Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

# Bachelor's Thesis Nr. 213b

Systems Group, Department of Computer Science, ETH Zurich

Quantifying and Explaining Unfairness in Online Video Streaming

by
Dimitri Wessels

Supervised by
Prof. Dr. Ankit Singla and Melissa Licciardello

October 2017 – April 2018

inf | Informatik
Computer Science

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

## Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

---

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Quantifying and Explaining Unfairness in Online Video Streaming

**Authored by** (in block letters):

*For papers written by groups the names of all authors are required.*

| **Name(s):** | **First name(s):** |
|---|---|
| Wessels | Dimitri |
| | |
| | |
| | |
| | |

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

| **Place, date** | **Signature(s)** |
|---|---|
| Zürich, 23. 04. 2018 | D. Wessels |
| | |
| | |
| | |
| | |

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*

**Abstract**

In recent years many leading video streaming providers (e.g. YouTube, Netflix and Vimeo [1, 2, 3]) have employed adaptive bitrate (ABR) to improve user-perceived quality of experience (QoE). There is an abundance of research on designing better ABR algorithms (e.g. [4, 5]) and on understanding how fairness of competing players can be achieved (e.g. [6, 7, 8]).

In this thesis we want to focus on measuring and quantifying existing unfairness of ABR algorithms in use at YouTube and Vimeo, a topic that is not well-understood yet. To make such measurements possible for end users across a variety of services, we developed a system which can examine various parts of video streaming providers and can be extended to accommodate for streaming providers not supported by the current platform.

We present evidence that if a YouTube client directly competes with a Vimeo client, the YouTube player gets a larger share of the available bandwidth.

# Contents

# Chapter 1

# Introduction

## 1.1 Overview

Today over 70% of the worldwide consumer internet traffic is generated by video streaming [9]. A large part of it is attributed to the growth of mobile traffic in the last years. User perceived quality of experience (QoE) is crucial for success in the highly competitive streaming market [10]. Unsurprisingly, there is a great deal of research on improving QoE as the technologies used are evolving rapidly.

#### Adaptive Streaming

In an effort to deliver the best possible quality to clients, even if the network conditions are unstable, adaptive bitrate streaming was introduced. This technology allows to change the quality during video streaming, but comes with many difficulties. This thesis takes a closer look at what happens during adaptive streaming and shows fairness is not guaranteed for competing adaptive streaming clients.

## 1.2 Contributions

The focus of this thesis is quantifying unfairness on widely used online video streaming platforms. We implemented an evaluation harness [11] in Python to measure and quantify how streaming clients interact with each other while running in parallel. At the moment our system supports experiments on YouTube and Vimeo but it is extensible. The system is presented in chapter 3.

Our results show that YouTube's adaptive bitrate algorithm is overly aggressive during start up and that TCP alone isn't enough to ensure fairness. After presenting the results in chapter 4 we discuss possible solutions in chapter 5.

## 1.3 Structure of this document

The thesis is organized as follows: We discuss the technology behind video streaming and the tools used in chapter 2. In chapter 3 we introduce the design and implementation of our evaluation harness. We show how the system can be used and what the thought process behind the components is. At the end of the chapter we specify how to extend our work for experiments on streaming platforms not currently implemented. We present our obtained results in chapter 4 and conclude in chapter 5.

# Chapter 2

# Background

In this chapter we give a more in-depth overview of adaptive streaming and take a look at the utilities behind our testing environment.

## 2.1 Dynamic Adaptive Streaming over HTTP

**Introduction** In 2012 Dynamic Adaptive Streaming over HTTP (DASH) was proposed. DASH splits every video into multiple non-overlapping segments, each usually worth some seconds of playback time. Each such segment is available in different formats and qualities (e.g. 240p or 360p). The client requests individual segments instead of the whole file. During playback segments of different qualities can be combined. Typically a video streaming client using DASH downloads a manifest file at the start of the video playback containing the different segments and bitrates available. The part of the client responsible for choosing the next segment to be requested is called an adaptive bitrate (ABR) algorithm.

Normally the ABR algorithm tries to balance requests in a way that allows to get the highest bitrate possible without running out of buffer. More specifically a client tries to maximize a quality of experience function. It is important to note that the ABR algorithm is not part of DASH and is not standardized. There is an active effort to develop better ABR algorithms [4, 5]. The most successful ABR algorithms like YouTube's implementation are secret and of great interest to the research community.

### 2.1.1 ABR algorithm

An ABR algorithm decides which segment should be requested next. This algorithm is not part of the DASH standard and each party is free to implement an own version of it. There is a reference implementation of DASH available [12] but the major players each have implemented their own flavor. Generally an ABR algorithm requests the highest bitrate available that is still supported

by the available bandwidth. However, choosing a bitrate is hard for various reasons:

1. *Network throughput instability*: The available bandwidth is hard to estimate and changes frequently, especially on mobile devices.

2. *Competing goals*: Avoiding rebuffering events and attaining the highest bitrate possible are directly competing with each other

3. *Coarse grained decisions*: There are only few different bitrates available for each segment and the differences between them are large.

### 2.1.2 Multiple competing players

Choosing a bitrate gets much more complicated if multiple players compete with each other. Consider a situation with two competing players $p_1$ and $p_2$, both using the same bottleneck, for example an home router. Assume both players are initially downloading segments of the same bitrate $b_0$. $p_1$ wants to start downloading segments in a higher quality with bitrate $b_1 > b_0$. If the available bandwidth doesn't allow $p_1$ to download the higher quality segments with bitrate $b_1$, we expect to see rebuffering events for $p_1$ happen. The other player $p_2$ notices that the available bandwidth decreased after $p_1$ started downloading segments at a higher bitrate. In response $p_2$ starts downloading segments in a lower quality $b_2 < b_0$. $p_1$, who has now more bandwidth available than at the beginning stabilizes on $b_1$ and $p_2$ stabilizes on $b_2$ with $b_2 < b_0 < b_1$.

Ideally all competing players get an equal share of the total bandwidth available. In practice tcp is not ideal as players requesting larger bitrate get a larger share of the bandwidth [13].

### 2.1.3 ABR in practice

In existing literature ABR algorithms are often described as operating in two distinct modes of operation: **Buffering** and **Steady** state ([14, 15]). Generally, a player tries to fill its buffer as quickly as possible during the buffering phase. When there is a risk of buffer depletion (e.g. amount of seconds left in the playback buffer is lower than some constant $x$) the player is in the buffering phase. If the player has reached a large enough buffer size it switches to steady state mode. The risk of buffer under-run is low in this mode of operation. During steady state the player just tries to sustain the buffer size instead of actively trying to grow it, as a result, a player requests segments less aggressively in steady state than in buffering mode.

#### Buffering and Steady state: An example

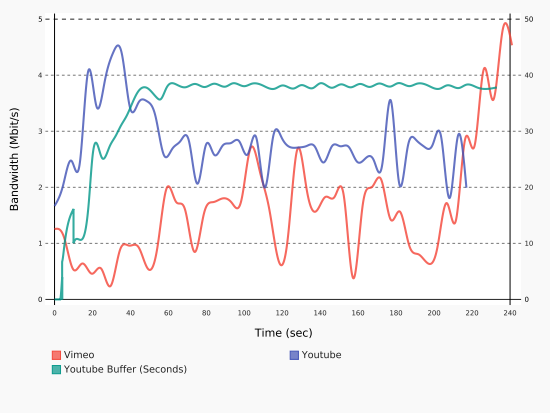


Figure 2.1: Bandwidth and YouTube's playback buffer over time. Constant 5000 kbit/s bottleneck.

**2.1** An example of buffering and steady state is presented here. The whole experiment is discussed in section 4.2. Refer to that section for more information about how the experiment was conducted or what its significance is. Here we see YouTube claiming a large share of the available bandwidth (the blue line). YouTube's playback buffer reaches roughly 40 seconds worth of video segments after about 60 seconds. Then, YouTube's ABR algorithm scales back and allows Vimeo to use more of the available bandwidth. Here YouTube starts in buffering mode and switches to steady state after about 60 seconds.

### 2.1.4 Quality of Experience

An ABR algorithms final goal is to optimize user perceived Quality of Experience (QoE) to increase user engagement [10]. The key features of QoE functions in adaptive video streaming are:

- *Average video quality:* The average video quality over all segments. Higher is better.
- *Average quality variations:* The magnitude of changes between video segments. Lower is better.
- *Time spent rebuffering:* The time spent rebuffering during the video playback. Lower is better.
- *Start up time:* The time until the first frame of the video is played. Lower is better.

How each features is weighted is up to the specific implementation.

### 2.1.5 Implementation in YouTube and Vimeo

It is important to keep in mind that DASH is not a protocol - it is a common standard only specifying an architecture. Many details such as the choice of the video container (e.g. WebM or mp4) are left to the party implementing DASH. As a result Vimeo and YouTube both have some important differences.

#### Vimeo

Vimeo started using DASH towards the end of 2015. All videos that are uploaded to Vimeo today use DASH but there are still some older videos, from before 2015, for which adaptive streaming is not available.

**A typical DASH session** First a GET request is made to `https://player.vimeo.com/video/<video_id>/config` with several arguments passed. The server responds with a config file holding a list of different Content Delivery Network's (CDN) and formats available. The client chooses a CDN and makes a request to get a file called *master.json*. Contained in the *master.json* file are details about the different segments. In particular the file contains a mapping between segments and URLs to request that segment. To understand a video session on Vimeo our tools rely heavily on that file.

#### YouTube

YouTube started using DASH around 2013 [16] and uses it as default playback mechanism since 2015.

**Segment length** YouTube doesn't use fixed length segments - it reduces segment length dynamically if the available bandwidth decreases before switching to a lower bitrate. A behavior not observed in any other implementations making direct comparison hard.

**Communication** Much of the communication between YouTube and the client is facilitated by specific parameters in the request headers [17]. To understand the behavior of YouTube's ABR algorithm three parameters are important:

- *itag:* This is an integer representing a format. As an example itag 139 maps to DASH MP4 Audio in m4a format.
- *range:* This is the byte range of a requested segment.
- *rbuf:* rbuf describes the current buffer size.

**Regional differences** There are also some regional considerations in YouTube's ABR algorithm. For example, YouTube is less likely to play a video at the lowest bitrate, even if that means rebuffering for clients from the United States. For other regions YouTube starts rebuffering sooner [16].

## 2.2 Technologies used

### 2.2.1 BrowserMob Proxy

BrowserMob Proxy (BMP) [18] is an open-source utility. BMP deploys a proxy server which is controlled using a simple REST protocol. While BMP provides different ways to control HTTP traffic, in this thesis, we use it only to capture HTTP Archive (HAR) files [19].

### 2.2.2 HTTP Archive (HAR)

In this thesis our main goal is to measure and quantify interactions between adaptive streaming clients. We need a way to store information about browser sessions. Specifically we are interested in what websites were accessed, at what time were they accessed, how much data was sent and received and what the headers and cookies looked like. The HTTP Archive (HAR) [19] format is a good fit for that problem. It stores all the data we need and is easily parsed, since it uses JSON. We collect HAR files with BrowserMob Proxy (see section 2.2.1) but there is wide variety of tools supporting the capture of HAR files, such as YouTube and Firefox with their developer tools.

### 2.2.3 Traffic Control (TC)

One of the challenges of this thesis was the decision of how to control the bandwidth during experiments. We tried various different methods: BrowserMob Proxy, Selenium with chromedriver and traffic control (TC). Controlling the bandwidth with BrowserMob Proxy proved unreliable. For this reason we use BrowserMob Proxy only to capture HAR files. Using Selenium directly with chromedriver, a webdriver to control Google Chrome, to control the bandwidth was precise and easy but restricted us to only use Google Chrome or Chromium and prevented us from using Firefox. It is also not possible to globally limit the bandwidth for several selenium instances running in parallel. TC proved to have none of those problems and is now used for controlling the bandwidth in our system.

TC is a powerful built in Linux command line tool that offers a huge amount of options (e.g. prioritizing flows that conform to custom rules and filtering unwanted traffic).

### 2.2.4 Selenium

**Non-obvious inputs** YouTube and Vimeo consider some parameters in their adaptive bitrate algorithms that are not immediately obvious. YouTube's ABR algorithm only selects video segments in resolutions that are below or equal to the physical resolution of the monitor they are played on. As an example if watching a video on YouTube on a Monitor with a native resolution of 1920x1080 pixel, YouTube will not select a segment in 4k resolution. Other factors such

as screen size, viewing habits and so on may also input parameters to their algorithms.

**Selenium** For our experiments we want to simulate regular video sessions as closely as possible. If we would just download videos from Vimeo and YouTube in parallel with a command line tool, parameters such as our native resolution or even our current bandwidth are not available to the streaming platforms. To automate realistic video sessions we use Selenium [20], a framework for automating web browsers. Selenium has support in popular programming languages such as Python and Java and is easy to work with.

**WebDriver** At the core of selenium is the *WebDriver* API. It makes direct calls to browsers using each browser's native support for automation [21].

## 2.3 Related Work

Players in the industry struggle to keep up with their competition. Studies have shown that user engagement is heavily related to rebuffering time and video quality [10]. With the challenges introduced by moving to higher and higher bitrates in recent years (e.g. 4k, virtual reality and 60 frames per second) and the difficulties with unstable mobile traffic, video streaming is a fast moving field.

**Adaptive bitrate streaming** There is a large body of work on improving adaptive bitrate streaming [4, 5] and on deployed streaming algorithms such as YouTube's implementation [17, 22].

**Competing players** Some work [8, 7] is focused on understanding how fairness across multiple clients running different adaptive bitrate algorithms could be achieved. They argue for a router-assisted bandwidth allocation algorithm.

**Approach in this work** To the best of our knowledge there is no existing work that systematically quantifies unfairness on existing platforms and releases a solution to measure it. This thesis tries to close that gap. We argue that it is not only important to understand how competing video players behave today, but want also to provide a tool that makes it possible to verify if today's behavior is the same as yesterdays behavior.

# Chapter 3

# Design and Implementation

## 3.1 Overview

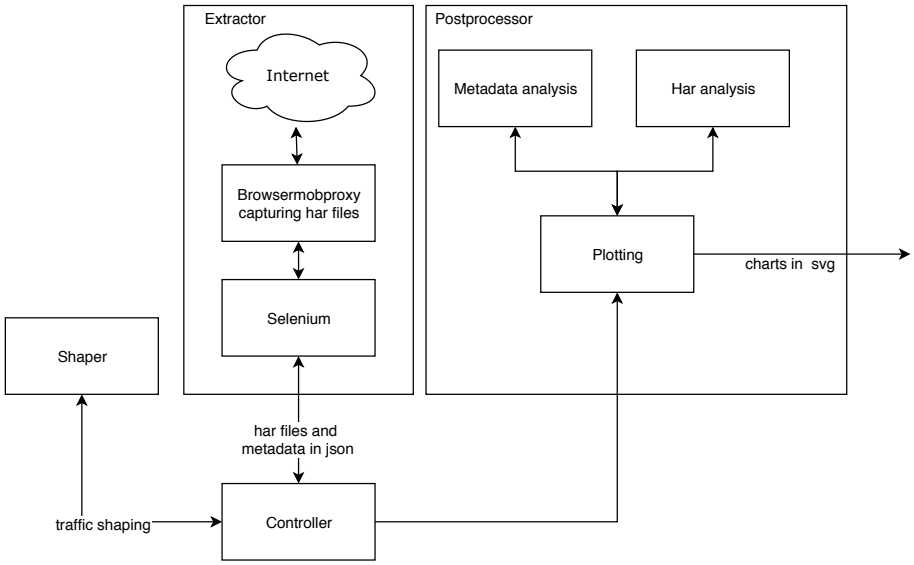


Figure 3.1: High-level overview of the testing environment

This section discusses the design and implementation of the testing environment [11] that allowed us to run the experiments described in chapter 4. First we give an overview of our system, then we present an example and explain how the components of the system work.

We designed a system that allows to easily run experiments on YouTube, Vimeo and on both platforms in parallel. It's designed in a way to easily add other video streaming platforms . The system consists of a controller relying on three key components (Figure 3.1):

1. *Shaper:* To globally limit the available bandwidth for our video players we use Shaper, a module handling trafic shaping with Linux' tc (see section 2.2.3).

2. *Extractor:* An extractor uses Selenium (see section 2.2.4) to establish a video session and Browsermob Proxy to capture the requests and responses during that video session. extractor

3. *Postprocessor:* Processes the retrieved HAR and metadata files to generate plots and statistics.

Before describing each component in greater detail we present the code to run a simple experiment which was used to generate figure 4.1, 4.2 and 4.4 in section 4.

```
# Importing Shaper for traffic shaping and YouTube and Vimeo specific extractors
from shaper import Shaper
from extractor import Vimeo, Youtube
from postprocessor.plotter import plot_combined
# VIMEO_AWAKENING and YOUTUBE_AWAKENING are two URLs pointing to the same video
from config import VIMEO_AWAKENING, YOUTUBE_AWAKENING

# We need to name an experiment,
# this name is used by the extractors and the plotting library
# to determine the path of the various files produced
NAME = 'combined_e1'
shaper = Shaper()
shaper.limit_download(5000)

# setup an extractor for vimeo and for youtube
extractors = [
    Vimeo(VIMEO_AWAKENING, shaper=shaper, name="vimeo_" + NAME),
    Youtube(YOUTUBE_AWAKENING, shaper=shaper, name="youtube_" + NAME)
]

# start both extractors and wait for them both to finish
for extractor in extractors:
    extractor.start()
for extractor in extractors:
    extractor.join()

shaper.reset_ingress()

# Generate three plots: total data downloaded, bandwidth, playback quality
plot_combined(NAME, primary='mb')
plot_combined(NAME, primary='bandwidth')
plot_combined(NAME, primary='quality')
```

Listing 1: Experiment running the same video on YouTube and Vimeo

**Listing 1** First we import the necessary libraries [line 2 to 6]. Next, we limit the download speed to 5000kilobit/s [line 12 and 13]. We run two extractors [line 16 to 25] until they have both finished. Finally we reset the network conditions back to normal and generate plots with the retrieved data [line 30 to 32].

## 3.2 Shaper

**Shaper.** Shaper is a python class wrapping the behavior of TC (see section 2.2.3) providing methods to limit download speed and to reset all rules created. It's also possible to read the current download limit from a Shaper object.

TC can't limit download speed directly - since we can't ask the Internet to send us only a limited amount of data. Instead we drop all the packets which go over the defined limit and let TCP take care of the rest.

## 3.3 Extractor

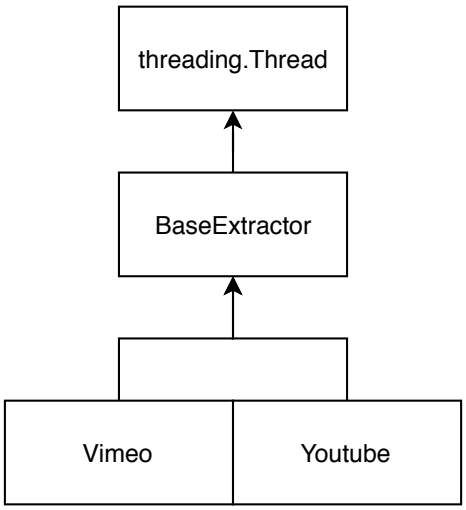


Figure 3.2: available extractors

**Extractor.** An extractor is used to simulate a video session on a video streaming provider. It handles setting up and connecting to a Browsermob Proxy (see section 2.2.1) instance, in order to intercept, capture and decrypt traffic, which is then made available to other parts of the system as a HAR file (see section 2.2.2). It also sets up a selenium (see section 2.2.4) session. Currently Google Chrome and Firefox are supported webdrivers.

Extractor exposes classes for specific streaming platforms. Currently *Vimeo* and *Youtube* are implemented for running experiments on YouTube and Vimeo. They both inherit common behaviour from *BaseExtractor* which is not intended to be used directly. *BaseExtractor* inherits from Thread, which makes *Vimeo* and *Youtube* runnable in parallel.

*BaseExtractor* handles shared functionality like setting up Browsermob Proxy and Selenium. *Youtube* and *Vimeo* handle platform specifics. They continually monitor the video beeing played with the respective api [23, 24] and aggregate data about video quality and quality switches, available bandwidth, video status (playing, paused, rebuffering, ended, not started) and timings. In order to implement this browser-based monitoring we embed the video into a custom web page.

## 3.4 Postprocessor

**Postprocessor.** During an experiment two files are created: The HAR file of the video session and a file containing metadata collected by an extractor. A selection of different facts we want to extract from an HAR file:

1. How much data was downloaded?
2. How much bandwidth was available at what time during the video session?
3. How much of the bandwidth was spent on downloading video segments?
4. How long where the video segments and at what time where they played?
5. What was the quality of the video segments?
6. How much bandwidth was wasted by downloading segments that where never played?

To answer those questions, we implemented the *VimeoHarparser* and *Youtube-Harparser* classes.

Questions 1 to 4 can be answered with just the data contained in the HAR file and don't require any additional information. For the other questions we need to handle YouTube and Vimeo differently.

**Vimeo specifics.** For Vimeo we can directly use the same resource their adaptive bitrate streaming algorithm uses: A file which is downloaded at the start of every video session called *master.json*. Contained in this file are the URLs to obtain a specific segment in a specific resolution with a specific start- and endtime. Each such segment has a unique url. For Vimeo each segment is of a fixed length (usually 6 seconds). We map each request saved in our HAR file with the corresponding entry in the master.json file.

**Youtube specifics.** Youtube doesn't download a master.json file as vimeo does and also doesn't have a unique url for each possible segment. Instead the client sends parameters in the HTTP request headers. We are interested in two parameters: *itag* and *range*. *itag* denotes the format of the current segment and

*range* is the byte range parameter [17]. There is a large list of supported formats by YouTube which is quickly growing, for a list of them see [25]. To translate the byte range to playtime seconds we download the video in the format identified by the *itag* parameter with [26]. Then we decode the downloaded containers [27].

Postprocessor also offers ways to plot processed data. We use pygal [28] as underlying plotting library.

## 3.5 Adding support for other platforms

To add support for another adaptive streaming provider two key components need to be added:

- *A platform specific extractor:* An extractor to conduct the experiment. The extractor should inherit from *BaseExtractor*. *BaseExtractor* defines the expected methods and fields and makes sure that the extractor is compatible with the existing setup.
- *A platform specific HAR parser:* To actually process the collected data a platform specific HAR parser is necessary. It is expected that the parser inherits from *HarBase*. *HarBase* is an abstract class defining necessary methods and fields. In particular we need to make the data comparable to the data obtained on other streaming platforms. As an example *HarBase* requires the implementation of a method *starttime*, which should return the time at which the videoplayback started.

## 3.6 Measuring video quality on Vimeo

In this section we take a look at different ways to measure video playback quality in Vimeo. It is an example of how we extracted parameters from video platforms and made them comparable.

To measure video quality on Vimeo we have two options:

1. *Using the JavaScript API*: The extractor running a video session on Vimeo could use the JavaScript API [24] provided and poll for quality changes **during** playback.
2. *Analyzing the request headers*: We are capturing a HAR (see section 2.2.2) file and we can analyze the request headers **after** the video session.

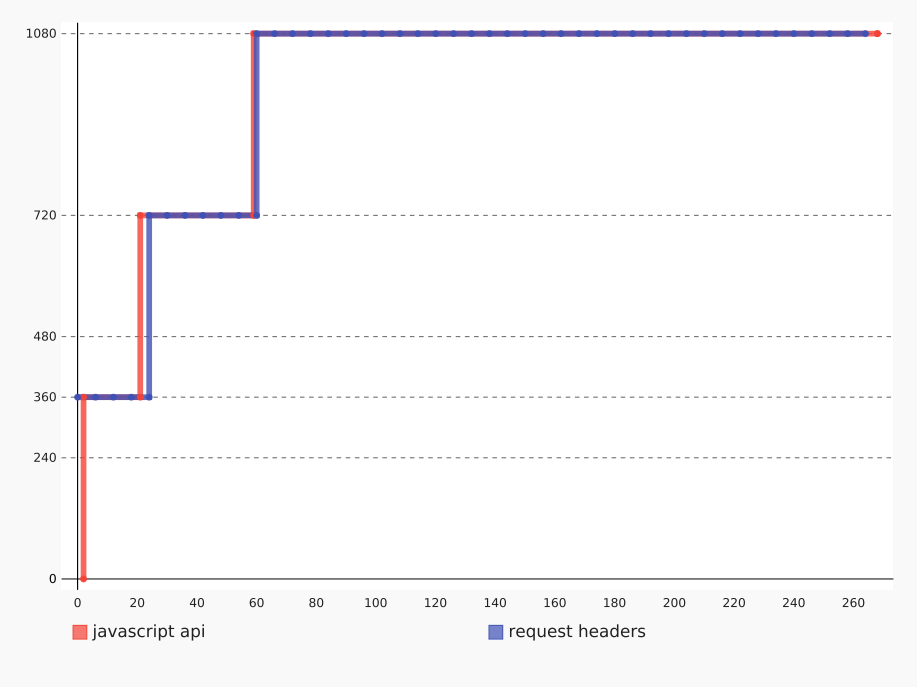


Figure 3.3: Playback quality during a video session

**JavaScript API** For the first option we use the *getVideoWidth()* and *getVideoHeight()* methods. The documentation of [24] about *getVideoWidth()* states:

> Get the native width of the currently-playing video.

**Request headers** For the second approach we capture a HAR [19] file during video playback. First we extract the "master.json" file, a file containing meta data used by Vimeo to map URLs to segments stored on the video server. We filter all request made that are not directly related to the video playback and use the master.json file to learn which of the remaining requests map to which quality.

**Figure 3.3** In Figure 3.3 we see the quality of one video session as reported by both methods. We used our usual testing setup for this experiment. To see quality switches we started with a bandwidth of 3000 kbit and raised it by 1000 kbit every 10 seconds. There is a slight difference of bandwidth reported by the api and by looking at the request headers. The JavaScript API sometimes reports quality switches a bit earlier than they actually happen. Further testing revealed that the JavaScript API does not report the quality of the currently-playing video, but the quality of the next video chunk. Since the video chunks served by Vimeo are up to 6 seconds long, the JavaScript API reports quality switches up to 6 seconds too early.

# Chapter 4

# Results

In this chapter we run several experiments measure how fair or unfair the current ABR algorithms of Vimeo and YouTube are. We look at video sessions by comparing different data points such as bandwidth, data downloaded, resolution of videos played and buffer level. We find that YouTube always gets a larger share of the bandwidth than Vimeo.

**Summary of the experiments**

1. *YouTube competing with Vimeo:* In section 4.2 and 4.3 we compare YouTube and Vimeo with both clients starting at the same time. In section 4.4 we take a look at what happens if YouTube or Vimeo is started later than the other client. We come to the conclusion that YouTube usually gets a larger share of the available bandwidth.
2. *Two YouTube instances competing with each other:* In section 4.5 we test if a YouTube client that is started while another instance is already running an advantage or a disadvantage. We conclude that timing isn't important for competing YouTube players and that they will eventually get an equal share of the available bandwidth.
3. *Google Chrome or Firefox*: In section 4.6 we show that it makes no difference if Firefox or Google Chrome is used for any of the experiments conducted.

## 4.1 Methodology

**Methodology.** We play the same video in parallel on YouTube and Vimeo. The video played is available in a wide range of qualities (from 360p to 4k) on Vimeo and on YouTube. During the video session we limit the available bandwidth with the shaper module introduced in Chapter 3 to 5000 kilobit/s. During playback a HAR file and additional metadata is collected for YouTube and Vimeo.

## 4.2 YouTube competing with Vimeo

Here we show four different diagrams depicting several aspects of the same setup. We start with a very simple experiment, were YouTube and Vimeo begin playing at the same time and share the same bottleneck of 5000 kbit/s. We look at the experiment in terms of total amount of data downloaded, bandwidth over time and video quality over time. This experiment shows that under stable network conditions YouTube always gets a larger share of the bandwidth. The video [1] has a wide range of different bitrates available.

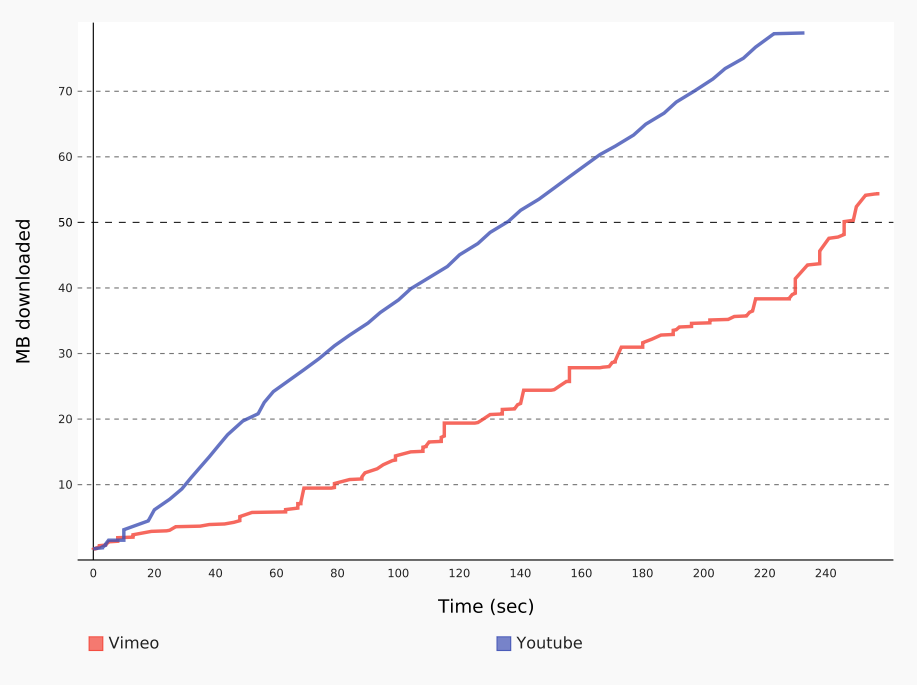


Figure 4.1: Total amount of data downloaded. Constant 5000 kbit/s bottleneck.

**Figure 4.1** Even without looking at user perceived QoE we immediately see that YouTube got a much larger share of the available bandwidth in Figure 4.1. While Youtube was able to deliver the video in 1080p resolution for most of the time, Vimeo delivered a lower quality. Only after about 200 seconds when YouTube finished downloading all the segments of the video Vimeo got more bandwidth. Figure 4.1 represents a specific playback and will change from video session to video session - but repeating the experiment always wielded similar results and we believe that Figure 4.1 is representative for a video session under the conditions specified.

[1] https://www.youtube.com/watch?v=6D-A6CL3Pv8, https://vimeo.com/93003441

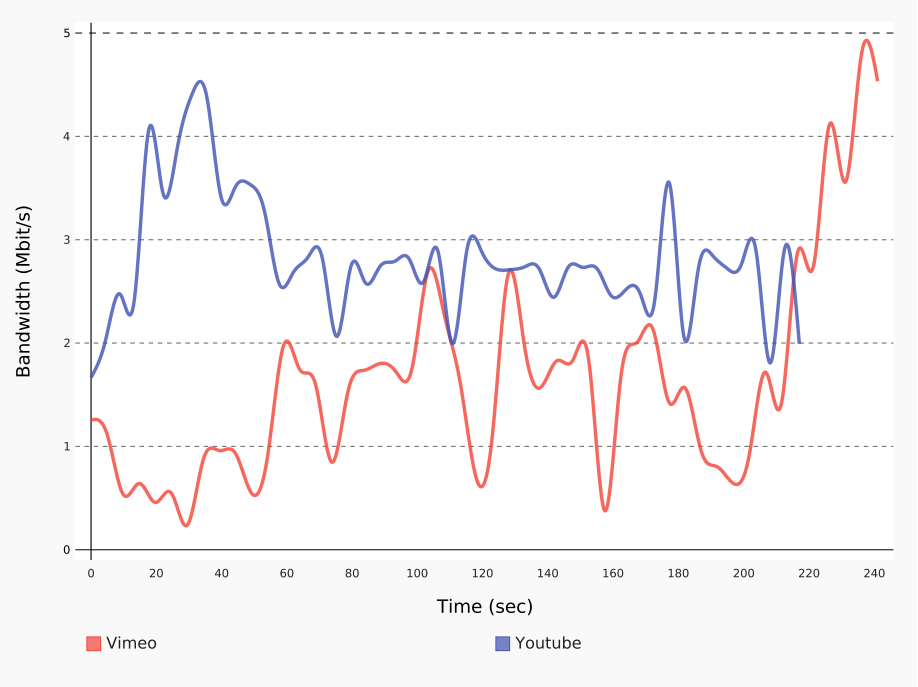


Figure 4.2: Bandwidth over time. Constant 5000 kbit/s bottleneck.

**Figure 4.2** Figure 4.2 shows the exact same experiment as figure 4.1 but this time we show a 15 seconds moving average of the bandwidth available instead of the total amount of data downloaded. Initially YouTube gets a much larger share of the bandwidth, then starting roughly after 60 seconds the bandwidth available to YouTube and to Vimeo seems to equalize. YouTube finishes downloading the last segment after approximately 220 seconds and Vimeo starts claiming the available bandwidth.

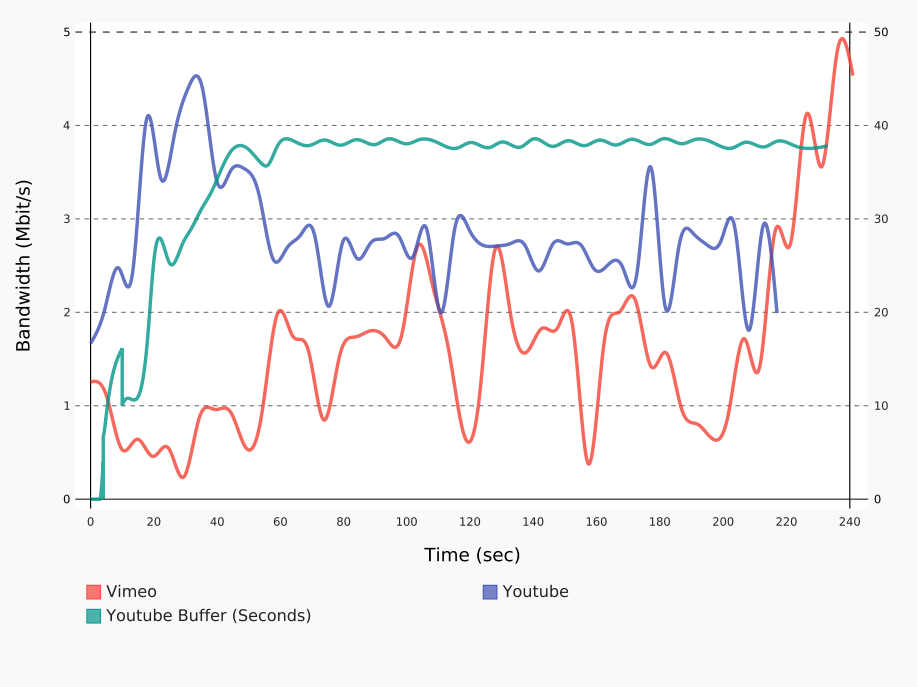


Figure 4.3: Bandwidth and YouTube's playback buffer over time. Constant 5000 kbit/s bottleneck.

**Figure 4.3** We now overlay the buffer that is available in the YouTube player over figure 4.2. Here, the available buffer is presented in seconds downloaded but not yet played (the y-axis legend at the right). In the first 20 seconds of the video we see the playback buffer filling up quickly, then it drops to zero before finally filling up again. This is explained by the changing quality at the start of the video. YouTube starts streaming the video in 1280x720 pixels (720p) resolution and later switches to 1920x1080 pixels (1080p). More importantly looking at YouTube's available playback buffer also reveals the reason why YouTube backs off after 60 seconds and lets Vimeo take a larger share of the bandwidth. YouTube simply reached a threshold where it is not interested in further growing its playback buffer.

In a talk given by Google's YouTube team in 2015 [16] they mentioned that YouTube tries to grow its buffer to approximately 40 seconds of playback time, we were able to confirm that they still use the same number.

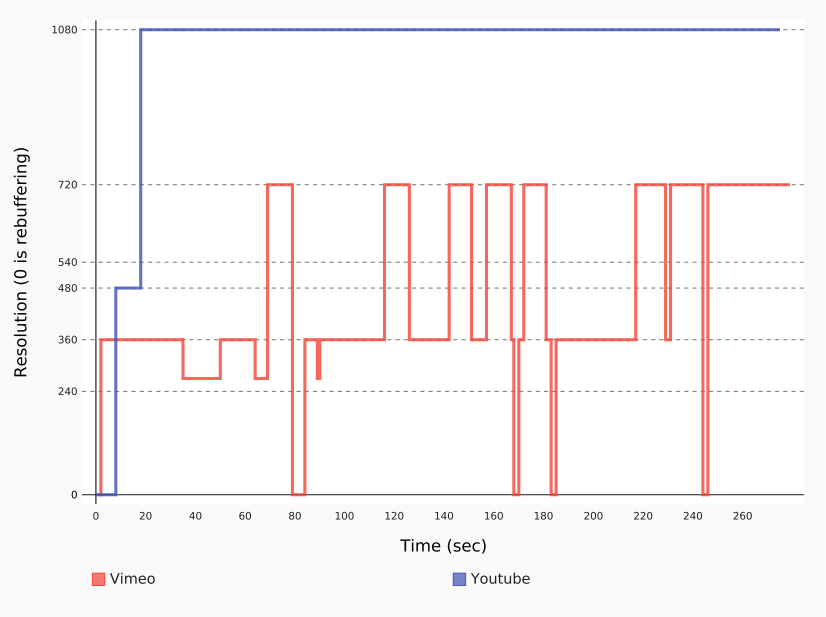


Figure 4.4: Quality delivered during video playback. Constant 5000 kbit/s bottleneck.

**Figure 4.4** Lastly we want to show what quality was streamed during the video session by YouTube and Vimeo. Notice that a resolution of 0 is a rebuffering event. We see that YouTube goes for 720p immediately and is forced to rebuffer at the very start to jump to that quality. Vimeo has a smaller start-up delay but delivers lower quality segments. We see YouTube claiming a higher share of the bandwidth quickly and Vimeo never being able to compete. This leads us to our next question: Can this observed unfairness be explained with unfortunate timings? Maybe if Vimeo was allowed to start before YouTube and could stream a high bitrate for some time, YouTube would not be able to secure a larger share of the total bandwidth?

**Significance of this experiment** We showed that under fairly normal conditions, a single limited link shared by two different clients, TCP alone isn't enough to guarantee fairness. The most interesting bits this experiment revealed, in no particular order:

*Unfairness*: Competing players don't get an equal share of the available bandwidth. In particular YouTube gets a larger share of the bandwidth than Vimeo.

*YouTube's behavior*: YouTube is overly aggressive while building its buffer until it has accrued roughly 40 seconds of downloaded but not yet played

data.

*Startup*: The first few seconds of the video playback reveal that YouTube tries to claim a large share of the available bandwidth by choosing high-quality segments even if rebuffering events happen, potentially to sway competing players to choose lower-quality segments.

We thus conclude that YouTube's adaptation algorithms compete unfairly with Vimeo.

## 4.3 YouTube competing with Vimeo: A longer video session

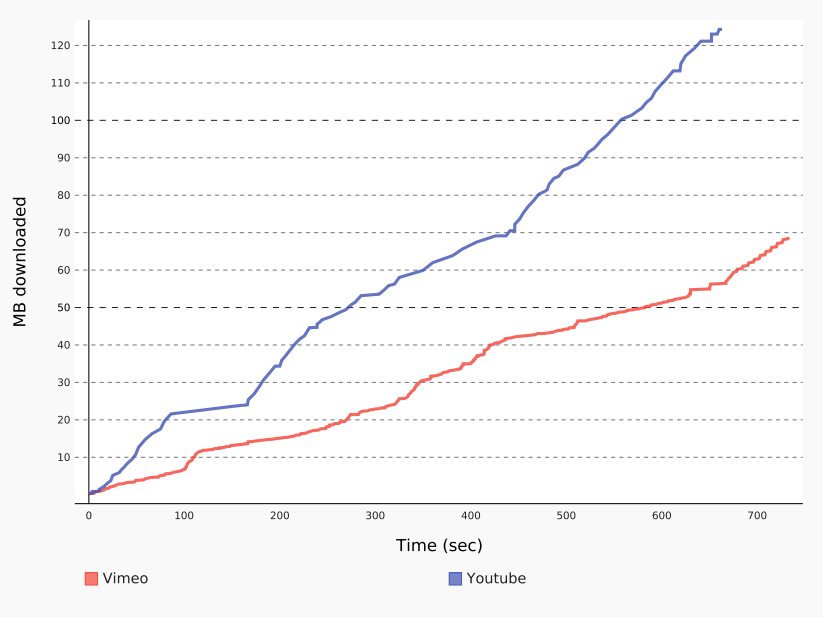


Figure 4.5: Total amount of data downloaded. Constant 3000 kbit/s bottleneck.

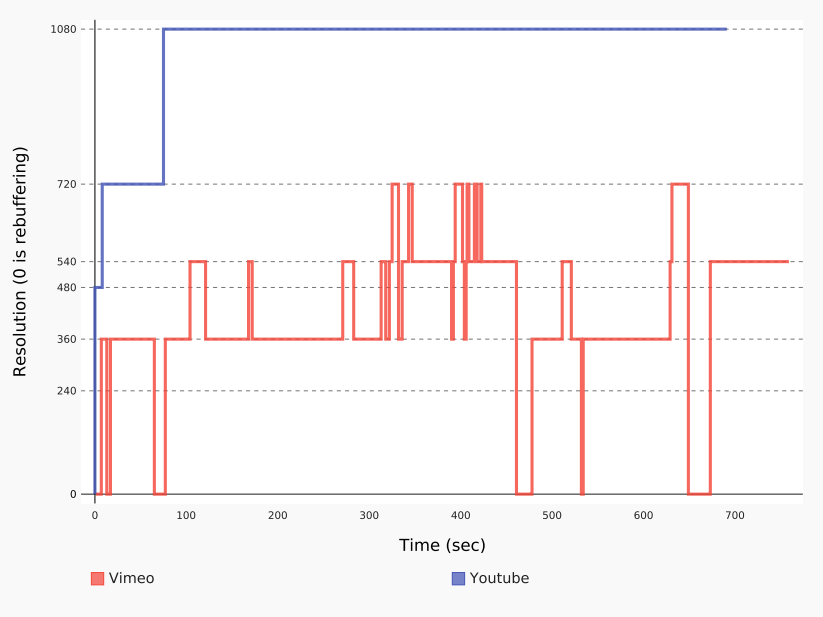

Figure 4.6: Quality delivered during video playback. Constant 3000 kbit/s

**Constant Bitrate: Experiment 2** We want to show that unfairness between YouTube and Vimeo also happens during a longer video session. For this experiment we limit the bandwidth to 3000 kilobit/s and choose a 11 minutes and 31 seconds long video [2].

**Results** YouTube gets a larger share of the available bandwidth (the bandwidth is not directly pictured, but the total data downloaded is represented in figure 4.5). The observed unfairness of the experiment in section 4.2 doesn't resolve automatically after some time.

The experiment revealed another interesting property: Vimeo changed playback quality surprisingly often (see 4.6). We noticed in all our experiments that YouTube is much more reluctant than Vimeo to switch video quality.

## 4.4 Are Latecomers penalized: Experiment 3

To make sure that the previous results are not just caused by unfortunate timings we test what happens if we start YouTube later than Vimeo and vice versa. Again we limit the shared bandwidth to 5000 kbit/s. With this experiment we show that the results of the previous section are independent of timings. We use the same video as in section 4.2.

[2] https://www.youtube.com/watch?v=B7RrPMvLFqI, https://vimeo.com/255482875

### 4.4.1 Vimeo first

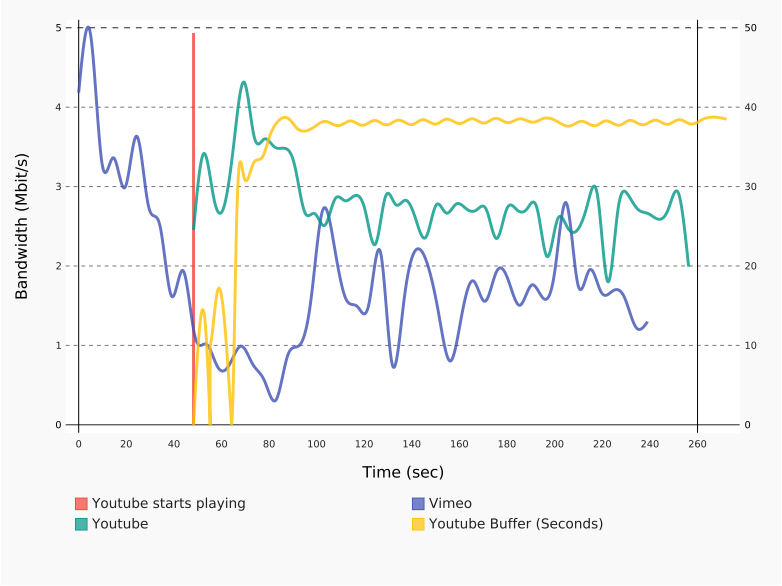


Figure 4.7: Bandwidth. YouTube is the latecomer.

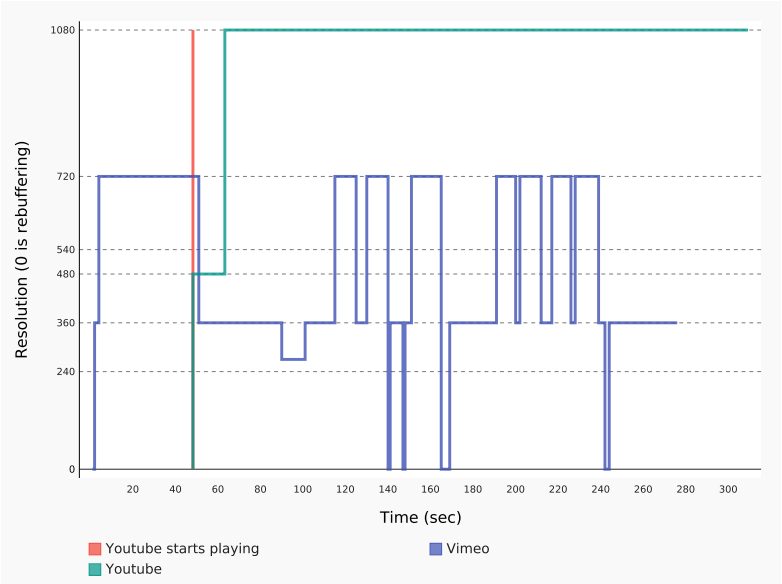


Figure 4.8: Quality. YouTube is the latecomer.

### 4.4.2 YouTube first

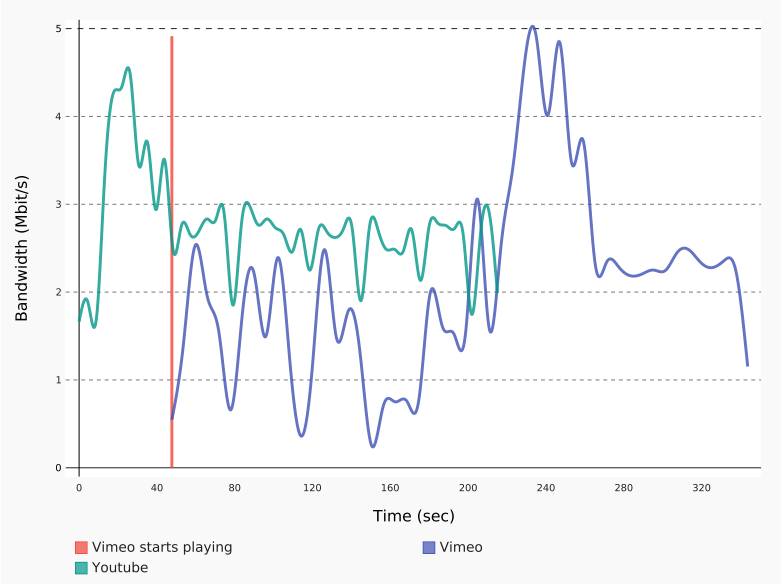


Figure 4.9: Bandwidth. Vimeo is the latecomer.

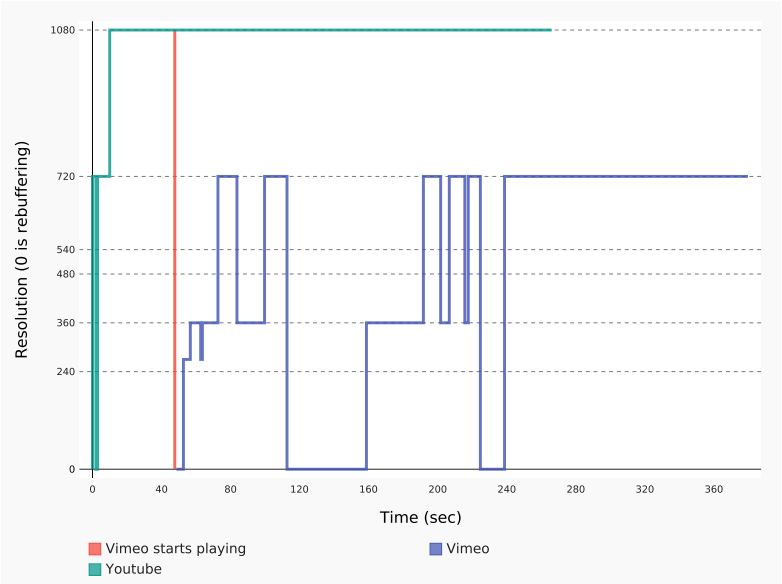


Figure 4.10: Quality. Vimeo is the latecomer.

**Timings** The figures presented show a very similar pattern as the figures presented in the previous section, where we started both players at the same time. In both experiments, in the first one, where we started Vimeo first, and in the second one, where we stared YouTube first, the bandwidth wasn't distributed fairly. YouTube always managed to deliver a higher bitrate and got a larger share of the bandwidth. In both experiments Vimeo even had to rebuffer while YouTube was playing the video at the highest resolution.

**Significance of this experiment** We conclude that the timings don't matter in this particular case and search for possible reasons why YouTube is so dominating elsewhere.

## 4.5 Two competing YouTube clients

In this section we want to analyze what happens if two YouTube clients compete with each other while sharing a limited bottleneck. We first present two experiments showing that a latecomer doesn't have an inherent disadvantage. Then we show a third experiment supporting our claim that, with competing YouTube players, their share of the available bandwidth eventually converge to a similar level.

**Experiment 3 and 4** We limit the available bandwidth to 5000 kilobyte/s and start a first YouTube client. After 45 seconds we start a second YouTube client. We present the bandwidth used and the quality of video segments served by both players on figure 4.11, 4.12, 4.13 and 4.14.

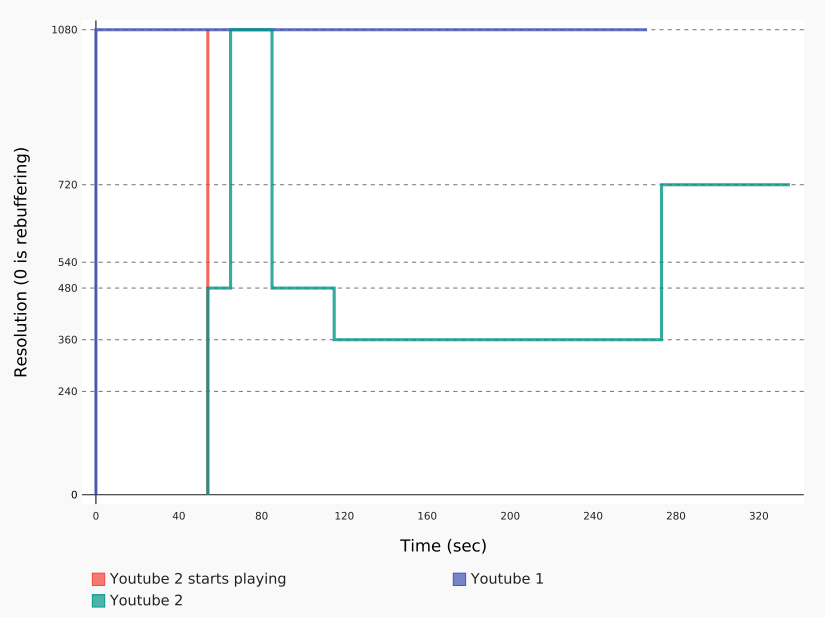


Figure 4.11: Experiment 3: Quality. Constant 5000 kbit/s

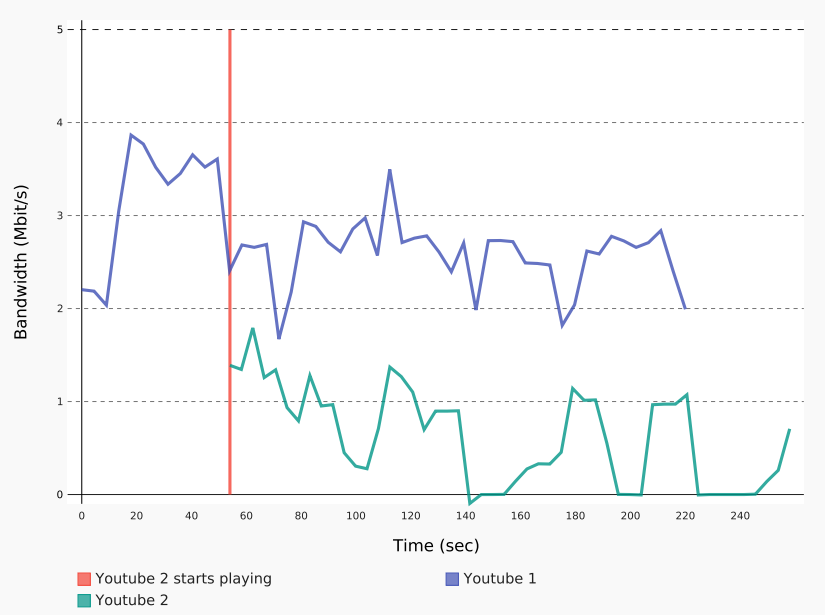


Figure 4.12: Experiment 3: Bandwidth. Constant 5000 kbit/s

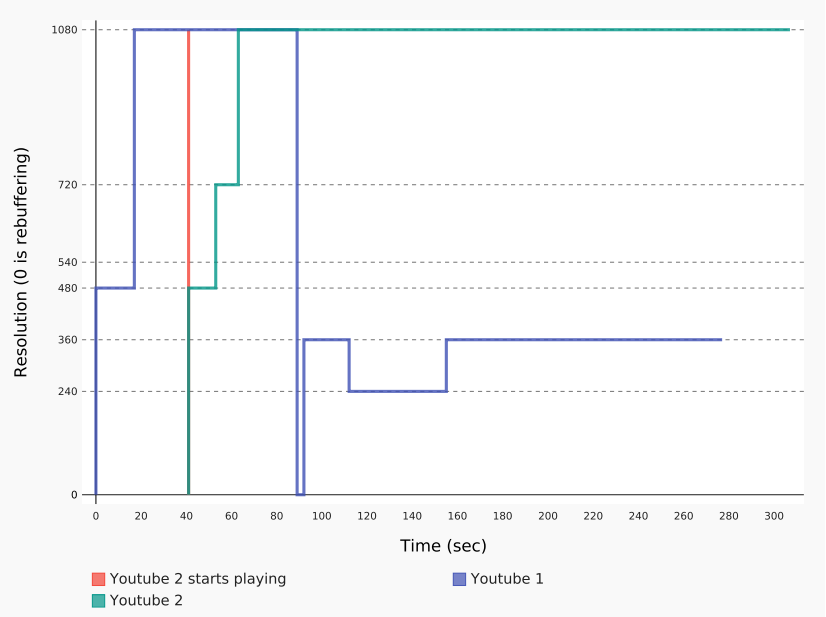


Figure 4.13: Experiment 4: Quality. Constant 5000 kbit/s

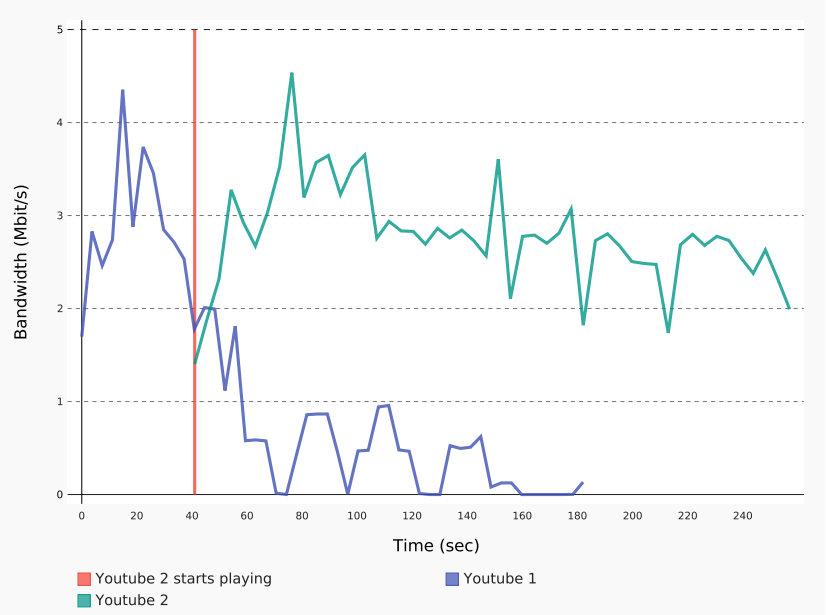


Figure 4.14: Experiment 4: Bandwidth. Constant 5000 kbit/s

**Figure 4.11, 4.12, 4.13 and 4.14** In figure 4.11 the first client got a much larger share of the bandwidth than the second client. In figure 4.13 it was the

other way around: The second client gets a larger share of the bandwidth than the first player. Of course, the client with the higher bandwidth also delivers video segments in a higher resolution.

**Experiment 5** Further experiments revealed that competing YouTube clients eventually achieve fairness. An example is shown in figure 4.15 and 4.16 which represent the quality and bandwidth used by the clients in experiment 5. While the example shown achieves a roughly equal share of the bandwidth after about 5 minutes, we have also encountered situations in which it took YouTube significantly longer (up to 10 minutes) to achieve fairness.

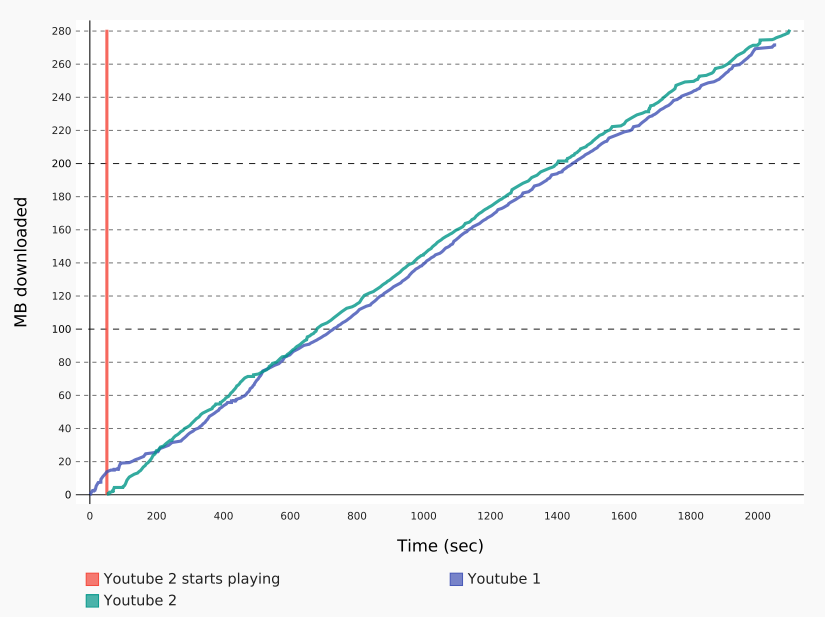


Figure 4.15: Experiment 5: Quality. Constant 3000 kbit/s

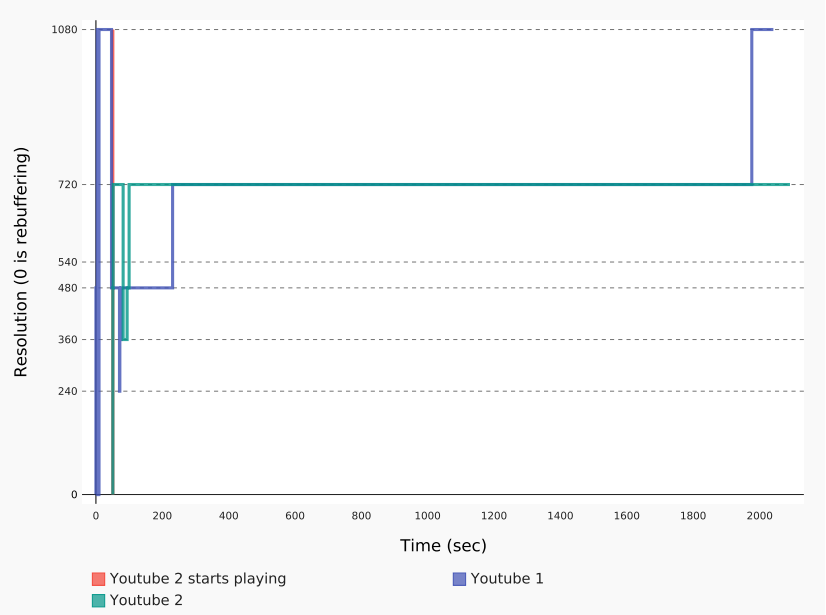


Figure 4.16: Experiment 3: Bandwidth. Constant 3000 kbit/s

## 4.6 Google Chrome

We repeated various experiments with Google Chrome instead of Firefox. In particular we tested:

1. A Vimeo client using Chrome competing with a YouTube client using Chrome
2. Two YouTube clients both using Chrome competing with each other
3. A YouTube client using Chrome competing with a YouTube client using Firefox

**QUIC** For this experiment we had the QUIC [3] flag on default [4]. This resulted in QUIC being disabled [5]. The same flag resulted in QUIC being enabled if chrome is not controlled by automated test software. It would be interesting to see how the results change if we could force Chrome to use QUIC, especially in combination with BBR (a congestion control algorithm), but at present this capability is not exposed to users.

**Results** We repeated each experiment ten times and compared the average share of the bandwidth used by each client. We found no difference to the analogue experiments using Firefox.

[3] QUIC [29] is a network protocol developed by Google
[4] The "Experimental QUIC protocol" setting found on chrome://flags/
[5] verified by consulting chrome://net-internals/

# Chapter 5

# Conclusion

## 5.1 Overview of the results

**YouTube competing with Vimeo**

- YouTube usually gets a larger share of the available bandwidth and is able to deliver its video segments in a higher quality.
- It does not matter how long playbacks are: YouTube and Vimeo never converge to a similar share of the available bandwidth. TCP is not enough to guarantee fairness.
- It does not matter if Firefox or Google Chrome is used.

So far we have seen that YouTube beats Vimeo in terms of quality delivered and amount of rebuffering events but we haven't tried to explain those results. We see two key factors contributing:

1. *YouTube uses a more aggressive ABR algorithm during the start-up phase*: We have seen that YouTube tries to fetch high-quality segments right from the start and doesn't adjust even if there are short rebuffering events. Vimeo seems to take a slower, more careful approach. This results in the situations observed, where YouTube wins out.
2. *YouTube's file size is smaller*: YouTube uses *WebM* as container with *VP9* and *Vorbis* or *Opus* as video and audio codecs while Vimeo uses *mp4* as container and *H.264* as video codec. While the specifics are out of scope for this thesis we observed that in many cases YouTube has a smaller file size for the same video in the same quality. This allows YouTube to be more efficient than Vimeo, allowing it to deliver higher quality.

**YouTube competing with YouTube**

1. Eventually, the bandwidth shares of two competing YT players will converge to roughly equal values. In our experiments this took up to ten minutes.

2. It does not matter if Firefox or Google Chrome is used.

## 5.2 Unresolved issues and opportunities for future work

**Extending the testing environment** At the moment our platform supports Vimeo and YouTube, but there are other large platforms, such as Netflix, and there will be new platforms someday. We believe that adaptive streaming will stay an active topic of research for quite some time.

**Why are the splits of the available bandwidth so unfair?** This thesis shows that competing adaptive streaming clients don't automatically get a fair share of the available bandwidth under normal TCP. We explain this with YouTube's more aggressive start-up phase and smaller file size, but additional work will be necessary to determine whether there are other contributing factors.

# Bibliography


[1] "Delivering Live YouTube Content via Dash." https://developers.google.com/youtube/v3/live/guides/encoding-with-dash.

[2] "Netflix Tech Blog: Optimizing the Netflix Streaming Experience with Data Science." https://medium.com/netflix-techblog/optimizing-the-netflix-streaming-experience-with-data-science-725f04c3e834.

[3] "The Vimeo Blog: Adaptive streaming." https://vimeo.com/blog/post/adaptive-streaming-and-4k-coming-soon-for-your-vid.

[4] H. Mao and R. N. aand Mohammed Alizadeh, "Neural adaptive video streaming with pensieve," in *SIGCOMM 2017*.

[5] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over http," in *SIGCOMM 2015*.

[6] S. Akhshabi, "What happens when http adaptive streaming players compete for bandwidth?," in *NOSSDAV 2012*.

[7] A. Mansy, M. Fayed, and M. Ammar, "Network-layer fairness for adaptive video streams," in *IFIP 2015*.

[8] X. Yin, M. Bartulovic, V. Sekar, and B. Sinopoli, "On the efficiency and fairness of multiplayer http-based adaptive video streaming," in *ACC 2017*.

[9] "Cisco VNI: Forecast and Methodology, 2016-2021, April 2018."

[10] F. Dobrian, A. Awan, D. Joseph, A. Ganjam, J. Zhan, V. Sekar, I. Stoica, and H. Zhang, "Understanding the impact of video quality on user engagement," in *SIGCOMM 2011*.

[11] "Testing environment written for this thesis." https://github.com/alskgj/quantifying_unfairness.

[12] "Dash-Industry-Forum: dash-js." https://github.com/Dash-Industry-Forum/dash.js/wiki.

[13] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari, "Confused, timid, and unstable: Picking a video streaming rate is hard," in *IMC 2012*.

[14] S. Akhshabi, A. C. Begen, and C. Dovrolis, "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http," in *MMSys 2011*.

[15] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson, "A buffer-based approach to rate adaptation: Evidence from a large video streaming service," in *SIGCOMM 2014*.

[16] "Google I/O 2013 - Adaptive Streaming for You and YouTube." `https://www.youtube.com/watch?v=UklDSMG9ffU`.

[17] A. Mondal, S. Sengupta, B. R. Reddy, M. Koundinya, C. Govindarajan, and P. De, "Candid with youtube: Adaptive streaming behavior and implications on data consumption," in *NOSSDAV 2017*.

[18] "BrowserMob Proxy." `https://github.com/lightbody/browsermob-proxy`.

[19] "HTTP Archive 1.2 Spec." `http://www.softwareishard.com/blog/har-12-spec/`.

[20] "Selenium - Web Browser Automation." `https://www.seleniumhq.org/`.

[21] "Selenium WebDriver: Reference, 2018." `https://www.seleniumhq.org/docs/03_webdriver.jsp`.

[22] C. Sieber, P. Heegaard, T. Hossfeld, and W. Kellerer, "Sacrificing efficiency for quality of experience: Youtube's redundant traffic behavior," in *IFIP 2016*.

[23] "YouTube Player API Reference for iframe Embeds." `https://developers.google.com/youtube/iframe_api_reference`.

[24] "Vimeo Player API." `https://github.com/vimeo/player.js`.

[25] "Formats and Resolutions of Youtube Videos." `http://www.genyoutube.net/formats-resolution-youtube-videos.html`.

[26] "youtube-dl." `https://rg3.github.io/youtube-dl/`.

[27] "ffprobe." `https://www.ffmpeg.org/ffprobe.html`.

[28] "Pygal - Sexy python charting." `http://pygal.org/en/stable/`.

[29] "QUIC, a multiplexed stream transport over UDP." `https://www.chromium.org/quic`.