде всех устраивает.

Prefetching — это когда вы знаете, что после окончания одного видео захотите играть в ленте следующее, и заранее подгружаете.

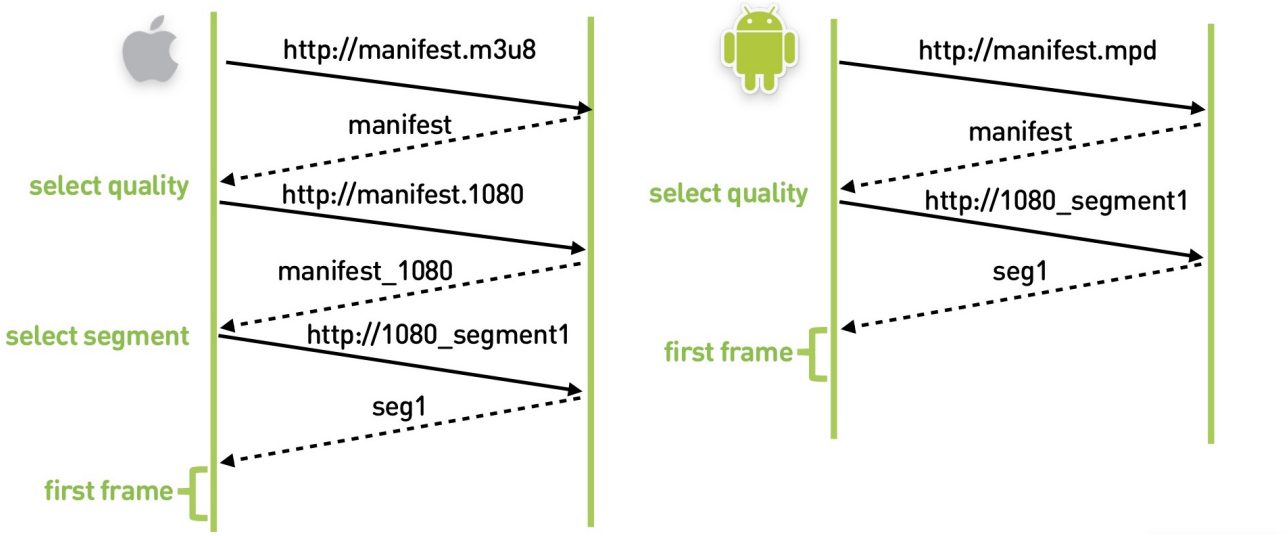
Есть проблема с багфиксами нативных плееров. В случае с ExoPlayer вы просто накатываете его в новую версию своего приложения, а вот в нативных AVPlayer и MediaPlayer баг будет исправлен только в следующем релизе ОС. Мы с этим больно сталкивались: в iOS 8.01 наше видео стало играть плохо, в iOS 8.02 перестал работать весь портал, в 8.03 всё снова заработало. И от нас в этом случае ничего не зависело, мы просто сидели и ждали, пока Apple выкатит следующую версию.

Команда ExoPlayer-а говорит про неэффективность энергопотребления в случае с аудио. Есть общие рекомендации от Google: для проигрывания аудио используйте MediaPlayer, для всего остального Exo.

Поняли, будем для видео на Android использовать ExoPLayer с DASH, а на iOS — AVPlayer с HLS.

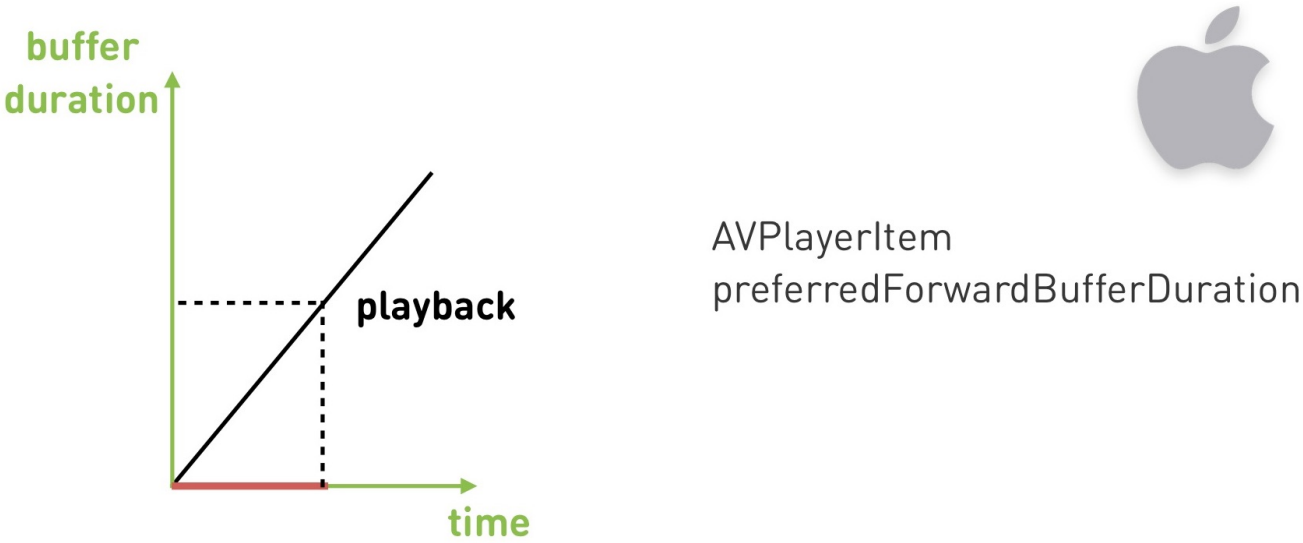
Быстрый первый кадр

Снова вспомним о времени до первого кадра. Как это выглядит на HLS iOS: первый RTT за манифестом, затем ещё один RTT за вложенным манифестом, только потом — получение сегмента и проигрывание. В Android на один RTT меньше, стартует чуть лучше.

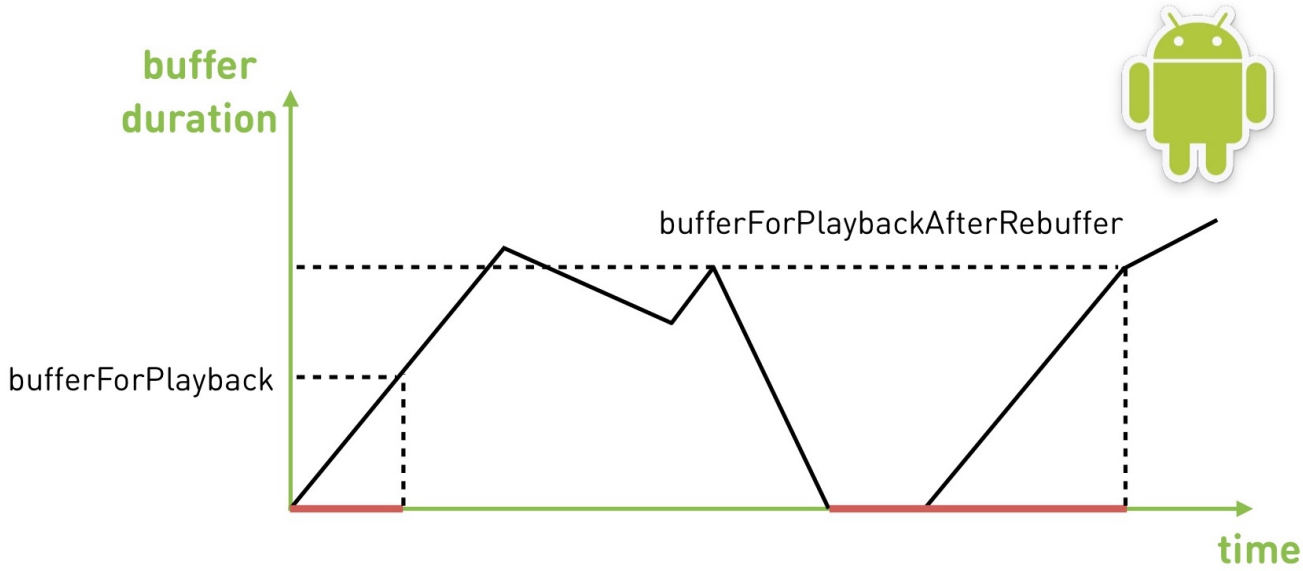


Размер буфера

Теперь давайте разберемся с буферами. У нас есть минимальный объём данных, которые нужно скачать перед тем, как мы начнём играть. В AVPlayer это значение настраивается с помощью AVPlayerItem preferredForwardBufferDuration.

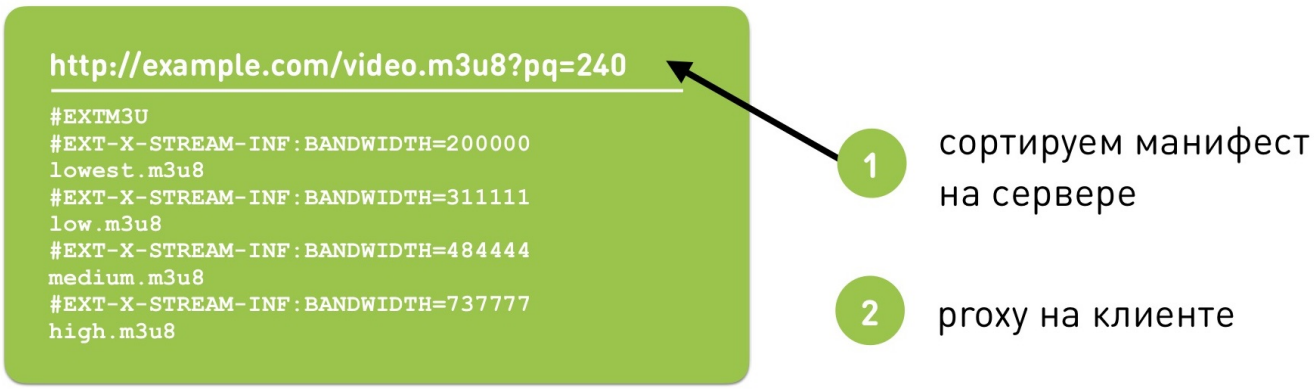


На Android в ExoPlayer механизмов для настройки гораздо больше. Есть такой же минимальный буфер, который нужен для старта. Но есть ещё и отдельная настройка для ребуферинга (если у вас отвалилась сеть, данные из буфера кончились, а затем она вернулась):



В чём профит? Если у вас хорошая сеть, вы быстро стартуете и боретесь за быстрый первый кадр, в первый раз можно попробовать рискнуть. Но если в процессе проигрывания сломалась сеть, очевидно, что на ребуферинге нужно попросить больше буферизации для проигрывания, чтобы не случилось повторной проблемы.

Изначальное качество



В HLS на iOS есть классная проблема: он всегда начинает играть с первого качества в m3u8-манифесте. Что ему отдадите, с того и начнёт. А уже потом измерит скорость скачивания и начнёт воспроизводить в нормальном качестве. Понятно, что допускать такое не стоит.

Логичная оптимизация — пересортировать качество. Или на сервере (добавив дополнительный параметр в preferredquality, он пересортировывает манифест), или на клиенте (сделать proxy, который будет это делать за вас).

А на Android есть для этого параметр DefaultBandwidthMeter. Он даёт значение, которое считает дефолтной пропускной способностью вашей полосы.

```
@Override
public synchronized long getBitrateEstimate() {
  return bitrateEstimate;
}
```

initialBitrateEstimate (Country, [Wifi, 2G, 3G, 4G])

Country	WiFi	2G	3G	4G
	Mbit/sec	Kbit/sec	Mbit/sec	Mbit/sec
US	5.6	126	0.7	1.6
RU	5.6	126	1.3	4.2
DE	5.6	114	0.95	2.7

Как он устроен: там в коде огромная таблица констант, а параметры простые — страна (регион) и тип соединения (wi-fi, 2G, 3G, 4G). Какими бывают значения? Например, если у вас Wi-Fi и вы находитесь в США, ваш initial bandwidth — 5,6 мбит. А если 3G — 700 кбит.

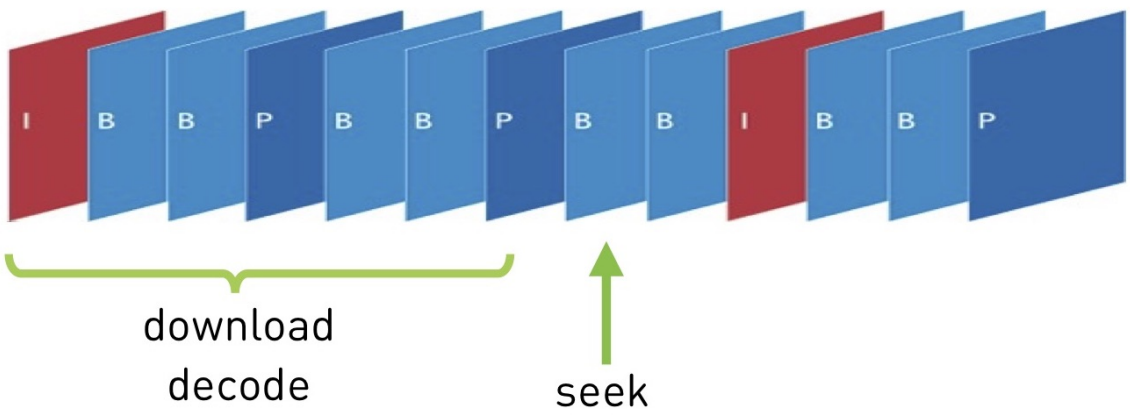
Видно, что по оценкам Google в России 4G в 2-3 раза быстрее, чем в Америке.

Понятное дело, что Россия — страна большая, и нас такая настройка совсем не устроила. Поэтому, если хотите сделать просто, запоминайте предыдущее значение для текущей сети, вычитайте на всякий случай единичку, и запускайте.

А если у вас большое приложение, которое проигрывает видео по всему миру, соберите статистику по подсетям, и рекомендуйте с сервера то качество, с которого нужно стартовать. Учитывайте, что после буферизации желательно увеличивать значение буфера (на Android это легко позволяется).

Как ускорить перемотку

Когда ваши пользователи перематывают видео в конкретное место (seek), вы можете попасть не в опорный кадр, а между ними. Соответственно, всё, что было с предшествующего опорного до него, нужно скачать и декодировать.

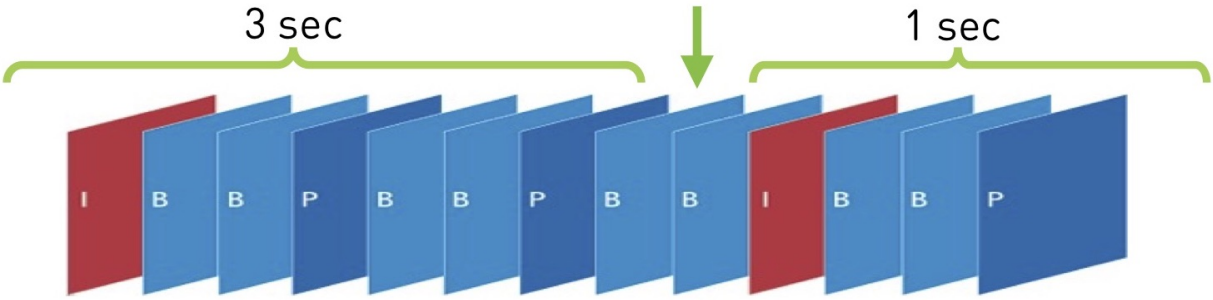


На самом деле, если пользователь смотрит двухчасовой фильм, то плюс-минус секунда для него не важна. Поэтому на iOS, если у вы знаете, что видео аккуратно порезано по определённым интервалам, можно рассчитать и отправлять в тот опорный кадр, где он есть (плюс небольшую дельту, чтобы точно оказаться после него, а не перед).

В ExoPlayer с версии 2.7.0 появилась возможность указать, как вы хотите перематывать, и есть вариант «в ближайший кадр». В этом случае он будет искать ближайший кадр на одну секунду вперёд и на три назад. Какой найдёт, в такой и перемотает.

```
player.setSeekParameters(SeekParameters.*_SYNC);

DEFAULT == EXACT
CLOSEST_SYNC
PREVIOUS_SYNC
NEXT_SYNC
new SeekParameters(3000000, 1000000);
```



Если видео запускают не с начала (а почти все видеохостинги запоминают время, до которого пользователь в прошлый раз досмотрел ролик), и вы перематываете в какую-то позицию, никогда не делайте на Android сначала prepare(mediaSource), а затем seekTo(). Если делать так, он сначала подготовится, чтобы играть с самого начала, а затем перемотает. Поменяйте эти строчки местами — это позволило нам сильно ускориться:

```
player = ExoPlayerFactory.newSimpleInstance(...);
...
player.prepare(mediaSource);
player.seekTo(resumePositionMs);
player.setPlayWhenReady(true);
```


```
player = ExoPlayerFactory.newSimpleInstance(...);
...
player.seekTo(resumePositionMs);
player.prepare(mediaSource, /* resetPosition= */ false, ...);
player.setPlayWhenReady(true);
```

Ещё, когда вы меняете видео (сначала воспроизводилось одно, затем другое), лучше не отпускать кодек. Это очень дорогая операция (порядка 100 мс), а следующее видео вы наверняка будете играть с теми же настройками декодера, и он вам полностью подойдет.

```
void swapVideo(Uri uri)
  player.stop();
  mediaSource = buildMediaSource(uri);
  player.prepare(mediaSource);
}
```

Рендеринг

На iOS всё рендерится просто, а вот в Android есть множество разных legacy-вещей.
Многие рендерят на TextureView. Вариант хорош тем, что это отдельная область памяти, вы целиком копируете кадр, она хорошо анимируется, синхронизирована с UI. Но есть минусы — большая задержка старта и высокое энергопотребление.

Есть SurfaceView. Там можно быстро стартовать, но он представляет собой дырку в видеопамяти. Поэтому на некоторых старых Android-устройствах при скроллинге появляется дырка в виде различных артефактов. YouTube изначально никогда не скроллил видео при воспроизведении, поэтому их это устраивало.

Поэтому есть ещё GLSurfaceView — промежуточный вариант между первыми двумя. Если вы запилите свой рендеринг, то сможете починить проблему медленной текстуры на старых устройствах.

TextureView	SurfaceView	GLSurfaceView
+ sync app UI pre M-MR1	+ frame release timing	mid
- less power efficient	+ power efficient	mid
~ 300-400 мс	~ 60 мс	~ 100 мс

Что в итоге: мы обнаружили, что если аккуратно потюнить ExoPlayer, можем сделать первый кадр быстрее на 23%. Количество «крутилок» уменьшили на 10%. И весь этот тюнинг нам добавил около 4% просмотров. Нужны ли вам эти 4% — решайте сами, но это нетрудно.

Итог: рекомендации по Android

- Используйте MediaPlayer для музыки, для всего остального существует ExoPlayer
- Оптимизируйте start, seek, swap
- Напишите свой эстиматор, его легко заменить
- Используйте правильную view согласно рекомендациям

Итог: рекомендации по iOS

С iOS всё сложнее:

- У нас есть лишний RTT на HLS в AVPlayer
- Проприетарный эстиматор
- Тормозит основной поток после AVPlayer#pause

- Нативный — нет исходников, обновления только с релизом iOS

Поэтому мы решили запилить свой собственный DASH-плеер, взяв за основу «архитектуру любого live-плеера». Мы использовали:

- cURL или GCDAsyncSocket
- AVAssetReader, потом отказались от него
- CADisplayLink
- AVSampleBufferDisplayLayer

Это трудоёмко, но мы получили ряд ускорений. Время до первого кадра сократилось на 28%, «крутилки» снизились на 6%. Но приятнее всего, что при переходе с HLS на DASH мы увеличили средний потребляемый битрейт на 100 кбит/сек, а число просмотров на 6%.

И рекомендации по iOS получаются такими:

- Оптимизируйте start и seek
- Используйте HLS over Fragmented mp4
- Напишите свой DASH-плеер

Думаем, что попробуем сделать наш плеер ещё и кроссплатформенным.

Выводы по первой части:

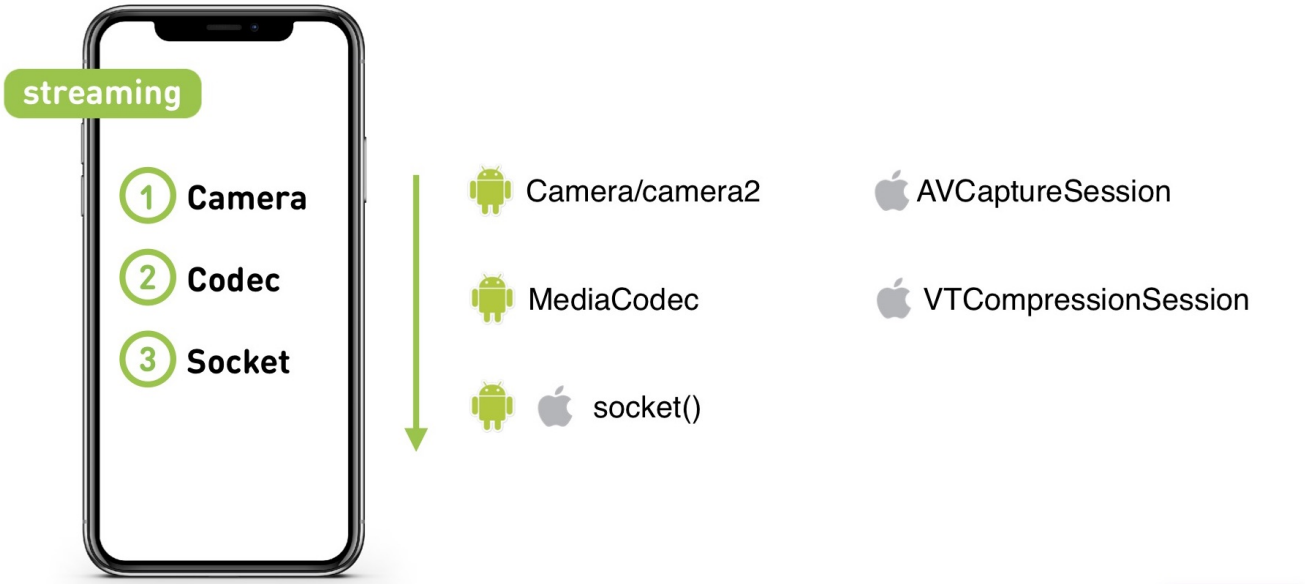
Вы узнали о видео, стандартной архитектуре плеера, сравнении и тюнинге плееров.

- Выберите подходящий формат стриминга (только не mp4)
- Выберите подходящий плеер (ExoPlayer, AVPlayer)
- Соберите статистику по firstFrame, seek, emptyBuffer
- Тюньте плеер и пилите свой эстиматор
- Напишите свой плеер (если конкретно вам действительно надо это делать)
- Если хотите сделать что-то серьезное, поднимите планку. Мы подняли планку до 4K, и нашли все баги: в performance, в парсинге, во всём.

А теперь — о стриминге.

Часть вторая: Стриминг видео своими руками

Что делать, если нужно отправлять видео с нашего мобильного устройства?



Нужны API для захвата с камеры и энкодинга. Эти API предоставляют доступ к камере и аппаратному энкодеру на iOS и Android, что очень важно — он работает намного быстрее, чем софтверный.

Сокет: можно использовать какой-нибудь wrapper в вашем любимом фреймворке, а можно использовать POSIX-сокет, делать всё в нативе, и тогда вы сможете сделать кроссплатформенную сетевую часть.

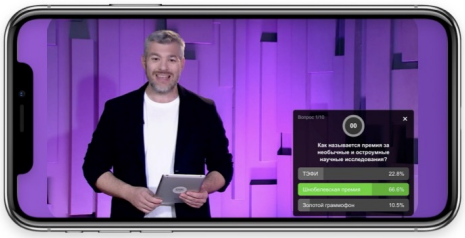
Чего мы хотим добиться от стриминга?

- Низкая задержка
- Хорошее качество трансляции
- Стабильность видео и звука
- Быстрый старт

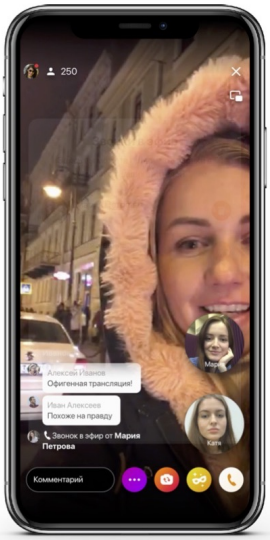
А с чем придётся бороться?

- Низкий bandwidth
- Задержки
- Прерывание звука и видео
- Задержка старта (N x RTT, обычно удобно рассчитывать старт как количество RTT)

Зачем нужна низкая задержка



Интерактивы со зрителями прямых эфиров



Звонки в прямой эфир

Спортивные трансляции в 4K

Первый кейс — интерактив со зрителями. У нас есть такие фичи, как викторина и звонки в прямой эфир, там очень важна задержка.

В викторине задают вопрос и ставят таймер: если у пользователей разное количество данных в буфере, они будут не в равных условиях, и время для ответа будет разным. А единственный способ добиться одинакового буфера — делать low latency.

Сценарий, который сейчас в разработке — звонки в прямой эфир. К стримеру можно позвонить в прямой эфир и с ним поговорить.

Третий кейс — спортивные трансляции в 4K. Например, вы смотрите с пивом и чипсами чемпионат мира по футболу, а соседи за стенкой смотрят то же самое. Если у них гол уже произошёл, а у вас ещё 30 секунд буферинга, они начинают радоваться и галдеть гораздо раньше и спойлерят весь кайф. Люди будут уходить к конкурентам, у которых задержка меньше.

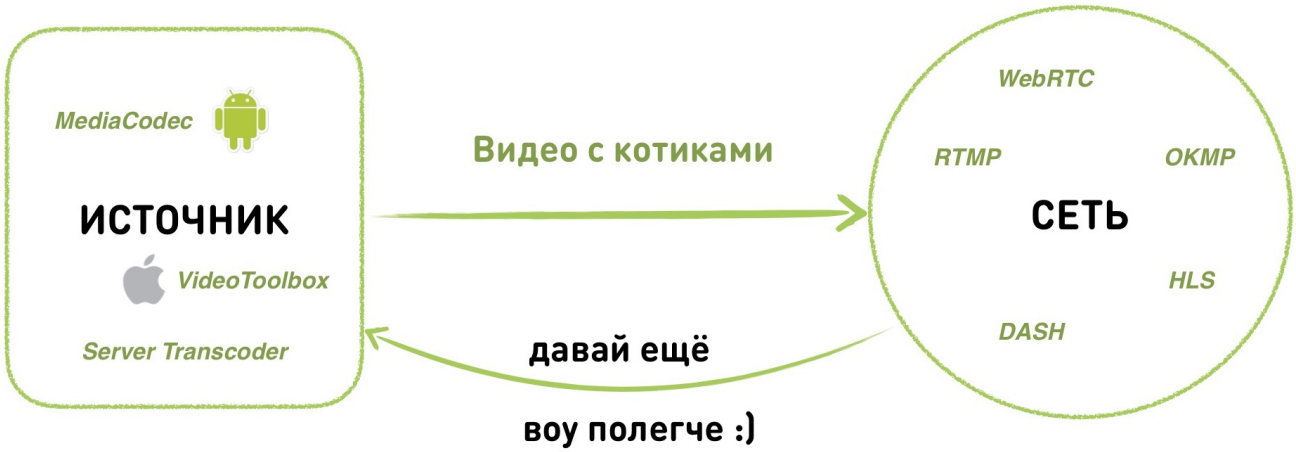
Адаптация

Сети у нас, конечно, разные, поэтому под каждую нужно подстраиваться. Для этого мы меняем битрейт видео и аудио (причём у видео он может меняться в 100 раз).

Какие-то кадры мы можем дропать, если видим, что не успеваем адаптироваться или исчерпали возможности по адаптации.

А ещё нам нужно менять разрешение видео. Если мы весь диапазон битрейтов в 100 раз будем кодировать в одном разрешении, получится плохо. Скажем, если с одним и тем же битрейтом 300 kbps закодировать FullHD-видео и 480p, то вроде бы превосходящий FullHD будет выглядеть хуже. При большом разрешении кодеку тяжело, картинка рассыпается: вместо того, чтобы тратить биты на кодирование самой картинки, он кодирует overhead-штуки. Так что разрешение должно соответствовать битрейту.

Общая схема кодирования с адаптацией следующая:



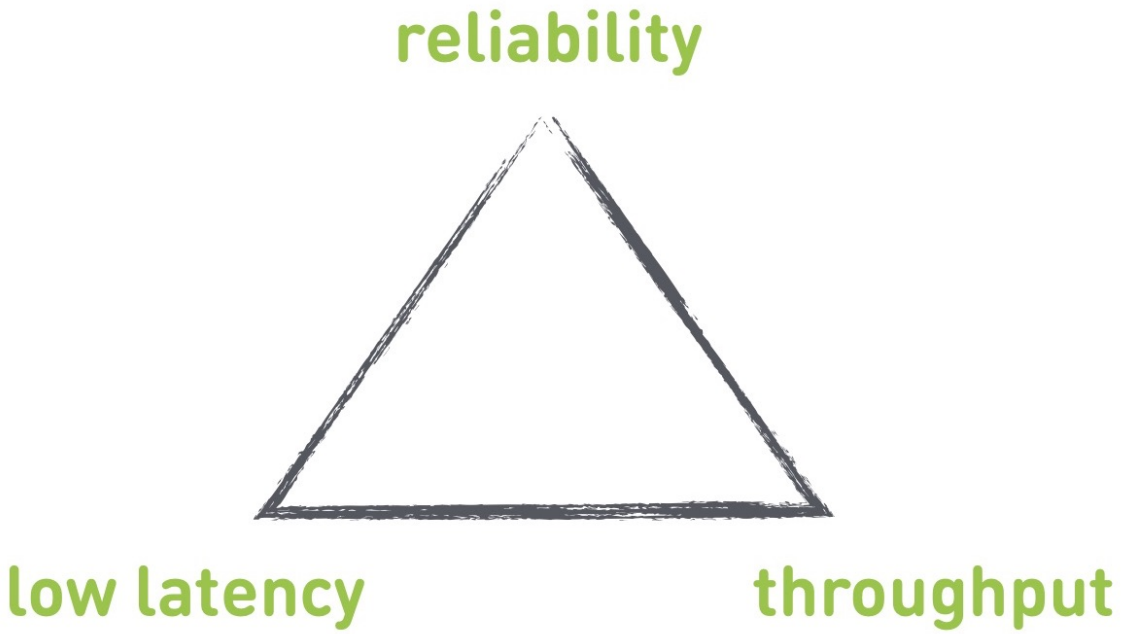
У нас есть некоторый источник, который может на лету менять битрейт, и есть сеть. Все данные уходят в сеть, от сети мы в каком-то виде получаем фидбек и понимаем, можно ли ещё увеличить объем посылаемых данных, или нужно уменьшать.

У стримера в качестве источника выступает MediaCodec или VideoToolbox (в зависимости от платформы). А при воспроизведении всё делает Server Transcoder.

В сети — различные сетевые протоколы, про которые уже поговорили и ещё поговорим.

Треугольник компромисса

Когда мы начинаем вникать в стриминг, натыкаемся на определенное количество компромиссов. В частности, есть треугольник, в углах которого reliability — надежность (отсутствие дропов), throughput — пропускная способность (насколько мы используем сеть) и low latency — низкая задержка (получаем ли мы низкую задержку).



Если мы начинаем оптимизировать один из этих параметров, остальные неизбежно проваливаются. Мы не можем получить всё и сразу, приходится чем-то жертвовать.

Протоколы

Протоколы, которые мы посмотрим сегодня: RTMP и WebRTC — стандартные протоколы, OKMP — наш кастомный протокол.

Стоит упомянуть, что RTMP работает поверх TCP, а другие два — UDP.

RTMP

Что он дает? В определенном роде, это стандарт, который поддерживается всеми сервисами — YouTube, Twitch, Flash, OK. Они используют его, чтобы пользователи могли заливать live-стримы. Если вы хотите лить live-стрим на какой-то сторонний сервис, скорее всего, придется работать с RTMP.

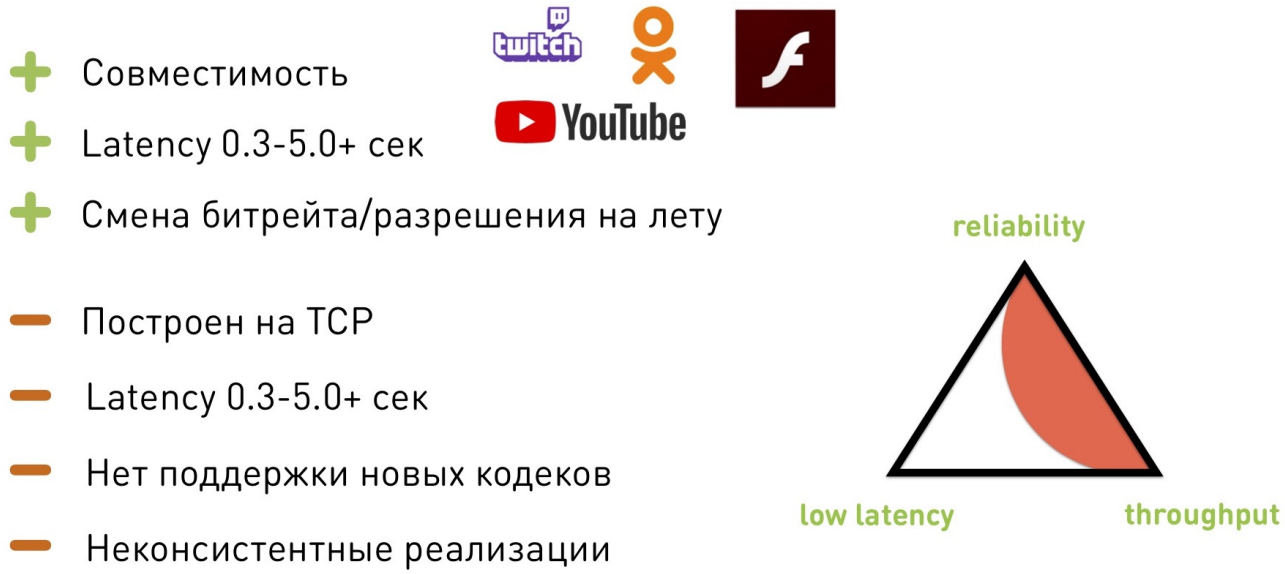
Минимальная задержка, которую нам удавалось достигать от стримера до плеера — 300 мс, но это в идеальной сети в хорошую погоду. Когда у нас реальная сеть, задержка обычно вырастает до 2-3 секунд, а если совсем всё плохо с сетью, может вырасти до десятков секунд.

RTMP поддерживает смену разрешения и битрейта на лету (остальные упомянутые протоколы тоже, но про RTMP встречается ошибочная информация, что смены на лету нет).

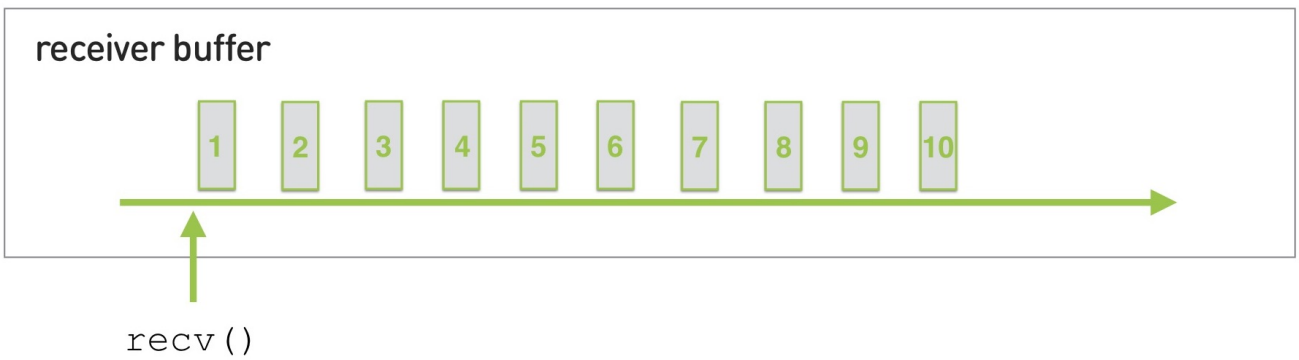
Из минусов: построен на TCP (позже объясним, почему это плохо), задержка неконтролируемая.

Если посмотреть на треугольник, RTMP не сможет дать low latency. Можно получить, но совсем не гарантированно.

К тому же RTMP немного подзаброшен: в нём нет поддержки новых кодеков, так как Adobe этим не занимается, а документация достаточно древняя и кривая.



Почему TCP не подходит для live-трансляций? TCP даёт гарантию доставки: данные, которые вы положили в сокет, будут доставлены ровно в том порядке и том виде, в котором туда положили. Ничего не будет дропнуто или переставлено. TCP либо это сделает, либо умрёт. Но это означает, что исключена гарантия задержки — он не сможет дропнуть старые данные, которые уже, может, и не нужно досылать. Начинает расти буфер, бэклоги и так далее.



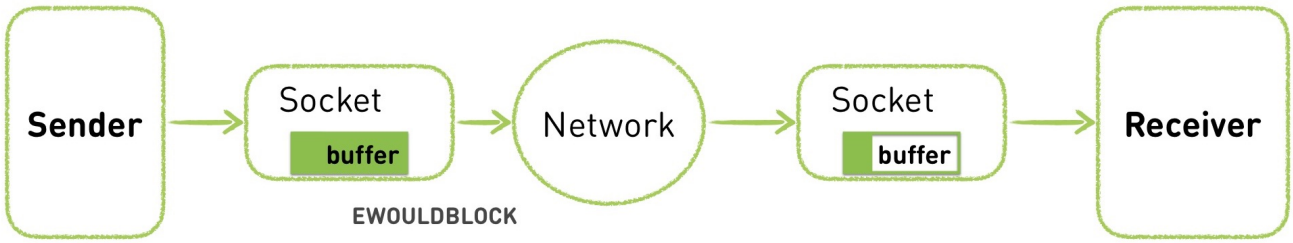
Как иллюстрация — проблема Head of Line blocking. Встречается не только в стриминге, но и во многих других кейсах.

Что это? У нас есть изначально пустой receiver buffer. Нам откуда-то прилетают данные: много данных и много IP-пакетов. Мы получили первый IP-пакет, и на приемнике методом recv() можем этот пакет вычитать, получить данные, проиграть, отрендерить. Но потом вдруг потерялся второй пакет. Что дальше происходит?

Чтобы восстановить потерянный IP-пакет, TCP должен сделать retransmission. Чтобы это произошло, нужно потратить RTT, при этом retransmission тоже может потеряться, и мы зациклимся. Если пакетов много, это обязательно произойдет.

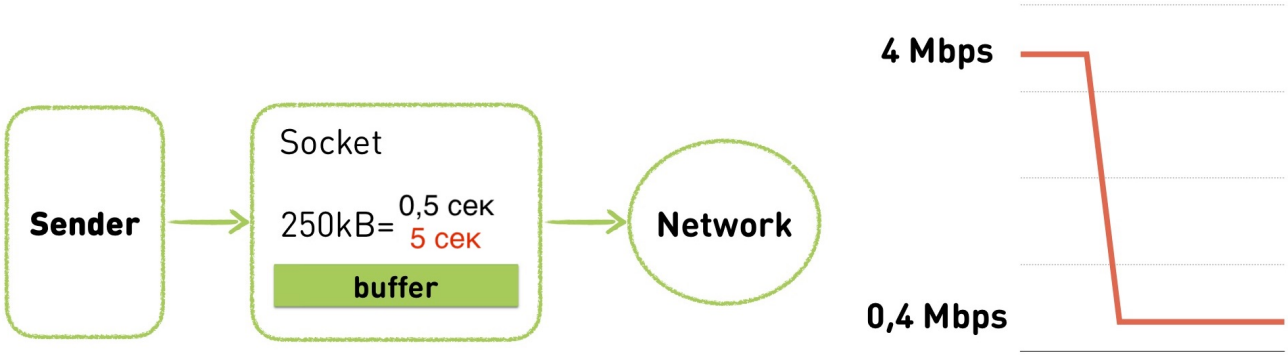
После этого приходит много данных, которые мы не можем вычитать, так как стоим и ждём второй пакет. Хотя он показывал кадр трансляции, который произошёл пять минут назад и уже не нужен.

Чтобы понять ещё одну проблему, посмотрим на адаптацию RTMP. Адаптацию мы делаем на стороне отправителя. Если сеть не может пропихнуть данные с той скоростью, с которой их подкладываем в сокет, заполняется буфер, и сокет говорит EWOULDBLOCK или блокируется, если в этот момент используется блокирующий.



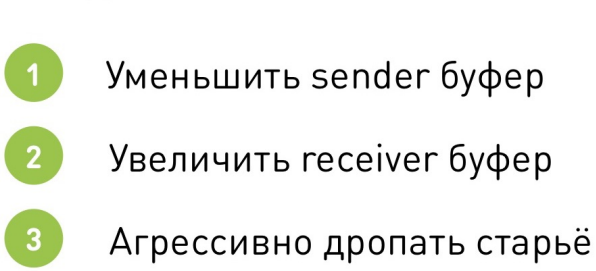
Только в этот момент мы понимаем, что у нас проблемы, и нужно снизить качество.

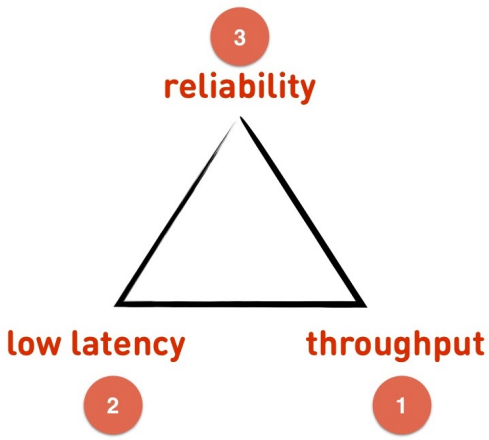
Допустим, у нас есть сеть с определенной скоростью 4 мбит/сек. Мы выбрали размер сокета 250 КБ (соответствует 0,5 секунд на нашей скорости). Вдруг сеть провалилась в 10 раз — это нормальная ситуация. У нас стало 400 кбит/сек. Буфер быстро заполнился за полсекунды, и только в этот момент мы понимаем, что нужно переключиться вниз.



Но теперь проблема — у нас 250 КБ-буфер будет передаваться 5 секунд. Мы уже капитально отстаём: нужно пропихнуть сначала старые данные, а только потом пойдут новые и адаптированные, чтобы догнать realtime.

Что делать? Тут как раз актуален наш «треугольник компромиссов».





- Можем уменьшить sender-буфер, поставить вместо 0,5 сек — 0,1 сек. Но мы теряем пропускную способность, так как часто будем «паниковать» и переключаться вниз. Более того, TCP работает таким образом, что, если поставить sender-буфер меньше, чем RTT, нельзя использовать полную пропускную способность канала, она будет кратно уменьшаться.
- Можем увеличить receiver-буфер. С большим буфером приходят данные, какие-то неравномерности в пределах буфера можем сгладить. Но, естественно, теряем low latency, так как сразу поставили 5-секундный буфер.
- Можем агрессивно дропать старые данные. В TCP единственный вариант для этого — оборвать соединение, и пересоздать его. Мы теряем reliability, так как в это время плееру нечего показать.

WebRTC

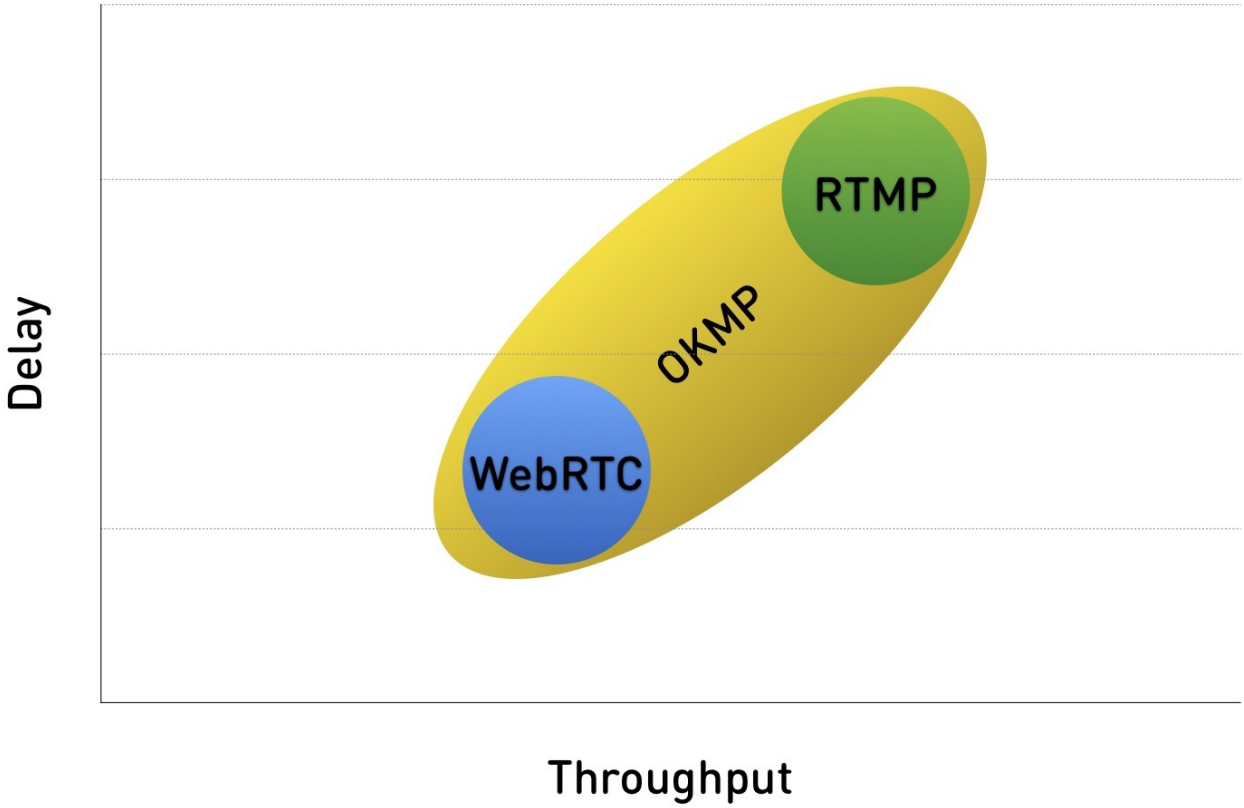
Это библиотека на C++, которая уже учитывает опыт и работает поверх UDP. Билдится под iOS, Android, встроена в браузеры, поддерживает HTML5. Так как она заточена под P2P-звонки, задержка 0,1-1 сек.

Из минусов: это монолитная библиотека с обилием legacy, которые невозможно убрать. К тому же из-за ориентированности на P2P-звонки она приоритезирует low latency. Казалось бы, мы этого и хотели, но ради этого она жертвует остальными параметрами. И там нет настроек, позволяющих поменять приоритеты.

Также надо учитывать, что библиотека клиентская, заточена под разговор двух клиентов без сервера. Сервер нужно искать сторонний, либо писать собственный.

Что выбрать — RTMP или WebRTC? Мы реализовали оба протокола и протестировали их в разных сценариях. На графике у WebRTC низкий delay, но низкий throughput, а у RTMP наоборот. А между ними — дыра.

И нам захотелось сделать протокол, который полностью покроет эту дыру и сможет работать как в режиме WebRTC, так и в RTMP. Сделали и назвали его OKMP.

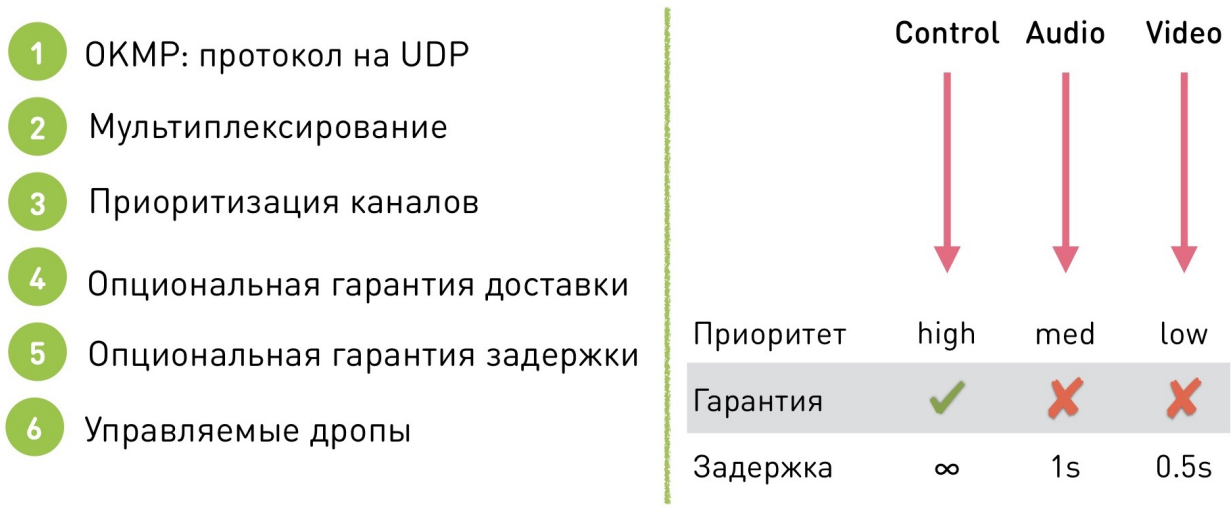


OKMP

Это гибкий протокол на UDP.

Поддерживает мультиплексирование. Что это означает: внутри сессии есть несколько каналов (в случае OK Live — управляющий, аудио и видео). Внутри каждого канала данные гарантированно доставляются в определённом порядке (но сами доставляются не гарантированно), а между каналами порядок не гарантирован, так как не важен.

Что это даёт? Во-первых, это нам дало возможность приоритизировать каналы. Можем сказать, что у управляющего канала высокий приоритет, у звука средний, а у видео — низкий. Дрожание видео и неравномерность доставки видео проще замаскировать, и у пользователя меньше проблем от проблем видео, чем от неприятного заикания аудио.



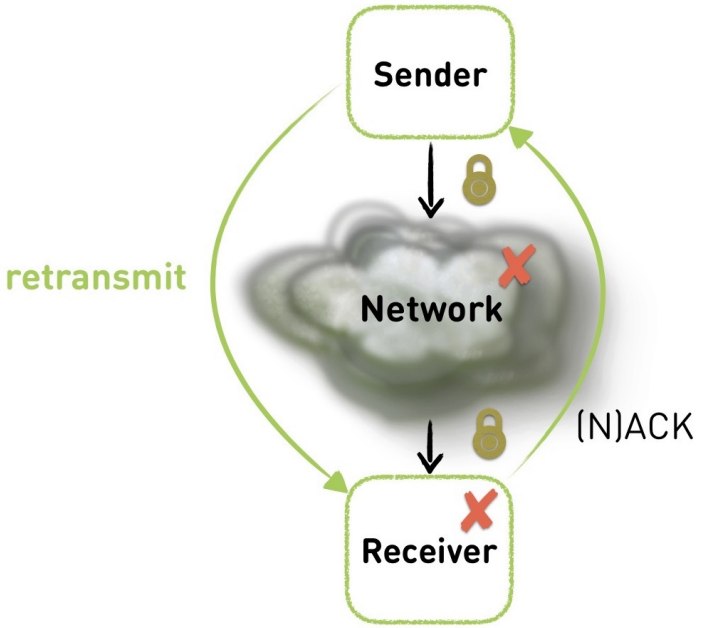
Кроме этого, в нашем протоколе есть опциональная гарантия доставки. Мы можем сказать, что на определенном канале работаем в режиме TCP, с гарантированной доставкой, а на остальных позволяем какие-то дропы.

Благодаря этому можно сделать и гарантию задержки: на TCP-канале гарантии задержки нет, а на остальных, где разрешены дропы, выставили порог, после которого данные начинают дропаться и старые данные перестаём доставлять.

Например, для аудио это 1 секунда, а для видео 0,5 секунд. Почему порог разный? Это ещё один механизм приоритезации.

Раз нам важнее, чтобы аудио было плавным, то дропать начинаем в первую очередь видео.

Наш протокол гибко настраивается: нет единого режима работы, мы на лету меняем настройки, чтобы переключаться в нужный режим без видимых эффектов для пользователя. Зачем? Например, для тех же видеозвонков: если в стриме начинается видеозвонок, тихо переводим в режим low latency. А потом обратно в режим throughput для достижения максимального качества.
Сложности реализации



Конечно, если решите писать свой протокол на UDP, наткнётесь на некоторые проблемы. Используя TCP, мы получаем механизмы, которые на UDP придется писать самостоятельно:

- Packetizing/Depacketizing. Нужно самостоятельно нарезать данные на пакеты, размером примерно 1,5 КБ, чтобы они влезли в MTU сети.
- Reordering. Вы отправляете пакеты в одном порядке, а они переставляются в пути и приходят в другом. Чтобы это побороть, нужно выставлять sequence с номером пакета, и на ресивере их переставлять обратно.
- Losses. Разумеется, есть потери. Когда происходит потеря, receiver должен отдельно сказать sender'у, что «вот эти пакеты я получил, а эти не получил», а sender должен ретрансмитить недостающие пакеты. Или дропнуть их.
- Flow Control. Если Receiver не получает данные, не успевает за тем темпом, с которым мы в него пихаем, данные могут начать теряться, мы должны эту ситуацию обработать. В случае TCP сокет send заблокируется, а в случае UDP не заблокируется, вы должны сами понимать, что receiver не получает данные, и понижать количество отправляемых данных.
- Congestion Control. Похожая штука, только в этом случае умер network. Если мы отправляем пакеты в умерший network, угробим не только наше соединение, но и соседние.
- Encryption. Нужно позаботиться о шифровании
- ... и ещё много о чем

OKMP vs RTMP

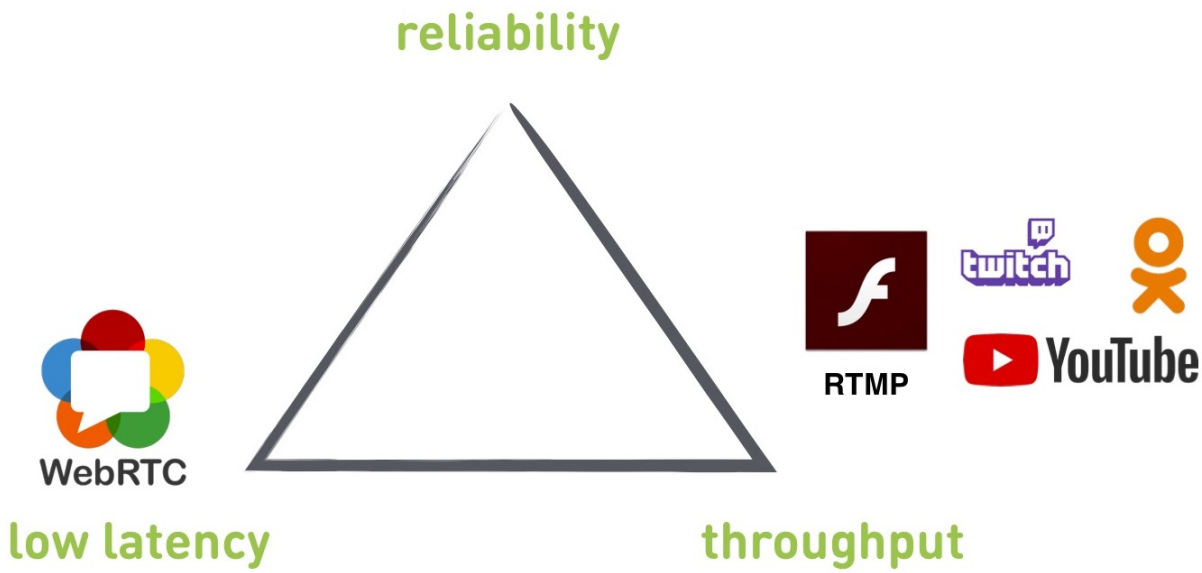
Что мы получили, начав использовать OKMP вместо RTMP?

- Средний прирост битрейта по OKLive — 30%.
- Jitter (мера неравномерности прихода пакетов) — 0% (в среднем та же).
- Jitter Audio — -25%

- Jitter Video — 40%

Изменения в аудио и видео — демонстрация приоритетов в нашем протоколе. Аудио мы даём больший приоритет, и оно стало приходить ровнее за счёт видео.

Как выбрать протокол для стриминга



Если нужна низкая задержка — WebRTC.

Если хотите работать с внешними сервисами, публиковать видео на third party-сервисах, придется использовать RTMP.

Если хотите протокол, заточенный под ваши сценарии — реализуйте свой.

Теги: iOS, Android, видео, HLS, MPEG-DASH, WebRTC, RTMP, OKMP, OK Live

Хабы: Блог компании Одноклассники, Разработка под iOS, Разработка мобильных приложений, Разработка под Android

10 января 2019 в 11:03

Реактивный раздатчик ok.ru/music

+16 4,5k 28 4

20 августа 2015 в 09:59

Загрузка видео «без единого разрыва»

+26 31,3k 138 12

Комментарии 2

goodvin1709 17 сентября 2019 в 12:51 +3

Спасибо за очень хороший материал, все очень хорошо описано.

lim14 17 сентября 2019 в 14:04 +2

Очень достойная статья. Посоветуйте какую-нибудь информацию, чтобы ещё больше углубиться в streaming видео

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

Дуров рассказал, как будет монетизироваться Telegram

+29 38,9k 10 80

Рок-звёзды без премий: как Netflix хакнул систему мотивации сотрудников

+96 33,9k 119 98

Почему Джеф Безос – самый опасный политикан на планете

+63 64,9k 155 549

В Россию ввезли крупнейшую партию оборудования для майнинга за $40–60 млн

+15 25,2k 10 77

Приложение для женщин, которое делают мужчины. Как это работает

Мегапост