

Bendwit: Platform for preparing content for adaptive streaming

Author
Joris Herbots

Promotor
Prof. dr. Peter Quax

Co-promotor
Prof. dr. Wim Lamotte

Mentor
dr. Maarten Wijnants

2016-2017



Acknowledgements

Firstly, I would to thank my mentor Dr. Maarten Wijnants for his continuous guidance, motivation and support of my bachelor thesis. His insightful feedback and knowledge have been of great value to me and have helped me a lot during the writing and development of this thesis. I could not have imagined having a better mentor for my bachelor thesis.

I would also like to thank my promotors professor Peter Quax and professor Wim Lamotte for providing this unique opportunity. Without their support, it would not have been possible to research this thesis subject.

My gratitude also goes to my fellow students and friends for their support and many stimulating discussions. In particular, my thanks goes to Maarten Vangeneugden for helping me come up with the name *Bendwit*.

Last, but not least, I would like to thank my parents and my sister for their moral support during the writing of this thesis and my life in general.

List of Abbreviations

API	Application Programming Interface
ARF	Alternate Reference Frame
ASIC	Application-Specific Integrated Circuit
CLI	Command Line Interface
CODEC	Encoding & Decoding
CORS	Cross-Origin Resource Sharing
CSV	Comma-separated values
CPU	Central Processing Unit
DASH	Dynamic Adaptive Streaming over HTTP
DASH-IF	DASH Industry Forum
FIFO	First in, first out
FPGA	Field-Programmable Gate Array
FPS	Frames Per Second
GRF	Golden Reference Frame
GOP	Group of pictures
HAS	HTTP Adaptive Streaming
HDS	Adobe HTTP Dynamic Streaming
HEVC	High Efficiency Video Coding
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commission
IIS	Internet Information Services
ISO	International Organization for Standardization
JCT-VC	Joint Collaborative Team on Video Coding
JSON	JavaScript Object Notation
KB	Kilobyte
MPD	Media Presentation Description
MSS	Microsoft Smooth Streaming
NAT	Network Address Translation
OS	Operating System
REST	Representational state transfer
RTP	Real-time Transport Protocol
RTSP	Real-time Streaming Protocol
SS	Microsoft Smooth Streaming
TCP	Transmission Control Protocol
TTML	Timed Text Markup Language
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
VCEG	Visual Coding Experts Group
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language

Contents

Acknowledgements	i
Abstract	v
Dutch Summary	vi
1 Introduction	1
1.1 Problem statement	1
1.2 Bachelor Thesis Outline	2
1.3 Competitors	2
1.3.1 Bitmovin	2
1.3.2 Coconut	2
2 Adaptive Streaming	3
2.1 Strengths of Adaptive Streaming over HTTP	4
2.1.1 Device specifications	4
2.1.2 Multiple representations	5
2.1.3 HTTP	5
2.1.4 Live Content versus On-Demand Content	5
2.2 Dynamic Adaptive Streaming over HTTP (DASH)	6
2.2.1 Inner workings	6
2.2.2 Supported codecs	7
2.2.3 Manifest	7
2.3 HTTP Live Streaming (HLS)	10
2.3.1 Manifest	10
2.4 Other Adaptive Streaming Implementations	11
2.4.1 Microsoft Smooth Streaming	11
2.4.2 Adobe HTTP Dynamic Streaming	11
3 Used Tools and Codecs	12
3.1 Tools	12
3.1.1 FFmpeg	12
3.1.2 FFprobe	14
3.1.3 MP4Box	16
3.2 Codecs	16
3.2.1 H264	16
3.2.2 VP8	18
3.2.3 AAC	18
4 Bendwit	19
4.1 Bendwit workflow	19
4.2 Command Line Tool	21
4.2.1 Choice of Programming Language	21
4.2.2 Inner workings	21
4.2.3 Configuration File	22
4.2.4 Currently supported codecs	27
4.2.5 Encountered problems	28

4.3	Job Scheduler	31
4.3.1	Choice of Programming Language	31
4.3.2	Inner Workings	31
4.4	REST API	33
4.4.1	Choice of Programming Language	33
4.4.2	NGINX	33
4.4.3	Inner Workings	34
4.5	Website	39
4.5.1	Choice of Programming Language	39
4.5.2	Inner Workings	39
4.6	Python Developer API	49
4.6.1	Functional methods	50
4.6.2	Object-Oriented Abstraction	51
5	Evaluation - Time Study	53
5.1	Test Cases	53
5.2	Findings	55
6	Conclusion	58
7	Future work	60
7.1	Containerized Nodes	60
7.2	Bendwit CLI Performance Boost	60
7.3	Interactive REST API	61
	Appendices	62
A	JSON Schemas	63
A.1	Probes Schema	63
A.2	HAS Transcodes Schema	63
A.3	Media Schema	66
B	Evaluation Configurations	68
B.1	Sintel Open Movie REST API Configuration	68
B.2	Sintel Open Movie Manual Commands	70

Abstract

Media sharing, in particular video and audio, is becoming more and more popular these days and forms the bulk of all Internet traffic. People share their personal video recordings on social media like Facebook¹, amateur creators share their works on Youtube² and Vimeo³, professional media content providers like Netflix⁴ stream movies and series, ... All this media is then watched by millions of people all using different playback devices. Because of the heterogeneous group of playback devices, all with their own respective capabilities, it is not easy to create a piece of media which fits all these devices' playback capabilities. This is where adaptive streaming comes into play, adaptive streaming makes it possible to stream media to a wide range of playback devices over a variety of different connection speeds. The media itself is made available in different qualities which the client playback devices gets to choose from. Depending on the current environment, the client playback device will pick the best possible quality and continue switching between qualities if the environment changes during playback. An environment consist of the current available network connection, the device its capabilities, the media resolution, ...

This thesis consists out of two big parts. The first part consists of an introduction into the world of streaming and encompasses an in depth chapter about adaptive streaming itself. Older adaptive streaming protocols are compared to the newer variants. This chapter also describes the DASH and HLS adaptive streaming implementations in detail, these two implementations are currently the standard ones for adaptive streaming.

The second part describes a platform called *Bendwit*, which solves to problem of preparing media for adaptive streaming. Adaptive streaming is a very good solution to the playback device diversity problem and also the network throughput variability, but it introduces the need for encoding the media into different qualities, segmenting these qualities and finally creating a manifest file which tells playback devices what qualities are available and where to find them. This is not a trivial step and requires some knowledge about streaming. Bendwit tries to solve this problem for novice users by letting them prepare media for adaptive streaming by means of premade *profiles*, which are adaptive streaming qualities that try to reach as many playback devices as possible. Expert users on the other hand can use the Bendwit platform directly and manually configure the different quality settings. The platform itself consists of five different parts: the Bendwit CLI tool, the job scheduler, the REST API, the website and the developer API. The Bendwit CLI tool forms the core of the platform and encompasses all logic pertaining to the preparation of media content for adaptive streaming, it handles the creation of different qualities, the segmentation of these qualities and finally the creation of a manifest file. Users interact with the platform via the REST API which schedules the user request via the job scheduler. If the user desires a simpler way of interaction with the platform, they can do so by using the website interface or the developer API.

The full adventure of discovering the world of multimedia and the programming experience gained by creating a platform which incorporates all this knowledge is described in the upcoming chapters. These findings may be useful for people who are new to this world and want to learn more about adaptive streaming and how to prepare content for adaptive streaming.

¹<https://facebook.com/>

²<https://www.youtube.com/>

³<https://vimeo.com/>

⁴<https://www.netflix.com/browse>

Dutch Summary

Het delen van media, meer bepaald video en audio, is een steeds vaker voorkomend fenomeen dezer dagen. Het grootste deel van het Internetverkeer bestaat uit video. Mensen delen steeds vaker hun videomateriaal op verschillende platformen en bekijken ze achteraf met verschillende apparaten. Hierin schuilt een groot probleem, elk apparaat heeft zijn eigen mogelijkheden en speelt dus ook enkel media af die hieraan voldoet.

Binnen de wereld van media heerst er een enorme diversiteit tussen de verschillende media container formaten en de manier waarop de media wordt gecodeerd en gedecodeerd, dit noemt men codecs. Naargelang de capaciteiten van het afspeelapparaat, zal deze al dan niet het media container formaat kunnen uitlezen en de hierin gemultiplexte gegevens decoderen. Op het web behoren op dit moment H264, H265, VP8 en VP9 tot de populairste video codecs. Echter is het ondersteunen van een codec op een afspeelapparaat niet voldoende. Zo zijn bijvoorbeeld de H264 en H265 codecs nog eens verder onderverdeeld in profielen en niveaus, deze stellen een bovengrens waaraan een afspeelapparaat moet voldoen om de gecodeerde media stream live te kunnen decoderen en zodoende een vloeiend beeld aan de gebruiker te tonen. Hieruit wordt al snel duidelijk dat deze diversiteit een probleem vormt indien de media aan een zo groot mogelijk doelpubliek getoond moet worden. Zo zal een zwakker afspeelapparaat zoals een smartphone niet capabel zijn om een hoge kwaliteit H264 gecodeerde video stream af te spelen, maar een moderne desktop daarentegen wel. Een oplossing zou zijn om de media in een lager niveau te encoderen, zodat deze ook afspelt op zwakkere afspeelapparatuur. Echter brengt dit dan weer kwaliteitsverlies met zich mee voor mensen die de media willen bekijken op een desktop.

Als oplossing voor het voorgaande probleem bestaat er adaptieve transmissie, een manier om een mediabestand aan de gebruiker te tonen zonder kwaliteitsverlies waarbij de grote diversiteit aan afspeelapparatuur toch ondersteund wordt. Deze thesis heeft als onderwerp het *voorbereiden van content voor adaptieve transmissie*. De focus ligt hierbij op twee zeer populaire adaptieve transmissie implementaties genaamd MPEG-DASH en HLS, dit zijn implementaties die gebruik maken van HTTP voor de transmissie van de media. Concreet werkt adaptieve transmissie door een mediabestand te transcoderen naar verschillende kwaliteiten. Typisch houdt dit in dat er minstens een lage kwaliteit aanwezig is voor de zwakkere afspeelapparaten en een hoge kwaliteit voor de meer geavanceerde afspeelapparatuur. Deze verschillende kwaliteiten worden vervolgens opgedeeld in kleinere segmenten van typisch 2 à 10 seconden. Uiteindelijk wordt er een manifest bestand aangemaakt dewelke een overzicht geeft van de verschillende kwaliteiten en de (netwerk) locaties communiceert waar een afspeelapparaat deze kwaliteitssegmenten kan vinden. Indien een afspeelapparaat een mediabestand dat klaar is gemaakt voor adaptieve transmissie wilt afspelen, downloadt deze simpelweg het manifest en bepaalt hieruit met de huidige omgeving welke kwaliteit te downloaden. De omgeving wordt bepaald door de huidige bruikbare Internetsnelheid, de resolutie van de video, de aspect ratio van de display van het afspeelapparaat, . . . Gedurende het afspelen van de media zal het afspeelapparaat voortdurend de optie hebben om segmenten van een andere kwaliteit op te vragen, wat een vlotte weergave van de media garandeert zonder haperingen (bv. wanneer de Internetsnelheid plots snel zou dalen).

Adaptieve transmissie mag dan wel een oplossing vormen voor de grote diversiteit aan afspeelapparatuur, het brengt een extra vereiste met zich mee op vlak van voorbereiden van de media voor adaptieve transmissie. Een verwerkingsstap die niet zozeer gemakkelijk is voor iemand die geen weet heeft van welke instellingen het beste zijn voor de gebruikte codecs of hoe een manifest er exact moet uitzien. Dit is waar het Bendwit platform bij komt kijken. Bendwit is een platform voor de voorbereiding van media voor adaptieve transmissie, het automatiseert de volledige verwerkingsstap voor de gebruiker. De werkelijke implementatie van het platform zelf bestaat uit vijf verschillende delen: de Bendwit CLI tool,

de job scheduler, de REST API, de website en de developer API. De kern van het platform bestaat uit de Bendwit CLI tool, deze voorziet de volledige logica voor het transcoderen van media naar verschillende kwaliteiten, het segmenteren van deze verschillende kwaliteiten en uiteindelijk de aanmaak van een manifest bestand met de correcte gegevens voor het afspelen van de media. Gedurende de ontwikkeling is er gekozen om hiervan een standalone tool te maken, zodat deze ook rechtstreeks bruikbaar is voor expert gebruikers via een CLI. Indien het platform echter in zijn geheel wordt gebruikt, sturen gebruikers via de REST API verzoeken om media voor te bereiden voor adaptieve transmissie. Dit doen ze door een aantal kwaliteiten te vermelden dewelke de Bendwit CLI tool zal trachten te gebruiken gedurende het transcoderen. Deze stap is echter al meer gericht naar expert gebruikers. Beginnende gebruikers kunnen ook de website interface gebruiken, deze biedt een abstractie aan genaamd *profielen*. Een profiel is een voorgemaakte bundel van kwaliteiten die aan een zeker doelpubliek voldoet. Bij keuze van een profiel weet een beginnende gebruiker dat zijn media zal werken voor het gekozen doelpubliek. De website biedt tevens ook sleutelpagina aan dewelke sleutels maakt voor de REST API.

Deze thesis bestaat uit twee gedeeltes. Het eerste gedeelte is een diepgaandere beschrijving van adaptieve transmissie samen met een overzicht van de meest belangrijke codecs en hoe deze werken. Het tweede gedeelte beschrijft het zonet beschreven Bendwit platform in detail.

Chapter 1

Introduction

Real-time media traffic forms the bulk of modern Internet trafficking. Video traffic in 2020 is estimated to be 82 percent of the total consumer traffic [1]. In order to keep up with modern standards, media like video and audio have to adapt to the new standards; including multiple audio-channel support, high resolution video, ... This all comes down to more bandwidth usage and thus requires a capable network architecture to carry this load or a smart way of retrieving the media. Streaming is one of the most used techniques for retrieving media from a server.

1.1 Problem statement

The Internet at the the time of writing is still growing. A lot of households in the more developed countries have reliable and fast Internet connections. Belgium for example has an average connection speed of 15,96Mbit per seconds [2]. Such networks can easily handle high quality media content streaming. But less reliable connections will suffer from problems ranging from long loading times to poor media quality. In the past fifteen years, we have also seen a remarkable growth in mobile devices. Reports by big companies like Google and Cisco show us that more and more people are using these mobile devices as a daily driver to do their online searching, banking, gaming, media streaming, ... [3] It is estimated that traffic from wireless and mobile devices will account for two-thirds of total IP traffic by 2020 [1]. Mobile connections are by nature inferior to wired connections. They are prone to security issues, less bandwidth, package loss and so on; making media streaming more problematic.

Media streaming, from small Youtube videos¹, Netflix series and movies² to Spotify music³ are just a small portion of what media streaming entails these days. Due to the worldwide stable network infrastructure, people are relying more and more on streaming for their media consumption. Previously mentioned companies take advantage of this fact to provide us with media streaming as a service.

But what exactly does media streaming mean? Streaming stands for the real-time transfer of data over a network. In the case of media like audio and especially video this requires a lot of bandwidth; bandwidth is the amount of data that can be transferred over a network, usually expressed in bits per second. As mentioned earlier, the amount of bandwidth available depends on the connection speed for your device. This available bandwidth is not infinite and must thus be used wisely when streaming. It immediately becomes apparent that media streaming is not an easy process due to these factors. This is why media providers have to carefully consider how to provide media, how to store the media, how to speed up loading times ...

This is where adaptive streaming comes into play. Over the recent years, a lot of genius minds have figured out ways to provide media content over the Internet whilst taking into account the aforementioned problems. This is what we call *Adaptive Streaming*: a way to stream media content to large

¹<https://www.youtube.com>

²<https://www.netflix.com>

³<https://www.spotify.com>

heterogeneous groups of consumers. Under heterogeneity, we understand differences in contextual factors like terminal capabilities and network conditions. Chapter 2 explains this further into detail.

Preparing media for streaming can form a problematic and tiresome issue. This thesis will implement a platform which automates this process and makes it accessible for novice users as well as expert users. The interaction can take place via a REST API, website or a developer API which all communicate with a tool which is responsible for preparing the media for streaming.

1.2 Bachelor Thesis Outline

This bachelor thesis consists of two parts. The first part of which is a research part where adaptive streaming is the theme. More specifically the modern equivalent HTTP Adaptive Streaming. The second part consists of an implementation that allows media content to be prepared for adaptive streaming called Bendwit. This platform will make use of the two most commonly used HAS implementations: DASH and HLS, further explained in Sections 2.2 and 2.3 respectively.

The platform itself will consist out of 4 parts:

1. A command line tool for media preparation
2. A REST API which allows interaction with the command line tool
3. A website which will provide graphical easy-to-use access to the platform and its features
4. A developer API for developers who want to use the platform in their coding projects

The implementation of the platform is further explained in Chapter 4.

1.3 Competitors⁴

There already exist platforms which do what this thesis implementation does. These platforms offer similar solutions for encoding and preparing media for adaptive streaming, yet typically operate on a much bigger scale. Sections 1.3.1 and 1.3.2 briefly showcase two popular alternatives to Bendwit.

1.3.1 Bitmovin

Bitmovin⁵ describes itself as a software to solve complex video problems and is one of the more popular cloud based encoding solutions on the market to this day. Their platform allows video encoding and preparing media for adaptive streaming; it has support for MPEG-DASH, HLS and provides a progressive download fallback [4]. The encoding services are accessible through a website interface or through an extensive API which can either be accessed directly or through a wide collection of client API's developed for the most commonly used programming languages⁶.

Bitmovin also provides an HTML 5 player capable of MPEG-DASH, HLS and progressive download playback with cross platform device compatibility as one of its main goals [5].

1.3.2 Coconut

Coconut⁷ describes itself as a cloud based encoding solution for developers. Compared to the bigger cloud based encoding solutions like Bitmovin, Coconut is more focused on one task: offering encoding solutions. This is noticeable in their API, which provides a programming-like way of specifying the encoding configuration by means of variables and control structures [6]. At the time of writing, Coconut provides MPEG-DASH and HLS support [7, 8].

⁴Even though this section is called "Competitors", Bendwit does not directly compete with these other platforms.

⁵<https://bitmovin.com/>

⁶<https://github.com/bitmovin>

⁷<http://coconut.co/>

Chapter 2

Adaptive Streaming

Streaming typically takes place between a server and a client. The server contains the media and presents it to the Internet through a public address. The client is considered to be the device the user employs to fetch the stream.

As mentioned in Chapter 1, many adaptive streaming implementations exist these days. Older streaming based solutions like *Real-time Streaming Protocol (RTSP)*, heavily rely on server CPU's to process and distribute media to clients. The client only sends commands to the server, so all playback, state and distribution logic resides on the server side. RTSP uses *Real-time Transport Protocol (RTP)*, which is an application layer protocol that uses UDP as its transport layer protocol for sending media to clients [9]. The media itself is sent over the network at the bitrate at which it is encoded. Since RTP uses UDP, it inherits the non-reliable transmission channel and stateless design; thus making packet loss an inevitable part of the design. Many clients also reside behind NAT devices, blocking incoming UDP requests. This combination made RTP a difficult to use streaming protocol [10].

Nowadays streaming protocols look at the TCP transport layer protocol for sending media packets or even at HTTP as the application layer protocol for sending media data. This introduces new problems in the mix; TCP keeps track of the transmission channel state and thus consumes extra bandwidth [11]. It also contains built-in algorithms to keep network congestion at a minimum. These extras do not always have the desired effect when streaming and actually hinder the process. But using HTTP also helps with the aforementioned network address translation problems faced by UDP. Due to the widespread availability of HTTP and supporting hard- and software, streaming media becomes easy and is widely popular these days.

There exist two main streaming solutions over HTTP; *Progressive download* is the oldest one of them. Progressive download is a technique where a server supplies media over HTTP and the client can start media playback before the fetching process is fully finished. This process relies on meta data stored in the header part of the media file which is being fetched from the server. Progressive download yields the desired effect of streaming, but does not take client device specifications or other run-time factors like dynamic network conditions into account. This is where the newest technology called *HTTP Adaptive Streaming (HAS)* comes into play.

HAS takes device specifications into account when streaming. The main difference with aforementioned streaming solutions is that with HAS the server contains multiple quality representations of the same media. These include different bitrates, resolutions, languages ... When using HAS the client decides, based on device specifications, client characteristics and the currently available bandwidth, which media representation it should download. To make this adaptivity in the media streaming process, the media itself is not represented by one big file, but is split over several short media segments which all contain a piece of the media. HAS takes a different approach when it comes to media fetching. In older protocols like RTSP the server decides which media to send over to the client. With HAS, the client decides which media segments to fetch from the server. This introduces a separation of concerns in the fetching logic which benefits the server. Whereas with RTSP the server had to process each client's needs, this is now done by the client itself.

HAS introduces new needs for streaming servers. The media should be available in different pre-encoded media representations. Such servers need a HAS processing step before they can supply the media (See Figure 2.1). Doing this can be a tedious job which requires some technical know-how about different media bitrates, codecs accepted by different HAS implementations, which devices support which implementation and so on.

This chapter will go into detail about the strengths of HAS (see Section 2.1) and two of the most popular HAS implementations DASH and HLS (see Sections 2.2 and 2.3 respectively) and their inner workings.

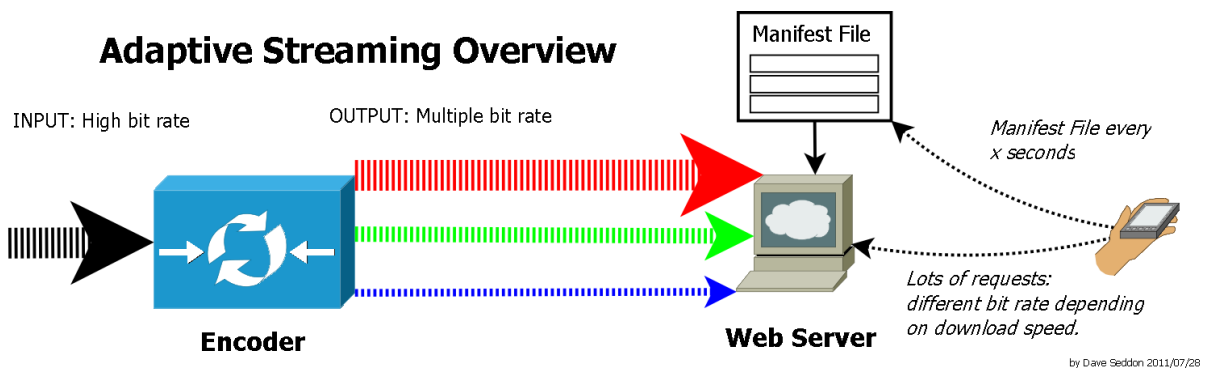


Figure 2.1: Adaptive streaming media preparation overview

2.1 Strengths of Adaptive Streaming over HTTP

The power and popularity of adaptive streaming over HTTP comes from the fact that the World Wide Web is widely available and implemented on almost every “network-oriented” device. Thus making it easy for clients with a web browser to enjoy HAS content.

2.1.1 Device specifications

One of the problems that arises when streaming media, is device specification diversity. Not all devices are equally capable for certain types of media playback. The reason for this being the video codecs used in media encoding. *Codec* is a portmanteau-word for *coding* and *decoding*, referring to algorithms used when compressing and decompressing the media files. Certain codecs use complex algorithms in this process, requiring the device to have a potent processing unit capable of real-time decoding the media being streamed. If a device is not capable of meeting the required processing power, playback of the media will be choppy or not even possible at all. Real-time decoding is usually done by the CPU of a device, which is called software encoding/decoding. There also exist hardware implemented co-processors which off-load the task of encoding/decoding from the CPU. In mobile devices this results in reduced power consumption since these co-processors are more efficient. Hardware (de)coders are usually implemented on an ASIC or FPGA [12]. GPU manufacturers like Nvidia and AMD also supply their hardware with firmware which can off-load encoding/decoding from the CPU [13, 14].

A high-level codec uses complex algorithms for compression that results in media files that do not suffer from quality loss whilst also keeping the file size small. Codecs that use less-complex algorithms can achieve the same quality but require a bigger file as side effect. For streaming, low file sizes are preferred, this yields the best result overall but also requires the device to have sufficient processing power available to decode the compressed media in real-time. At the time of writing, a lot of connected devices are not able to meet these high requirements, thus requiring media files with less complex compression. Many implementations of HAS solve this “device specifications” problem by supplying media files with different encoding settings. If a device has the required processing power, the client can choose to fetch the more complex encoded media file, resulting in less bandwidth usage overall. Devices such as smartphones

which generally do not contain powerful processing units will do the opposite and use more bandwidth to download media files with encoding settings that meet their respective processing power.

2.1.2 Multiple representations

The adaptive nature of HAS does not stop there. HAS also allows media files to have multiple representations. The media for example could be available in multiple resolutions, bitrates, contain multiple audio sources representing different languages. This is a unique feature which is only available in HAS implementations when compared with other streaming solutions like progressive download; which are limited to one representation. This makes it possible for HAS clients to seamlessly up-scale or down-scale the quality as needed, without affecting the smooth playback. By allowing such a big variety in media representations, HAS can supply the media to a big heterogeneous group of devices. Returning a bigger audience and thus a larger reach.

2.1.3 HTTP

Many media consumers nowadays do not realize they are using HAS because it is so simple to use. Every device with network capabilities and a browser can enjoy the fruits that adaptive streaming over HTTP provides. Older implementations use UDP as their transport layer protocol. This introduces issues when a firewall or NAT is in place. Devices behind a NAT or firewall do not have their ports or IP-addresses publicly available at all times for other devices to connect to. In order to use these UDP enabled streaming protocols, techniques like UDP-hole-punching are employed to solve the NAT-traversal issue [15]. In order to set up UDP-hole punching or allow certain UDP ports to be reachable by a streaming server, a user often needs to manually configure his firewall and NAT-devices.

HAS circumvents these problems by using the already implemented and widely used HTTP. Clients typically do not have to configure anything in order to be able to use HAS. HTTP uses the TCP transport layer protocol and all Web servers have to open up the standard ports 80 (http) and 443 (https) [16, 17], therefor additional configuration is not needed. Using HTTP is not the holy grail though, because of the fact that HTTP uses TCP as its transport layer protocol; HTTP inherits control-flow mechanisms like congestion control which can cause delay in media transmission [11].

2.1.4 Live Content versus On-Demand Content

When comparing all streaming implementations over HTTP, HAS is the only one with support for the live-streaming profile. Both *Live* and *On-Demand* profiles are built-in to the core architecture of HAS. Progressive download in contrast, only supports on-demand streaming [18].

	HAS	Progressive download	RTSP
Transport layer protocol	TCP	TCP	UDP
Requires additional transcodes	Yes	No	Possible
Multiple video/audio/subtitle streams support	Yes	Yes	No ¹
NAT Traversal issues	No	No	Yes
Firewall issues	No	No	Yes
Congestion control	Yes	Yes	No ²
Supports live content	Yes	No	Yes

Table 2.1: Summary of streaming implementation properties

¹Possible with the necessary adjustments/additions to the application layer.

²See Footnote 1. Possible to achieve with a custom application-layer implementation, requiring extra effort from the developer.

2.2 Dynamic Adaptive Streaming over HTTP (DASH)

DASH, also known as MPEG-DASH, stands for *Dynamic Adaptive Streaming over HTTP* and is one of the more popular HAS implementations currently available. DASH is being developed by *The Moving Picture Experts Group*, which is a working group from ISO/IEC and has been active since 1988 in the standardization of many video-domain related protocols [19]. Work on DASH began in 2010 and it became a standard in 2012 [20]. Currently many influential companies in the media streaming industry, like Google and Netflix, are using DASH [21]. Many of these companies have gathered to create the DASH-IF, creating guidelines on the usage of MPEG-DASH and also promoting its usage [22]. The ultimate goal of the DASH effort is interoperability and convergence of one open standard instead of the many, still in use today, proprietary implementations.

2.2.1 Inner workings

As mentioned in the introduction of Chapter 2, HAS implementations have a separation of concerns when it comes to streaming and choosing the content that needs to be streamed. This separation of concerns is made possible by what DASH calls a *Media Presentation Description (MPD)* file, this is further down explained in Section 2.2.3. When a client wishes to stream media, the playback device fetches the manifest file from the streaming server. Once the manifest file has been downloaded, the client is able to parse its contents and decide, based on the control heuristics, which segments need to be downloaded. The control heuristics decide which segments are to be fetched based on the information specified in the manifest file and run-time factors like available bandwidth, screen size and so on. If for example the playback device has a slow network connection, it will opt for segments with the lowest bandwidth. The control heuristics will always try their best to at least show something as fast as possible to the client. If during playback the device encounters a change in its environment, for example a less congested network connection, it can alter its pickings from the manifest file accordingly (e.g., upgrade to a better quality representation). Because the client is responsible for the picking logic, the server does not have to support this burden like with older adaptive streaming implementations, but this does induce complexity and the need for processing power at the client its side.

Figure 2.2 depicts the previously explained process in a diagram. The streaming server is represented on the left side, the client logic is represented on the right side. The connection of the two is represented by a HTTP 1.1 connection. As the MPD gets delivered to the client side, the client will decide which actions to take and request segments via the HTTP connection.

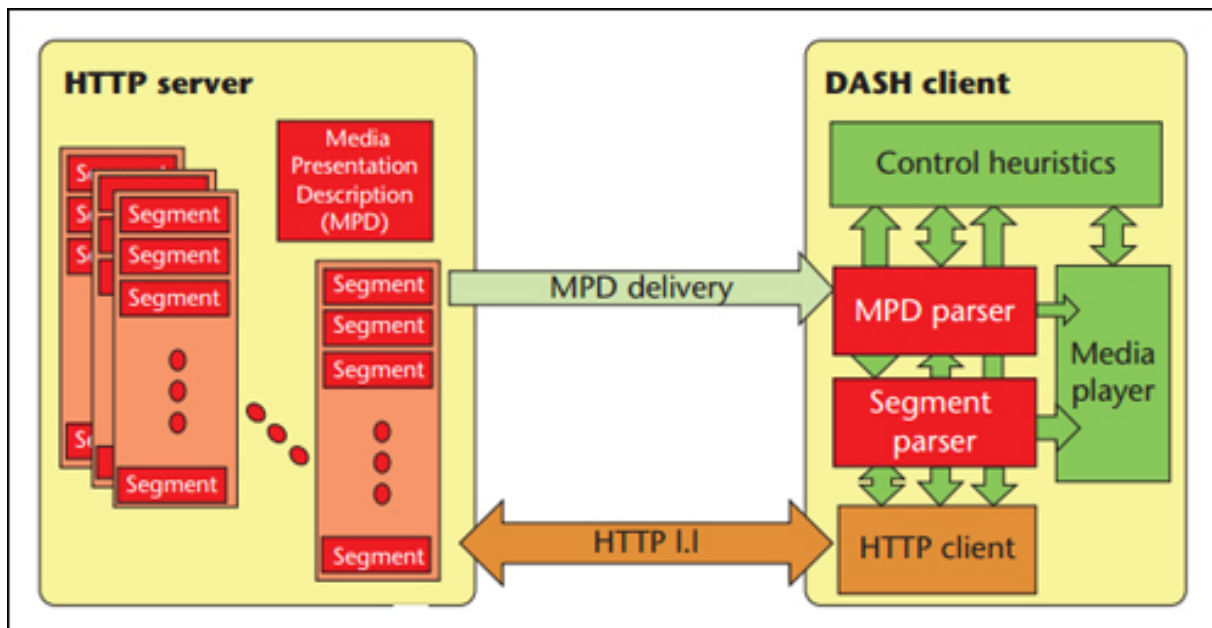


Figure 2.2: DASH Workings (Source: Iraj Sogadar/Microsoft)

2.2.2 Supported codecs

DASH is unique in its kind, in that it is codec agnostic. This implies that DASH does not care what sort of codec is used when encoding the media that it is supposed to stream. Other HAS implementations like Adobe HDS, Microsoft SS ... all push towards the use of H.264 [24, 23]; an MPEG standard for video coding and decoding [25]. Due to the popular choice of the H264 (further explained in Section 3.2.1), DASH-IF also opted for H264 as their main codec of choice [22].

2.2.3 Manifest

The DASH implementation uses XML for formatting its MPD file. Due to the nature of XML, this creates a hierarchical structure. Figure 2.3 depicts this. Every MPD file represents a single streamable piece of media. The media itself can contain multiple streams³. Every MPD file contains at least one *period*, a period represents a portion of the media being streamed, for example a chapter from a movie or a piece of advertisement. Every period contains at least one *adaptation set*, which represents a stream. Typically an MPD contains at least a video and audio adaptation set. Every adaptation set in turn contains at least one *representation*. A representation allows an adaptation set to be represented in multiple formats. A video adaptation set could for example contain 2 representations which have different bitrates or display sizes, allowing the DASH client to pick the one most fit for its environment. A representation contains *segments*, segments are the actual media files themselves. Usually a media representation is split into segments of 2 to 3 seconds in duration. Because of this, a single representation could easily contain over a 1000 segments. These are represented by *segment URLs* in a *segment list* or by a *segment template*. The template solution is the preferred way to go most of the time, since a representation containing a large segment list can result in large MPD files [20]; listings 2.2 and 2.1 show this difference. Listing 2.2 shows an example of a fully configured MPD containing one video adaption set, two audio adaptation sets and two subtitle adaptation sets. The video adaptation set contains two representations using the template system to locate its segments.

The creation of a DASH manifest can be done through various software packets and platforms, this is further explained in Chapter 3 and 1.3 respectively. The playback of a DASH manifest is done by a video player which can interpret the DASH format. For the Web there exists the BitDash player by Bitmovin⁴, the official Dash.js player from DASH-IF⁵, the Shaka Player by Google⁶,... For the desktop there exists a plugin for VLC media player which allows DASH playback [26] as well as libndash⁷, an open source framework which can be used to play DASH content in software.

³A stream represents a data sequence which can be read by programs. In the world of media, a stream usually represents a single piece of video, audio or subtitles. A single stream can contain multiplexed media.

⁴<https://bitmovin.com/html5-player/>

⁵<https://github.com/Dash-Industry-Forum/dash.js>

⁶<https://github.com/google/shaka-player>

⁷<https://github.com/google/ndash>

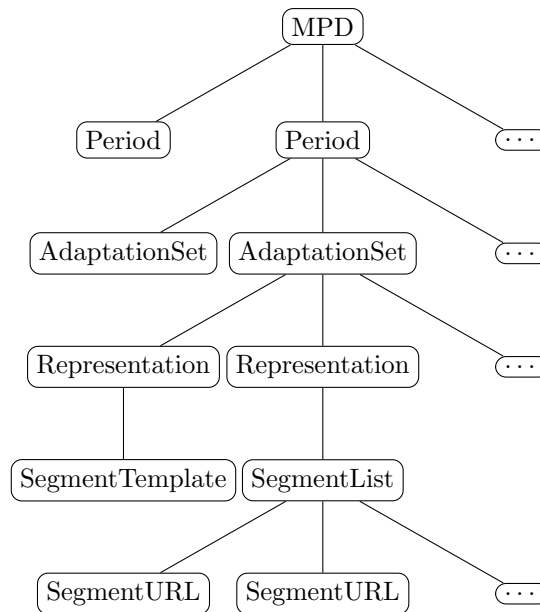


Figure 2.3: DASH MPD file structure

```

1  <!-- A representation with a SegmentList containing SegmentURLs -->
2  <SegmentList>
3    <Initialization sourceURL="video/500kbit/init.mp4"/>
4  </SegmentList>
5  <Representationid="480p 500kbps" mimeType="video/mp4" frameRate="24" bandwidth="520929
   " codecs="avc1.4d4015" width="638" height="272">
6    <SegmentList timescale="1000" duration="2000">
7      <SegmentURL media="video/500kbit/segment_1.m4s"/>
8      <SegmentURL media="video/500kbit/segment_2.m4s"/>
9      <SegmentURL media="video/500kbit/segment_3.m4s"/>
10     <SegmentURL media="video/500kbit/segment_4.m4s"/>
11     <SegmentURL media="video/500kbit/segment_5.m4s"/>
12     <SegmentURL media="video/500kbit/segment_6.m4s"/>
13     <SegmentURL media="video/500kbit/segment_7.m4s"/>
14     <SegmentURL media="video/500kbit/segment_8.m4s"/>
15     <SegmentURL media="video/500kbit/segment_9.m4s"/>
16     <SegmentURL media="video/500kbit/segment_10.m4s"/>
17     <SegmentURL media="video/500kbit/segment_11.m4s"/>
18     <SegmentURL media="video/500kbit/segment_12.m4s"/>
19     <SegmentURL media="video/500kbit/segment_13.m4s"/>
20     <SegmentURL media="video/500kbit/segment_14.m4s"/>
21     <SegmentURL media="video/500kbit/segment_15.m4s"/>
22     <SegmentURL media="video/500kbit/segment_16.m4s"/>
23     <SegmentURL media="video/500kbit/segment_17.m4s"/>
24     <SegmentURL media="video/500kbit/segment_18.m4s"/>
25     <SegmentURL media="video/500kbit/segment_19.m4s"/>
26     <SegmentURL media="video/500kbit/segment_20.m4s"/>
27   </SegmentList>
28 </Representation>
29
30 <!-- A representation with a SegmentTemplate -->
31 <Representation id="480p 500kbps" frameRate="24" bandwidth="520929" codecs="avc1.4
   d4015" width="638" height="272">
32   <SegmentTemplate timescale="1000" duration="2000" media="video/500kbit/segment_$
     Number$.m4s" initialization="video/500kbit/init.mp4" startNumber="1"/>
33 </Representation>

```


Listing 2.1: DASH MPD SegmentUrl vs. SegmentTemplate representations

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT1.500S" type="static"
  mediaPresentationDuration="PT0H0M32.973S" maxSegmentDuration="PT0H0M3.000S"
  profiles="urn:mpeg:dash:profile:isoff-live:2011,http://dashif.org/guidelines/
  dash264">
3   <Period duration="PT0H0M32.973S">
4     <AdaptationSet segmentAlignment="true" maxWidth="1920" maxHeight="1080"
      maxFrameRate="24" par="16:9" lang="eng">
5       <SegmentTemplate timescale="12288" media="$RepresentationID$/segment_${Number$.
        m4s" startNumber="1" duration="36864" initialization="$RepresentationID$/segment_.
        mp4" />
6       <Representation id="1" mimeType="video/mp4" codecs="avc1.42C01F" width="640"
          height="360" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="485958" />
7       <Representation id="2" mimeType="video/mp4" codecs="avc1.42C01F" width="640"
          height="360" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="779332" />
8       <Representation id="3" mimeType="video/mp4" codecs="avc1.42C01F" width="1280"
          height="720" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="1465307" />
9       <Representation id="4" mimeType="video/mp4" codecs="avc1.42C01F" width="1280"
          height="720" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="2374790" />
10      <Representation id="5" mimeType="video/mp4" codecs="avc1.42C01F" width="1920"
          height="1080" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="2956542" />
11      <Representation id="6" mimeType="video/mp4" codecs="avc1.42C01F" width="1920"
          height="1080" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="3945761" />
12      <Representation id="7" mimeType="video/mp4" codecs="avc1.42C01F" width="1920"
          height="1080" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="5898235" />
13    </AdaptationSet>
14    <AdaptationSet segmentAlignment="true" lang="eng">
15      <SegmentTemplate timescale="48000" media="$RepresentationID$/segment_${Number$.
        m4s" startNumber="1" duration="144000" initialization="$RepresentationID$/segment_.
        mp4" />
16      <Representation id="8" mimeType="audio/mp4" codecs="mp4a.40.2" audioSamplingRate
        ="48000" startWithSAP="1" bandwidth="60067">
17        <AudioChannelConfiguration schemeIdUri="
        urn:mpeg:dash:23003:3:audio_channel_configuration:2011" value="2" />
18      </Representation>
19      <Representation id="9" mimeType="audio/mp4" codecs="mp4a.40.2" audioSamplingRate
        ="48000" startWithSAP="1" bandwidth="39891">
20        <AudioChannelConfiguration schemeIdUri="
        urn:mpeg:dash:23003:3:audio_channel_configuration:2011" value="2" />
21      </Representation>
22    </AdaptationSet>
23    <AdaptationSet segmentAlignment="true" lang="eng">
24      <SegmentTemplate timescale="1000" media="$RepresentationID$/segment_${Number$.m4s
        " startNumber="1" duration="3000" initialization="$RepresentationID$/segment_.mp4"
        />
25      <Representation id="10" mimeType="application/mp4" codecs="wvtt" startWithSAP="1
        " bandwidth="1322" />
26    </AdaptationSet>
27  </Period>
28 </MPD>

```

Listing 2.2: DASH MPD example

2.3 HTTP Live Streaming (HLS)

HLS is a HAS implementation being developed by Apple [27]. It contains many similarities to the previously explained DASH (see Section 2.2). HLS has been in development since 2009 and has currently been deployed to all Apple devices still receiving updates [28]. The main difference with DASH is that HLS is not a global standard but a proprietary Apple specification for media streaming. All developers are required to follow the strict rules set by Apple before they can deploy their software on Apple hardware [29]. As a result of this, Apple has created a closed ecosystem where HLS is the primary HAS implementation being used in all Apple software and hardware. Many other organizations like Google, Mozilla, Microsoft... have accepted this and are also deploying HLS support in their software/browsers to keep a competing position with Apple software.

2.3.1 Manifest

HLS structures its manifest in UTF-8 plain text files containing only URIs for its media locations and descriptive tags. A HLS manifest, also called a playlist, is identified with the *.m3u8* or *.m3u* extension and the mime-type *application/vnd.apple.mpegurl* or *audio/mpegurl*. A playlist can either be a master playlist or a media playlist. A master playlist only contains URIs to media playlists, whilst a media playlist only contains URIs to actual media resources like video, audio, subtitles ... [28]/

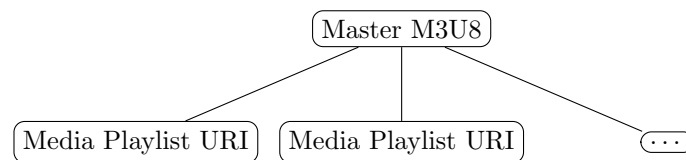


Figure 2.4: HLS M3U8 Master Playlist Structure

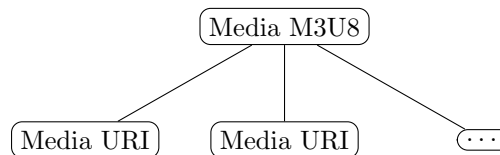


Figure 2.5: HLS M3U8 Playlist Structure

```

1 #EXTM3U
2 #EXT-X-TARGETDURATION:10
3 #EXT-X-VERSION:3
4 #EXTINF:9.009,
5 http://media.example.com/first.ts
6 #EXTINF:9.009,
7 http://media.example.com/second.ts
8 #EXTINF:3.003,
9 http://media.example.com/third.ts
10 #EXT-X-ENDLIST
  
```

Listing 2.3: HLS Media Playlist

```

1 #EXTM3U
2 #EXT-X-STREAM-INF:BANDWIDTH=1280000,AVERAGE-BANDWIDTH=1000000
3 http://example.com/low.m3u8
4 #EXT-X-STREAM-INF:BANDWIDTH=2560000,AVERAGE-BANDWIDTH=2000000
5 http://example.com/mid.m3u8
6 #EXT-X-STREAM-INF:BANDWIDTH=7680000,AVERAGE-BANDWIDTH=6000000
7 http://example.com/hi.m3u8
8 #EXT-X-STREAM-INF:BANDWIDTH=65000,CODECS="mp4a.40.5"
  
```

9 <http://example.com/audio-only.m3u8>

Listing 2.4: HLS Master Playlist

2.4 Other Adaptive Streaming Implementations

Even though DASH has been the standard for adaptive streaming since 2012 and Apple clings onto HLS, other adaptive streaming implementations have been created and are still used to this day. This section will describe two other adaptive streaming implementations invented by Microsoft and Adobe.

2.4.1 Microsoft Smooth Streaming

Microsoft Smooth Streaming is an adaptive streaming implementation based on progressive download [30]. It provides adaptive streaming based on the MP4 container format. Silverlight⁸ is the preferred client for using MSS as it comes with default adaptive streaming heuristics [31]. MSS is not Silverlight exclusive though, any client that capable of decoding H264 and/or VC-1 encoded video streams, AAC and/or WMA encoded audio streams and HTTP is able to implement MSS [30]. The Microsoft Server operating system can support adaptive streaming by enabling the IIS media services Smooth Streaming extension [31].

The way MSS works is by providing the client with a manifest that specifies different quality URL's for the media being streamed. The client then decides, based upon the current environment (same logic as DASH applies here, see Section 2.2.1 for more information) which quality it wants to download. The content itself is delivered in chunks as specified by the progressive download protocol [32]. The different qualities are stored as one media file at server side which is not necessarily the case with DASH.

2.4.2 Adobe HTTP Dynamic Streaming

HTTP Dynamic Streaming is an adaptive streaming implementation by Adobe. It is built natively into Adobe Flash Player⁹ and Adobe AIR¹⁰ [33]. Files are encoded into the *F4F* container format, which is based upon the MP4 container format [24, 34]. Only H264 or VP6 encoded video streams and MP3 or AAC encoded audio streams are possible.

Adobe HDS works exactly like MSS in that it provides a manifest to the client which specifies the different qualities that are available for the media being streamed [35].

⁸<https://www.microsoft.com/silverlight/>

⁹<https://get.adobe.com/nl/flashplayer/>

¹⁰<https://get.adobe.com/air/>

Chapter 3

Used Tools and Codecs

The world of media is one that has been in development for centuries. There are evidences that the concept of the camera-obscura has been known to men since the Palaeolithic era. Mankind has always tried to capture moments; be it with the older analog technology or the newer more used digital variant, sometimes called *new media*. This new media, stored in digital format, is easy to copy, modify, share, ... which contributes to its popularity.

Digital media is stored in so called *container formats*, sometimes also called *wrappers*. A container format defines the overall structure of a file, including how the file's video, audio, metadata and index information are multiplexed together [36]. A container format however does not define how the video and/or audio is encoded. Container formats are usually indicated by giving a media file a certain extension. For example, video can be identified by one of the following (but not limited to): AVI, MP4, WEBM, FLV, MOV, ... Every container format has its own traits and supported codecs which are all specified in its metadata such that a playback device can properly decode the content. Simpler container formats are exclusive to one audio or video stream (e.g., WAV is a popular Windows audio only container format). More advanced container formats allow multiple video and audio streams to be contained in a single file (e.g., MKV is an open standard container format intended to serve as a universal format for storing multimedia [37]). The choice of a container depends on the needs of the media creator and/or (the limitation of) the target audience.

This chapter will explain the different tools and media codecs used during the development of Bendwit (see Chapter 4 for more details). Section 3.1 will explain the different tools and why they were chosen. Section 3.2 will give a brief introduction to codecs and the codecs that Bendwit supports at the time of writing.

3.1 Tools

We can easily see that the digital media world is a broad one with many tools, codecs, container formats, ... With the scope of this thesis being Adaptive Streaming, only a few tools were fit for preparing content for adaptive streaming. To narrow it down: we need tools that can be operated through self-written code, either via an API or by passing command line arguments. Sections 3.1.1, 3.1.2 and 3.1.3 describe the chosen tools and why they were picked for this thesis.

3.1.1 FFmpeg

FFmpeg is an open source project that creates pieces of software for media handling. The name *FFmpeg* itself is used for one of the main pieces of software and the name of the project itself. The name is inspired by MPEG with “ff” standing for “fast forward” [38].

One of the requirements for this thesis was a tool which could perform the transcoding¹ of media files. This immediately brought up the popular choices: *FFmpeg* from FFmpeg and *avconv* from Libav. FFmpeg has been in development since 2000 [39]. Libav on the other hand is a fork that took place in 2010 because of internal team struggles [40]. Both projects share the same goal and differ only slightly. At the time of writing, FFmpeg is more stable when it comes to bugfixes[41], making it the preferred choice for this thesis.

Transcoding Process

The FFmpeg command line tool currently supports over a hundred codecs [42], making it easy to transcode media from various sources to the ones our thesis implementation will support (further explained in Chapter 4). FFmpeg has a powerful transcoding process, as depicted in Figure 3.1.

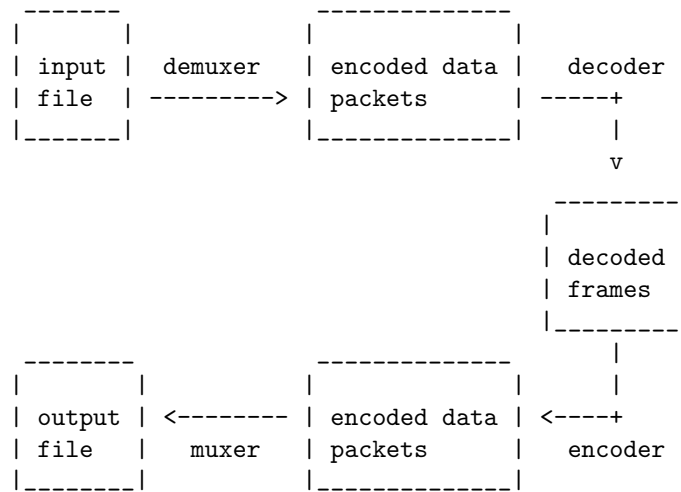


Figure 3.1: FFmpeg transcoding process (Source: FFmpeg documentation)

FFmpeg comes with a plethora of parameters for encoding media [43], the tool works by supplying parameters via the command line interface. FFmpeg can read from an arbitrary amount of input files and write to an arbitrary amount of output files². It handles audio and video conversion and manipulation in a fast manner. Individual stream options like codec choice and codec specific options (see Section 3.2 for more information) are also possible. Depending on the container format used, stream selection is either done automatically or through the more advanced manual `-map` parameter. FFmpeg chooses video streams automatically based on the highest resolution, audio streams based on the most channels and subtitles on the first stream it encounters [43].

FFmpeg will serve the purpose as a transcoding tool in the implementation of this thesis. In order to prepare media for adaptive streaming, the input file³ needs its streams split and transcoded into individual files of different qualities.

Adaptive Streaming Support

The way FFmpeg works, is by incorporating many external libraries and codecs and making them available from the confines of a single tool. At the time of writing, FFmpeg only includes the Google Webmproject tools[44] to create DASH content in WebM container format with the VP9 codec. Support for HLS content is built-in natively. Since the focus of this thesis is on DASH with codecs other than

¹The conversion of one encoding format to another.

²A file can be supplied as a pipe, network stream, local file, ...

³Bendwit will only work with one input file at a time. It will not make use of the possibility provided by FFmpeg to supply an arbitrary amount of input files.

VP9 and HLS content, another tool (explained in Section 3.1.3) will be required for DASH content preparation.

3.1.2 FFprobe

As explained in Section 3.1.1, the FFmpeg project bundles a lot of tools to make command line handling of media possible. One of these tools is FFprobe [45]. As the name suggests, the tool allows probing after metadata in media files. The metadata retrieved, depends on the container format and codec type being used. General metadata which always gets extracted includes (but is not limited to): stream codec, video height, video width, video framerate, audio channel layout, subtitle language, ...

FFprobe allows to specify a “writer” format through the `-print_format` parameter, which is essentially the format in which the metadata should be returned. At the time of writing FFprobe supports the following formats: default (human readable), csv, json, xml, ini and flatfile [46]. By printing out the probed metadata in a data format like json or xml, other programs are able to interpret it, and thus acquire basic information about the media file to work with. This makes FFprobe an interesting tool which will be used for basic information extraction in the thesis implementation. Listing 3.1 shows the probe results of the Sintel open movie media file project⁴ in the default writer format. It is even possible to probe the individual streams further by applying the `-show_streams` parameter to the FFprobe tool. Listing 3.2 show an extract of stream 0, which represents the video stream.

```

1 Input #0, matroska,webm, from 'Sintel.2010.1080p.mkv':
2   Metadata:
3     encoder           : libebml v1.0.0 + libmatroska v1.0.0
4     creation_time      : 2011-04-25T12:57:46.000000Z
5   Duration: 00:14:48.03, start: 0.000000, bitrate: 10562 kb/s
6   Chapter #0:0: start 0.000000, end 103.125000
7   Metadata:
8     title             : Chapter 01
9   Chapter #0:1: start 103.125000, end 148.667000
10  Metadata:
11    title              : Chapter 02
12  Chapter #0:2: start 148.667000, end 349.792000
13  Metadata:
14    title              : Chapter 03
15  Chapter #0:3: start 349.792000, end 437.208000
16  Metadata:
17    title              : Chapter 04
18  Chapter #0:4: start 437.208000, end 472.075000
19  Metadata:
20    title              : Chapter 05
21  Chapter #0:5: start 472.075000, end 678.833000
22  Metadata:
23    title              : Chapter 06
24  Chapter #0:6: start 678.833000, end 744.083000
25  Metadata:
26    title              : Chapter 07
27  Chapter #0:7: start 744.083000, end 888.032000
28  Metadata:
29    title              : Chapter 08
30  Stream #0:0(eng): Video: h264 (High), yuv420p(tv, bt709/unknown/unknown,
    progressive), 1920x818, SAR 1:1 DAR 960:409, 24 fps, 24 tbr, 1k tbn, 48 tbc
31  Stream #0:1(eng): Audio: ac3, 48000 Hz, 5.1(side), fltp, 640 kb/s
32  Metadata:
33    title              : AC3 5.1 @ 640 Kbps
34  Stream #0:2(ger): Subtitle: subrip
35  Stream #0:3(eng): Subtitle: subrip

```

⁴<https://durian.blender.org/>

```

36     Stream #0:4(spa): Subtitle: subrip
37     Stream #0:5(fre): Subtitle: subrip
38     Stream #0:6(ita): Subtitle: subrip
39     Stream #0:7(dut): Subtitle: subrip
40     Stream #0:8(pol): Subtitle: subrip
41     Stream #0:9(por): Subtitle: subrip
42     Stream #0:10(rus): Subtitle: subrip
43     Stream #0:11(vie): Subtitle: subrip

```

Listing 3.1: Probe output of the Sintel open movie project (durian.blender.org)

```

1  [STREAM]
2  index=0
3  codec_name=h264
4  codec_long_name=H.264 / AVC / MPEG-4 AVC / MPEG-4 part 10
5  profile=High
6  codec_type=video
7  codec_time_base=1/48
8  codec_tag_string=[0][0][0][0]
9  codec_tag=0x0000
10 width=1920
11 height=818
12 coded_width=1920
13 coded_height=818
14 has_b_frames=2
15 sample_aspect_ratio=1:1
16 display_aspect_ratio=960:409
17 pix_fmt=yuv420p
18 level=41
19 color_range=tv
20 color_space=bt709
21 color_transfer=unknown
22 color_primaries=unknown
23 chroma_location=left
24 field_order=progressive
25 timecode=N/A
26 refs=1
27 is_avc=true
28 nal_length_size=4
29 id=N/A
30 r_frame_rate=24/1
31 avg_frame_rate=24/1
32 time_base=1/1000
33 start_pts=0
34 start_time=0.000000
35 duration_ts=N/A
36 duration=N/A
37 bit_rate=N/A
38 max_bit_rate=N/A
39 bits_per_raw_sample=8
40 nb_frames=N/A
41 nb_read_frames=N/A
42 nb_read_packets=N/A
43 DISPOSITION:default=0
44 DISPOSITION:dub=0
45 DISPOSITION:original=0
46 DISPOSITION:comment=0
47 DISPOSITION:lyrics=0
48 DISPOSITION:karaoke=0

```

```

49 DISPOSITION:forced=0
50 DISPOSITION:hearing_impaired=0
51 DISPOSITION:visual_impaired=0
52 DISPOSITION:clean_effects=0
53 DISPOSITION:attached_pic=0
54 DISPOSITION:timed_thumbnails=0
55 TAG:language=eng
56 [/STREAM]

```

Listing 3.2: Probe output of the Sintel open movie project (durian.blender.org) with the `-show_streams` parameter set

3.1.3 MP4Box

GPAC is an open source software project which created the tool MP4Box: a multimedia packager with the capability of preparing content for adaptive streaming over HTTP [47, 48]. MP4Box was one of the first tools to support DASH manifest creation and media file segmentation, even before it was considered a standard [49]. Because of this, MP4Box offers a stable solution for DASH content creation which will be used in the thesis implementation.

3.2 Codecs

Every video and audio content stream inside a container format contains information which has been encoded with its specific *codec*, as briefly explained in Section 2.1.1. A codec can *encode* a piece of media into a specific stream format and later *decode* it again using compression and decompression algorithms. There exist many different codecs, some of which are more popular than others. The thesis implementation explained in Chapter 4 uses some of the more popular codecs in order to reach a wide target audience of playback devices capable of decoding the codecs explained further in Sections 3.2.1, 3.2.2 and 3.2.3.

3.2.1 H264

MPEG-4 Part 10, Advanced Video Coding (MPEG-4 AVC), more commonly known as *H264*, is a development effort by the MPEG and CPEG gathered under the joint name JCT-VC. H264 is a proprietary codec protected by a multitude of patents from different parties. The patents are grouped together into a license governed by MPEG-LA⁵ [50]. Royalty payment is required when H264 technology is used for commercial purposes. MPEG-LA has allowed free streaming of H264 encoded media if and only if the media being streamed is free for the end users [51].

As of 2014, Apple announced their full support for H264 and began using it as their primary video codec in all their software (Quicktime, iTunes, DVDs, ...) [52]. Because of Apple's massive success during the years to follow [53], H264 would gain more and more attention. It would even be such a thorn in the eye of bigger companies like Google that they would create an open source alternative called VP8 (see Section 3.2.2 for more information). As of 2017, Apple and others are moving to the successor of H264: *HEVC* also known as *H265*. However most streamable video content as of 2017 is still available in H264 encoded format, making it one of the more popularly used codecs.

Profiles and Levels

H264 defines a set of capabilities and constraints, called *Profiles* and *Levels* respectively.

A *profile* tells a playback device exactly what is needed in order to be able to decode a video stream. Many profiles exist, but the most important ones are `baseline`, `main` and `high` profiles. Baseline

⁵The MPEG-LA license does not include all patents technologies used within H264.

targets the more low-end playback devices or devices that require a more robust video stream due to poor Internet connection. Main was created for standard-digital TV broadcast quality video. High profiles on the other hand are created for high definition content [54]. Depending on the profile the playback device supports and the profile used to encode a video stream, H264 offers a multitude of different features. The baseline and main profiles for example only support up to 8bit color depth, high profiles on the other hand support up to 14bit color depth. The complexity of a profile defines how efficiently the compression takes place, this however also means more computing overhead in order to encode and decode the video stream.

A *level* imposes a constraint on top of a profile. It specifies the required decoding performance of a decoder, as it is expressed in terms of the maximum frame rate, resolution, bit rate, ... the decoder can handle [54].

Inter frame technology

A video stream consists of picture frames put in a sequence after each other. In order to compress a video stream as much as possible, some frames are expressed in terms of their neighboring frames when using inter frame coding technology. The amount of picture frames shown within a time periode is called the framerate, sometimes also referred to as the amount of *frames per second*. If for example a video stream uses a framerate of 60, the viewer will be shown 60 picture frames a second. Most frames will contain redundant information because of the high amount of frames shown within a one second period, which is exactly what inter framing utilises. By dividing a frame into smaller parts called *macroblocks*, a *block matching* algorithm can be used to try to estimate similar blocks in another frame. This information is encoded within a *motion vector* which describes the transformation required from one 2D image to another, or in the case of a video stream: the transformation of the current frame to the reference frame. This process is called *motion estimation*.

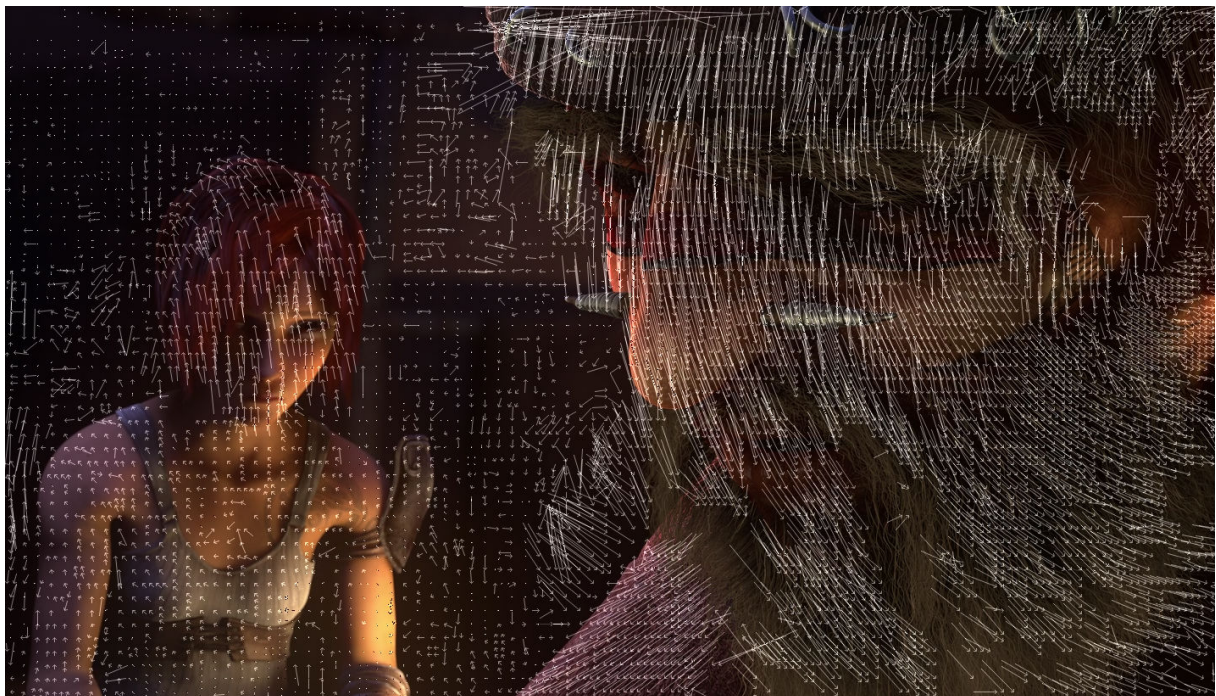


Figure 3.2: A forward motion prediction visualisation of the Sintel, Durian open movie project. An upward movement of the character on the left and a downward movement of the character on the right can be seen.

H264 makes use of inter framing in order to compress the file size of a video as much as possible within compliance of the specified profile. The codec makes use of three types of frames: *I frames*, *P frames* and *B frames*. I frames are self contained picture frames which consist of the full picture, no prediction or references to other frames are used in order to construct this frame, they are sometimes also referred to as *key frames*. P frames are predictive frames and are based on earlier pictures from I or P frames [55].

B frames on the other hand are bidirectional predictive frames which make use of earlier and/or later frames [56], making it the most compressed frame size. Together these frames are collected in a sequence called a *group of pictures*. Motion estimation only happens within a GOP. The amount of frames within a GOP is called the GOP size. A video stream itself consists of a multitude of GOPs. Only the main profiles and above make use of B frames, the performance needed to execute the prediction algorithms used within B frames cannot be met by low-end playback devices [54].

3.2.2 VP8

VP8 is an open source and royalty free codec owned by Google⁶, developed by On2 Technologies as a successor to VP7. On August 5 of 2009, Alphabet (Google's mother company) acquired On2 technologies. As of 2010, Google has declared the codec to be royalty free and has released the specification under the Creative Commons Attribution 3.0 license [57].

VP8 is very similar to the way H264 works, it was introduced and made royalty free in order to challenge the non-free H264 codec [58]. Just as H264, VP8 makes use of inter framing I and P frames to compress the file size, yet it does not utilise B frames. It was found that it is very rare for more than three reference frames to provide significant quality benefit [59]. Instead, VP8 makes use of a *golden reference frame* and/or an *alternate reference frame* [60]. The idea behind the GRF and ARF is to store a frame from an arbitrary point in the past to be used for inter frame predictions [60, 61], similar to the way B frames make use of past (or future) frames that are not adjacent to the current frame in the frame sequence.

When an H264 encoded and VP8 encoded video stream with the same target bitrate and aspect ratios were compared directly, the difference seen by the naked eye was so small it can be considered trivial [62].

3.2.3 AAC

Advanced Audio Coding is a proprietary audio codec designed in 1997. No license or payment is required in order to distribute content in AAC format. AAC codec developers on the other hand are required to pay a patent license [63]. The goal of the codec was to become the successor of MP3 and achieve a better quality at the same bit rate [64]. The AAC codec has been standardized by ISO and IEC in 2006 [65]. At the time of writing, AAC is the most widely supported audio codec and is being used as a standard in many devices and streaming purposes (e.g., Youtube videos [66]).

⁶<http://www.webmproject.org/>

Chapter 4

Bendwit

Bendwit¹ is a service platform that allows users to prepare media content for adaptive streaming. Access is provided through a *web platform*, a *REST API* and a *programming API*. The platform is built in such a way that users with minimal knowledge (henceforth denoted with the term novice users) are able to produce content for adaptive streaming via a minimalistic and easy to use interface. At the same time, Bendwit allows for a large degree of freedom by providing more advanced users (henceforth denoted with the term expert users) with the possibility of fine tuning how media content should be represented (resolution, codec used, bit rate, framerate, ...).

Section 4.1 will summarize Bendwit its high-level system design and how communication takes place between the different logical components it encompasses. Sections 4.2, 4.3, 4.4, 4.5, 4.6 will then describe the different building blocks of Bendwit in detail.

4.1 Bendwit workflow

Instead of building a monolithic system that contains all logic, API's and frontend matters, it was decided early on that using a layered design would result in a more flexible system that could serve more than one purpose. Each layer is exposed through a well-designed interface which allows for separation of concerns, facilitates interoperability and allows for future changes to the code itself. Figure 4.1 visualises the layered architecture of Bendwit. The core functionality of Bendwit resides within the CLI tool (depicted at the very bottom in the backend layer). Users make requests via the frontend layer which communicates with the backend layer. Novice users are able to issue requests via the website interface. Expert users on the other hand also have the choice to communicate with the Bendwit platform through the *REST API* directly or an abstraction of it called the *developer API*. In order to control the flow of requests in the backend, a job scheduler handles all incoming requests and queues them for processing.

Figure 4.2 depicts the communication onion model of Bendwit. User requests propagate from the outer layer inwards to the center and communication only takes place between subsequent layers.

¹The name Bendwit is a phonetic twist on the word “bandwidth”. It consists of the verb “to bend” and the noun “wit”. This all refers to the smart way HAS players handle the bandwidth that is available to them, thus bending the media in such a flexible way that ensures a continuous playback.

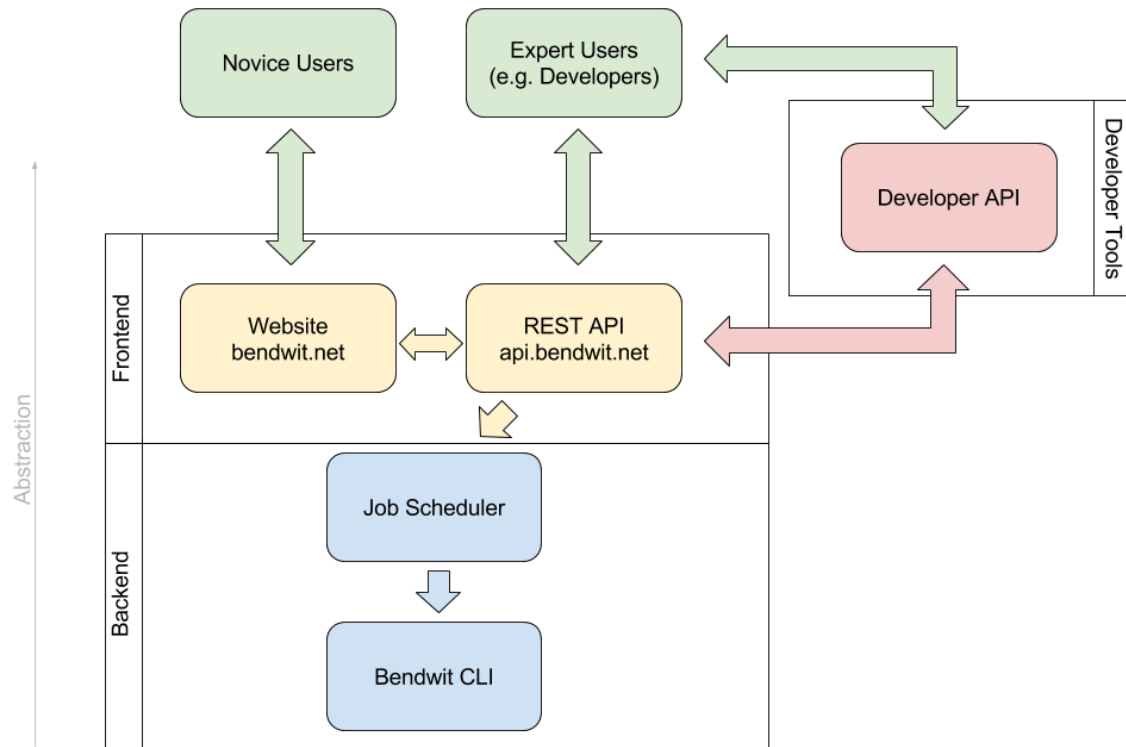


Figure 4.1: Bendwit multi layered workflow

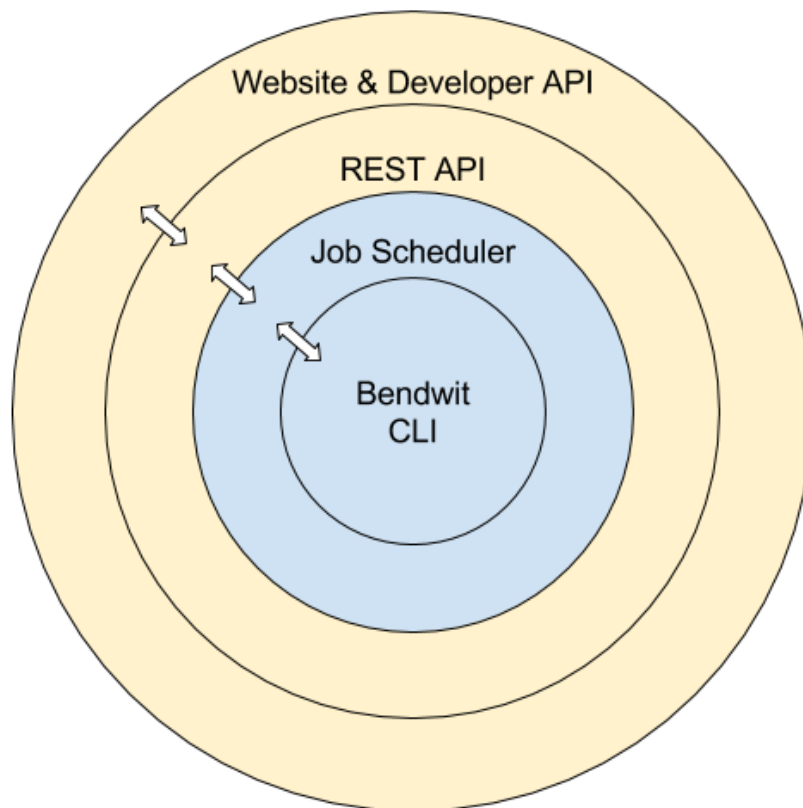


Figure 4.2: Bendwit onion model of the Bendwit communication layers

4.2 Command Line Tool

The command line interface tool is the core of the Bendwit platform, encapsulating logic pertaining to the preparation of media content for adaptive streaming. It handles media transcoding, media probing and HAS media preparation. As a result of the multi layered software approach, the CLI tool was developed as a standalone tool for Linux based operating systems. If so desired, it can hence be used by expert users directly, this way bypassing the interfacing sugar that is added by higher layers in the Bendwit software architecture.

4.2.1 Choice of Programming Language

Bendwit CLI was designed in the programming language Python version 3². Not only does Python offer the object-oriented paradigm needed for the software architecture (explained in Section 4.2.2), it also comes with a wide variety of prebuilt libraries like:

- `json` Used for parsing user configuration files and outputting probe data
- `subprocess` Used for spawning and operating the tools described in Chapter 3
- `logging` Built-in logging system

Python also simplifies avoiding bad coding practices like large switch/if structures by providing developers with the functional and lambda programming paradigms. These allow flexible systems which do not require maintenance when new features are added (see Section 4.2.2 “Parameter Manager” for more information).

4.2.2 Inner workings

The purpose of the tool is to prepare media for adaptive streaming. This can be described on a high level as follows:

1. Take a media file as input
2. Transcode this input to different representations³
3. Splice all the transcoded files in media segments of x seconds⁴
4. Create an adaptive streaming manifest file holding the necessary streaming metadata about the generated media segments

This process is possible with the use of the tools described in Chapter 3. Transcoding is done with FFmpeg, DASH content is produced with MP4Box and HLS content is also produced with FFmpeg. Probing the input media file for information is done with FFprobe.

The CLI tool is made in such a way that it prepares exactly one media artifact for adaptive streaming each time it is run. A user provides a configuration file for the run at initialisation that contains all the necessary information (see Section 4.2.3 for more details). Because multiple adaptations of the input media are allowed, the tool also provides an error checking system that outputs possible warnings that could occur: upscaling the resolution which causes artifacts like blur, the use of multiple aspect ratios which will cause the instantiation of multiple DASH adaptation sets, conflicting settings, ... As a result, a software architecture was needed that could provide such feedback information based on the input media probe and the user specified configuration (see “Software architecture” for more details).

²<https://www.python.org/>

³Representations are alternative versions of the media input artifact. For example a different bitrate, different resolution,

...

⁴DASH and HLS typically use segments with fixed durations between 1 and 10 seconds of duration

4.2.3 Configuration File

At initialisation a JSON formatted configuration file is specified with the `--config <filename>` option. In this file the user must specify the required `source_filename` and `output_type` keys. At the time of writing the tool supports 3 output types: `dash`, `hls` and `probe`. Without specifying the `output_directory` key, the output will be produced in the directory the tool is run. When `dash` or `hls` are selected as output type, the tool also requires the `intermediate_files` key to be supplied. This key contains a dictionary of codec types and a list of codec options for each desired media representation (i.e., intermediate file). Listing 4.1 shows a DASH preparation example, Listing 4.2 shows a probe request example. The possible keys for `intermediate_files` are identified by the CLI tool its present configuration classes (see “Software Architecture”, “Parameter Manager” and Section 4.2.4 for more information)

```

1 {
2   "source_filename" : "input_file.mkv",
3   "output_type" : "dash",
4   "output_directory" : "demo/",
5   "segment_length" : 3000,
6   "vacuum" : true,
7   "intermediate_files" : {
8     "h264" : [
9       {
10        "media_height" : 360,
11        "framerate" : 24,
12        "kilobitrate" : 500,
13        "h264_profile" : "baseline",
14        "level" : 3.1,
15        "source_index" : 0
16      },
17      {
18        "media_height" : 360,
19        "framerate" : 24,
20        "kilobitrate" : 800,
21        "h264_profile" : "baseline",
22        "level" : 3.1,
23        "source_index" : 0
24      },
25      {
26        "media_height" : 720,
27        "framerate" : 24,
28        "kilobitrate" : 1500,
29        "h264_profile" : "baseline",
30        "level" : 3.1,
31        "source_index" : 0
32      },
33      {
34        "media_height" : 720,
35        "framerate" : 24,
36        "kilobitrate" : 2400,
37        "h264_profile" : "baseline",
38        "level" : 3.1,
39        "source_index" : 0
40      },
41      {
42        "media_height" : 1080,
43        "framerate" : 24,
44        "kilobitrate" : 3000,
45        "h264_profile" : "baseline",

```

```

46     "level" : 3.1,
47     "source_index" : 0
48 },
49 {
50     "media_height" : 1080,
51     "framerate" : 24,
52     "kilobitrate" : 4000,
53     "h264_profile" : "baseline",
54     "level" : 3.1,
55     "source_index" : 0
56 },
57 {
58     "media_height" : 1080,
59     "framerate" : 24,
60     "kilobitrate" : 6000,
61     "h264_profile" : "baseline",
62     "level" : 3.1,
63     "source_index" : 0
64 }
65 ],
66
67 "aac" : [
68     {
69         "kilobitrate" : 56,
70         "channel_layout" : "stereo",
71         "source_index" : 1
72     },
73     {
74         "kilobitrate" : 36,
75         "channel_layout" : "stereo",
76         "source_index" : 1
77     }
78 ]
79 }
80 }

```

Listing 4.1: DASH with H264 and AAC codec configuration file example

```

1 {
2     "source_filename" : "input_file.mkv",
3     "output_type" : "probe",
4     "output_directory" : "demo/"
5 }

```

Listing 4.2: Probe configuration file example

Software Architecture

The ability to supply the user with a plethora of configuration options ensues from the modular architecture used by the CLI tool. Every media file consists of “streams”, which can be of the type video, audio or subtitles. Every stream is encoded with a certain codec. By creating information structures for every stream, we can compare them in order to discover conflicts (see “Conflict Management” for more information) and create encode commands for FFmpeg.

During development an object-oriented approach was used to design a class hierarchy that represents these configuration information structures. Figure 4.3 shows the class diagram.

There are in total three basic configuration types, which represent the three possible stream types: `BasicVideoConfig`, `BasicAudioConfig` and `BasicSubtitleConfig`.

The basic configuration classes serve the purpose of being expanded into more specific classes which represent codecs: `H264VideoConfig`, `Vp8VideoConfig`, `AacAudioConfig`, `VttSubtitleConfig`, ... the basic configuration classes are also used as **reference** classes for conflict management (see “Conflict Management” for more information) .

Internally the configuration objects belong to either of the two following categories: *input configuration objects* or *output configuration objects*. When a configuration object has the **reference** parameter set, it belongs to the *output configuration objects*. Output configuration objects are created from what the user specifies in the **intermediate_files** key inside the configuration file (see “Parameter Manager” for more information). Without the **reference** parameter set, the object serves the purpose of an *input configuration object* (see “Conflict Management” for more information). Depending on the type the configuration object belongs to, it will have following responsibilities:

	Input configuration objects	Output configuration objects
Manage stream type (and codec) specific data	✓	✓
Produce correct FFmpeg encode commands with the given parameters	✗	✓
Find possible conflicts between reference object and its internal parameters	✗	✓

Encode command creation

Creating the FFmpeg encode command is done with the `get_encode_command()` method that every configuration class contains. The output configuration object will create an FFmpeg valid command string for the codec type it represents. For the first entry of the H264 example given in Listing 4.1, the configuration class would produce following the FFmpeg encoding commands (See Section 4.2.4 for more information):

```

1 ffmpeg -y -i input_file.mkv -c:v libx264 -vf scale=w=-2:h=360:
  force_original_aspect_ratio=decrease -pix_fmt yuv420p -r 24 -profile:v baseline -
  x264opts keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -
  map_chapters -1 -pass 1 -passlogfile demo/pass_log_video_transcode_0.mp4 -b:v 500k
  -maxrate 500k -bufsize 1000k -f mp4 /dev/null
2
3 ffmpeg -y -i input_file.mkv -c:v libx264 -vf scale=w=-2:h=360:
  force_original_aspect_ratio=decrease -pix_fmt yuv420p -r 24 -profile:v baseline -
  x264opts keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -
  map_chapters -1 -pass 2 -passlogfile demo/pass_log_video_transcode_0.mp4 -b:v 500k
  -maxrate 500k -bufsize 1000k demo/video_transcode_0.mp4

```

The data used inside the encode command originates from the parameters supplied to the configuration object at creation. Every configuration object has a serializer method `_get_debug_string()`, which can be used internally for debugging purposes. The first entry of the H264 example given in Listing 4.1 would return the following string:

```

1 H264 Video configuration (Encoding options present) | Index 0 | Width Nonepx | Height
  360px | Frame rate 24fps | Numerical frame rate 24.0fps | Bit rate 500kbps | H264
  Profile baseline | H264 Level None | Display ratio None | Codec name 'libx264' |
  Filename 'FilenameNotSet.mp4'}

```

Conflict Management

Configuration objects have the option to supply a **reference** parameter. The value of that parameter is an input configuration object. By supplying a **reference** object, the output configuration object has the ability to check its own internal data against the reference, this is called *intra-conflict checking*.

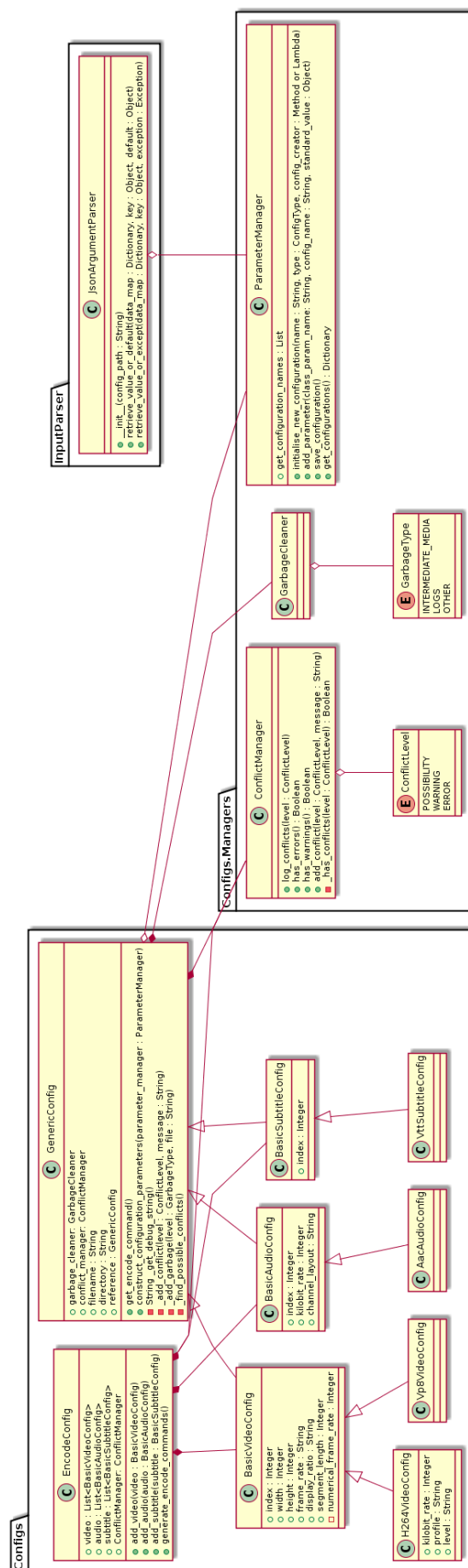


Figure 4.3: Class diagram of the configuration class architecture

This is a useful feature that enables checking encode parameters specified by the user against the source artifact. Next to intra checking, the system also provides a feature called *inter-conflict checking*. All output configuration objects are gathered in an `EncodeConfig` object. The purpose of this class is to globally check for conflicts between output configuration objects.

Encoding requires a lot of processing power. By providing the user with an early feedback mechanism concerning possible mistakes, time and processing cycles are saved. At initialisation the CLI tool probes the source artifact with FFprobe. All source artifact streams get dissected and receive their own respective input configuration class. At the time of writing, only the basic configuration classes (`BasicVideoConfig`, `BasicAudioConfig` or `BasicSubtitleConfig`) are used for this purpose. These created objects will serve as reference for the output configuration classes specified by the user in the configuration file supplied at initialisation. As described in the “Software Architecture” Section, input configuration objects only have 1 purpose: manage stream type specific data (e.g., `BasicVideoConfig` manages width, height, aspect ratio and frame rate).

Internally, every configuration object and the `EncodeConfig` object have a reference set to a `ConflictManager` object, whose purpose is to collect conflicts. The system differentiates between two types of conflicts: *warnings* and *errors*. Warnings are simple conflicts between user supplied parameters and source artifact probed data, which could cause unwanted results in the end product (e.g., the source artifact contains a resolution of 1920×1080 pixels, but the user supplies a configuration that requires a resolution of 3840×2160 ; this could potentially cause video artifacts like blur in the transcoded output which could be unwanted). Errors on the other hand are conflicts which are of such severity that they’ll cause FFmpeg to halt encoding (e.g., The FFmpeg X264 library that encodes and decodes H264 video streams can only handle resolutions which are even during encoding; a requested output of resolution 1921×1080 would halt the encoding process whilst 1920×1080 would not).

Garbage Cleaning

Every output configuration object is responsible for knowing how to encode the source artifact with the user supplied parameters in the configuration file. Since the configuration classes know exactly which files they will create, they pass their garbage files (logs, pass files, probe data, ...) to the garbage cleaner. When the `vacuum` option is set to `True` in the configuration file (see for example Listing 4.1), the system will automatically clean all flagged files.

Internally, every configuration object has a `garbage_cleaner` parameter whose value is a `GarbageCleaner` object. The configuration classes also include a `_add_garbage(level, file)` method which can be called to flag a file as being garbage. The system currently supports 3 garbage levels: `intermediate_media` for the intermediate media files produced during transcoding, `logs` for multi-pass encode logs (see Section 4.2.4 for more information) and `OTHER` for everything that does not fall within the previous garbage levels.

Parameter Manager

The `ParameterManager` class forms the bridge between the configuration classes and the user configuration input. Every configuration class knows best what parameters it needs in order to fulfill its duty within the hierarchy. All configuration classes contain a static `construct_configuration_parameters(parameter_manager)` method, which tells the `ParameterManager` object what to look for inside the user specified configuration file. The configuration class specifies 3 kinds of information needed in the `ParameterManager`:

1. The key to look for within the `intermediate_files` configuration dictionary, we will call this the *configuration class key* (e.g. `h264` will map the user specified data within that key to the `H264VideoConfig` configuration class)
2. What keys to look for within the the user supplied configuration class key, we will call these the *parameter keys* (e.g. `media_height`, `h264_level`, `h264_profile` are parameter keys within a

`h264` configuration)

- A default fallback value also gets set in case the user specified configuration did not contain the specified parameter key
3. A method reference which will create an instance of the configuration class, we will call this the *creator method*

At initialisation, all configuration classes register themselves with a `ParameterManager` object. This object will then contain all configuration class keys with their respective parameter keys to look for within the `intermediate_files` dictionary of the user specified configuration file. When a configuration class key gets found, the system maps all user specified parameter keys against the parameter keys it expects and builds a dictionary from these. That dictionary gets passed to the creator method⁵ along with the reference configuration object, garbage cleaner object and conflict manager object. The creator method will then create the output configuration object.

4.2.4 Currently supported codecs

Bendwit CLI supports two video codecs, one audio codec and one subtitle codec at the time of writing. Adding a new codec is as simple as adding a new extension to the configuration hierarchy and registering it with the other configuration classes. Adding or updating functionality becomes easy due to this modular approach and makes it possible to let Bendwit work with any codec imaginable. If, for example, a future FFmpeg update were to add a new codec, one would only need to add the configuration class and Bendwit CLI would handle the rest. The mechanism was built in such a fashion that it is future proof.

H264

The H264 codec is supported in the `H264VideoConfig` configuration class; which accepts the following settings via the user configuration file (see Section 3.2.1 for more information about H264):

Parameter	Value type	Description
<code>source_index</code>	Integer	Video stream index from input media artifact which will be transcoded with the given parameters
<code>media_height</code>	Integer	Height expressed in pixels (if <code>media_width</code> is not supplied, it will automatically scale based on the aspect ratio)
<code>media_width</code>	Integer	Width expressed in pixels (if <code>media_height</code> is not supplied, it will automatically scale based on the aspect ratio)
<code>framerate</code>	String	Framerate in frames per second or in fraction format(e.g. <code>24</code> , <code>24000/1000</code>)
<code>kilobitrate</code>	Integer	Bitrate in kilobit per second
<code>aspectratio</code>	String	Aspect ratio expressed as <code>width:height</code> (e.g. <code>16:9</code>)
<code>h264_profile</code>	String	Currently supported profiles: baseline, main, high, high10, high422 and high444
<code>h264_level</code>	String	Currently supported levels: 1, 1b, 1.1, 1.2, 1.3, 2, 2.1, 2.2, 3, 3.1, 3.2, 4, 4.1, 4.2, 5, 5.1, 5.2

GOP plays an important role in preparing media for adaptive streaming. Every segment is required to start with an I frame, the `H264VideoConfig` class will decide on a *GOP* size based on the given segment duration and the framerate, this way, every segment boundary starts on an I frame. The segment duration is expressed in milliseconds internally to avoid rounding errors with floating point calculation. The *GOP* size gets calculated as follows:

$$\text{GOP} = \frac{\text{FPS} \times \text{segment length}}{1000}$$

Problems arise whenever the calculated *GOP* size is not an integer. To fix this, the calculated floating point *GOP* size gets rounded down to an integer and the segment duration in milliseconds gets recalculated

⁵The system makes use of `**var` syntax within Python to map dictionary keys to method header parameters.

with the following formula:

$$\text{segment length} = \frac{\text{round}(\text{GOP})}{\text{FPS}} \times 1000$$

It is recommended that every intermediate file configuration has the same framerate, this way if GOP calculation problems arise, they will all be recalculated to the same length. DASH allows video segments of different lengths, Bendwit on the other hand opts for segments of the same length to support as many client players as possible. In order to achieve quality switching between all the different transcoded versions, they would need to have the same segment lengths. The conflict manager will throw warnings about this to prevent such results.

In order to enforce a certain *bitrate* onto the requested intermediate files, the `H264VideoConfig` opts for multi-pass encoding, more specifically two-pass encoding. The first pass will analyze the input media file. The second pass will perform the transcode with the analyzed data in order to achieve the highest quality possible with the given parameters. A maximum bitrate will be enforced by supplying the `kilobitrate` key, otherwise the bitrate of the media input file will be used.

AAC

The AAC audio codec is supported in the `AacAudioConfig` configuration class, which accepts the following settings via the user configuration file:

Parameter	Value type	Description
<code>source_index</code>	Integer	Audio stream index from input media artifact which will be transcoded with the given parameters
<code>channel_layout</code>	String	Currently supported audio layouts: mono, stereo/2.1, 5.1 and 7.1
<code>kilobitrate</code>	Integer	Bitrate in kilobit per second

WebVTT

The WebVTT subtitle codec is supported in the `VttSubtitleConfig` configuration class; which accepts the following settings via the user configuration file:

Parameter	Value type	Description
<code>source_index</code>	Integer	Subtitle stream index from input media artifact which will be transcoded with the given parameters

Supplying a subtitle stream index in the user configuration file is enough for Bendwit to convert it to WebVTT format and include the resulting WebVTT subtitles in the output.

4.2.5 Encountered problems

During development of Bendwit CLI, certain problems arose that received a bug fix or a workaround.

Incorrect Durations and Segmentation Warnings

Bendwit CLI extracts video, audio and subtitle streams into individual intermediate media files using FFmpeg. Video and audio are extracted into containers with exactly one stream, subtitles get extracted into plain text files. During DASH segmentation, Mp4box would throw a mix of the following two warnings:

[iso file] Unknown box type `gmhd` and [iso file] Unknown box type `gmin`. These two warnings did not clearly indicate where a problem had occurred, neither did searching for the warnings point to a solution. Box types `gmhd` and `gmin` indicate incorrectly formatted subtitles, which had nothing to do with the test cases. Another indicator that something went wrong, were the incorrect durations which Mp4box put into the manifest file.

Probing the video and audio intermediate media files showed that the container always carried 2 streams:

the selected stream for transcoding and a garbage stream. Adding the `-dn` option⁶ to ignore data streams did not ignore the garbage stream. A solution was found on the Ffmpeg mailing list⁷ and a Superuser topic⁸. By supplying the option `-map_chapters -1`, Ffmpeg will ignore chapter data streams, resulting in clean containers with only the selected stream. Without the chapter data stream present, Mp4Box would not choke on segmenting video files and correctly put the duration length into the manifest file.

DASH Playback Freezes

During development different media files were used to test the transcoding and segmenting ability of Bendwit CLI. A test setup consisted of the segmented media files, a DASH manifest file and a DASH player. All setups showed the exact same problem within the Dash.js player version 2.3⁹. When quickly seeking through the video and changing the playback time just before the player had a chance to buffer the previous content, the player would freeze indefinitely. This mostly seemed to happen around the beginning area of the video seekbar. When the Bitmovin HTML 5 player version 7¹⁰ was used, this problem could not be reproduced. This happened in the same time frame as the previously mentioned problem, indicating that the incorrect durations in the manifest file could be a possible trigger to this problem. Another possibility was incorrect GOP transcoding of the intermediate files, leading to misplaced “I” frames, causing the player to choke on these segments.

A small script was developed to check segmented video files for incorrectly placed “I” frames:

1. Concatenate all segments in correct order into one media file
2. Analyze all frames of this monolithic media file with FFprobe and output findings to a CSV file
3. Count all “I” frames and check for incorrect positions (an “I” frame should only occur at the beginning of a segment)

After analyzing multiple setups, no incorrectly transcoded or segmented media files were found.

A new release of Dash.js did not yield the same freezing results. A multitude of bugfixes were introduced concerning buffering and video seeking¹¹ between version 2.3 and 2.5. Leading to the belief the problem was never situated with Bendwit CLI. Listings 4.3 and 4.4 show examples of a manifest with subtitles embedded as segments versus a manifest with subtitles embedded as a TTML XML file.

WebVTT Subtitle Support in MP4Box

When segmenting for DASH, MP4Box only supports TTML encoded subtitles^[67] or subtitles as a stream inside a container. Ffmpeg can extract and convert a multitude of different subtitle encodings, including: ass, vtt, srt, ... but unfortunately not TTML. At the time of writing, an enhancement ticket asking for TTML support has been opened on the Ffmpeg tracker¹².

As a temporary workaround, Bendwit CLI provides MP4Box with a subtitle stream encoded in a container: `MP4Box -add <subtitle_file> <existing_intermediate_file>`

By supplying MP4Box with optional `:OPT` or `#OPT` suffixes during DASHing, we can ensure that the media file gets segmented in such a way that it does not contain multiplexed streams:

`<existing_intermediate_file>#video` extracts the video stream (`#audio` for audio) and

`<existing_intermediate_file>#trackID=2` extracts that second stream which is the added subtitle stream. This workaround does involve some overhead: a 1KB WebVTT subtitle file for a 30 second media file when segmented into segments of 3 seconds, required a total of 5,4KB. Listings 4.3 and 4.4 show

⁶<https://trac.ffmpeg.org/ticket/647>

⁷<http://ffmpeg.org/pipermail/ffmpeg-user/2012-May/006654.html>

⁸<https://superuser.com/questions/441361/strip-metadata-from-all-formats-with-ffmpeg>

⁹<https://github.com/Dash-Industry-Forum/dash.js/wiki>

¹⁰<https://bitmovin.com/html5-player/>

¹¹<https://github.com/Dash-Industry-Forum/dash.js/releases>

¹²<https://trac.ffmpeg.org/ticket/4859>

examples of manifests with subtitles embedded as MP4 segments and subtitles embedded as a TTML XML file.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT1.500S" type="static"
  mediaPresentationDuration="PT0H0M32.973S" maxSegmentDuration="PT0H0M3.000S"
  profiles="urn:mpeg:dash:profile:isoff-live:2011,http://dashif.org/guidelines/
  dash264">
3   <Period duration="PT0H0M32.973S">
4     <!-- Subtitle adaptation sets-->
5     <AdaptationSet segmentAlignment="true" lang="eng">
6       <SegmentTemplate timescale="1000" media="$RepresentationID$/segment_${Number}.m4s
  " startNumber="1" duration="3000" initialization="$RepresentationID$/segment_.m4s
  />
7       <Representation id="10" mimeType="application/mp4" codecs="wvtt" startWithSAP="1
  " bandwidth="1322" />
8     </AdaptationSet>
9   </Period>
10 </MPD>

```

Listing 4.3: DASH manifest file with subtitles embedded as MP4 segments

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT1.500S" type="static"
  mediaPresentationDuration="PT0H0M32.973S" maxSegmentDuration="PT0H0M3.000S"
  profiles="urn:mpeg:dash:profile:isoff-live:2011,http://dashif.org/guidelines/
  dash264">
3   <Period duration="PT0H0M32.973S">
4     <!-- Subtitle adaptation sets-->
5     <AdaptationSet segmentAlignment="true" lang="eng">
6       <Representation id="10" bandwidth="256" mimeType="application/ttml+xml">
7         <BaseURL>eng.ttml</BaseURL>
8       </Representation>
9     </AdaptationSet>
10   </Period>
11 </MPD>

```

Listing 4.4: DASH manifest file with subtitles embedded as a TTML XML file

FFmpeg segmenter

The segmenting and manifest creation for HLS is done with FFmpeg as MP4Box only supports MPEG-DASH. FFmpeg has the tendency to segment audio and subtitle files however it seems fit. Video streams are neatly segmented as told and showcase no problems when examined with the tool explained in the Section “DASH Playback Freezes”. Audio gets segmented almost perfectly and subtitle segments show big deviations. If for example a 10 second segment size is chosen, video segments will be 10 seconds long each, audio segments will vary between 10 tot 10,02 seconds each and subtitles will vary between 3 to 15 seconds. These different segment lengths in audio and subtitles make HLS playback in browsers showcase glitchy behaviour: new video segments will only be downloaded once the longest current segment is fully processed, thus showing a black screen for a certain amount of time. The current implementation has no workaround yet for this bug as this bug was discovered too late in the development cycle. An idea for a workaround however, is available. Instead of making use of the FFmpeg `-f segment` format, the CLI tool can generate a list of commands which extracts pieces of media by making use of the FFmpeg `-ss` seeking system¹³, which allows for frame-accurate extraction. By then manually creating the manifest file the deviation problem will be solved. This however requires a system that analyzes video, audio and subtitle streams, creates extraction commands for these streams and finally creates a manifest without the help of external tools like MP4Box or FFmpeg.

¹³<https://trac.ffmpeg.org/wiki/Seeking>

4.3 Job Scheduler

The job scheduler acts as a controller between the incoming user requests from the REST API and the Bendwit CLI tool. As explained in Section 4.2, the CLI tool is built as a standalone tool. As such, it has no notion about the world around it. The job scheduler has the following responsibilities:

- Queue incoming requests
- Download necessary media files locally such that the CLI tool can access them
- Prepare output directories
- Notify users about the completion of their request via an optional webhook

4.3.1 Choice of Programming Language

The job scheduler was designed with Python version 3¹⁴. This decision is motivated by the required functionality described in Section 4.3.2: JSON configuration files, concurrency, Linux inotify support¹⁵, directory management and the ability to easily use HTTP GET and POST. Python readily provides this functionality via these respective libraries:

- `json`
- `threading`
- `watchdog`¹⁶
- `os` and `shutil`
- `urllib3` HTTP communication

4.3.2 Inner Workings

The job scheduler is a piece of software which runs 24/7 and watches a certain *job directory* for file change events. Jobs are represented as JSON configuration files. Upon detecting such events, the job scheduler checks to see if the file that triggered the event is a valid JSON file with the following keys:

- `job` The type of job, current possibilities: `probe` or `has_transcode`
- `media_id` A media identification name
- `config_file` A Bendwit CLI configuration filename
- (Optional) `external_url` An HTTP media URL

The job scheduler itself contains multiple queues which are serviced by a multitude of worker threads. Depending on the user settings, supplied at the beginning of the job scheduler code, the job scheduler spawns multiple `ProbeWorker`, `HasTranscodeWorker` and `DownloadWorker` threads and exactly one `SortWorker` and `DogWatchEventHandler` thread. Each type of worker is linked to a queue, the system currently consist of four queues: *sort queue*, *download queue*, *probe queue* and *has_transcode queue*. Each queue works in a FIFO fashion, but other sorting algorithms can be applied to the queue items. For example, as part of future work, a priority sorted job scheduler could be implemented that gives precedence to high priority jobs (e.g., those jobs initiated by premium users).

Every newly added job gets added to the sort queue. It is the sort worker's responsibility to figure out where to store the job. If the job specified an `external_url` key, it will always be added to the download queue first. If no `external_url` key is specified, the system assumes that the involved media is already locally available and can be found with the `media_id` identification; as such, it will sort the

¹⁴<https://www.python.org/>

¹⁵<http://man7.org/linux/man-pages/man7/inotify.7.html>

¹⁶<https://pypi.python.org/pypi/watchdog>

job according to the `job` key into either the probe or the has_transcode queue.

The `ProbeWorker` and `HasTranscodeWorker` threads spawn dedicated Bendwit CLI instances with the specified `config_file` value as configuration file. Once these worker threads finish their jobs, they will clean up the job configuration and Bendwit CLI configuration files. If the user request specified a `webhook` (see Section 4.4 for more information), the system will send an HTTP POST to the specified webhook URL upon completing the job, signaling that the job is finished with a payload specific to the user request. Figure 4.4 depicts the flow of a job from start to end.

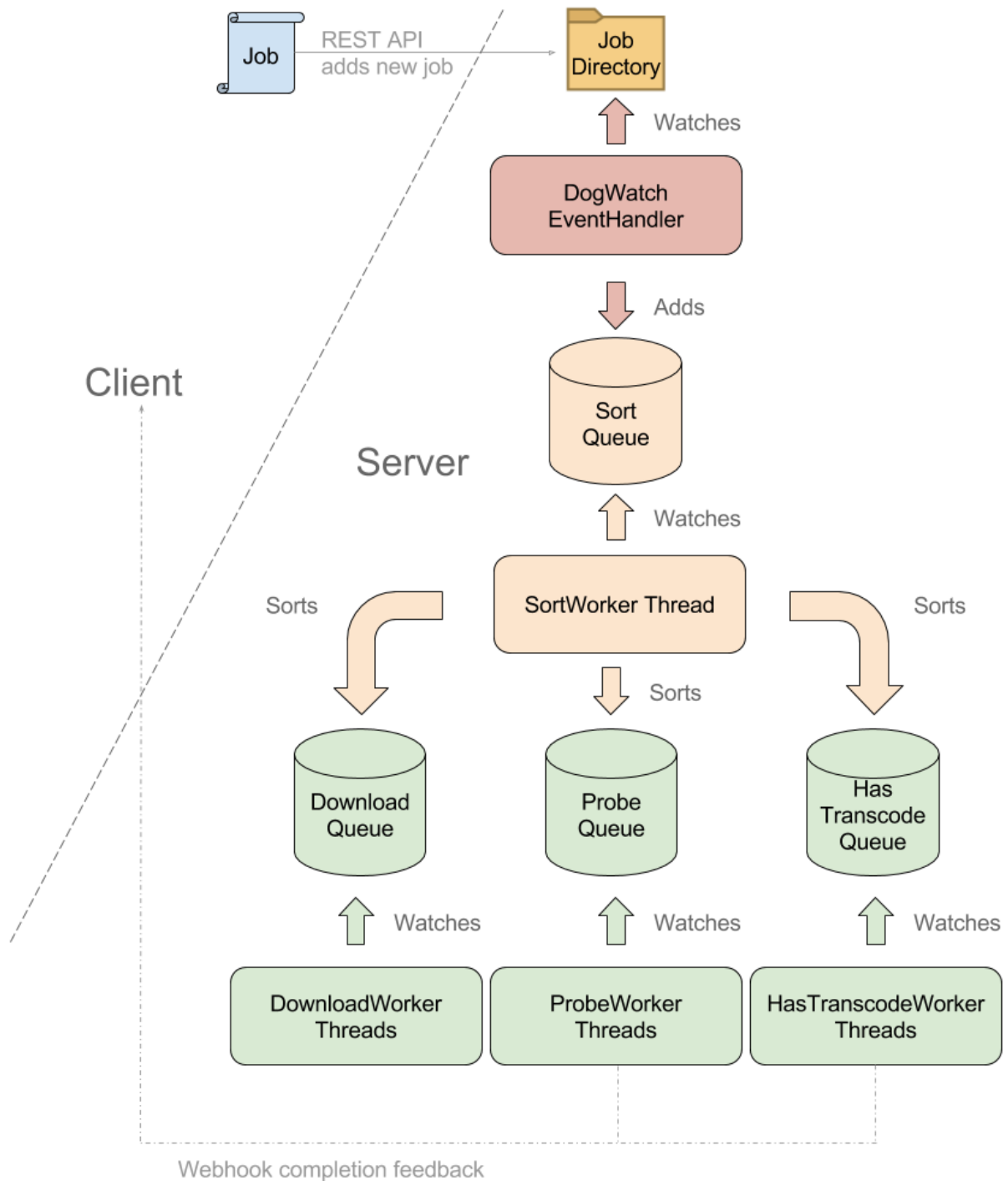


Figure 4.4: Bendwit job scheduler

4.4 REST API

All user requests are handled by the *REST API* which is publicly available via the link *api.bendwit.net* using representational state transfer services. The REST API has the following responsibilities:

- Create jobs for the job scheduler based on incoming user requests
- Create Bendwit CLI configs that accompany the job request
- Manage a database which stores information about media probes, media transcodes and manifests

4.4.1 Choice of Programming Language

The REST API and website (see Section 4.5 for more information) were designed as one part using the Python¹⁷ web-microframework Flask version 0.12.2¹⁸. Flask is a web-microframework based on Werkzeug¹⁹ and Jinja²⁰ and has a wide range of third-party plugins which simplify web development. The choice to merge the REST API and website under one codebase stems from the Flask architecture. It is possible to create multiple web server gateway interface entry points to the same codebase with Werkzeug. This enables us to create a REST API and website which use the same underlying code (e.g., models, configurations, Job Scheduler controller, ...) whilst providing different functionality.

In order to provide the REST API's functionality, the following flask plugins were used:

- `flask_restful` version 0.3.5²¹
- `flask_sqlalchemy` version 2.1²², which makes use of:
 - `sqlalchemy` version 1.2.0b2²³
- `jsonschema` draft 4²⁴

`flask_restful` provides an abstraction on top of Flask which makes it easier to develop REST functionality (e.g., auto parsing output formats XML, JSON, flat file, ...). The database model is defined in Python classes, which are mapped to database tables via the object-relational mapper of `sqlalchemy`, enabling easy modification and cross-database SQL functionality (e.g., development can take place with an SQLite database whilst in production a MySQL²⁵, PostgreSQL²⁶, ... interface can be used). Finally `jsonschema` provides a Python base way of checking JSON formatted strings, making it possible to syntactically check REST API request method bodies, which are JSON formatted, against a schema (see Section 4.4.3 for more information).

4.4.2 NGINX

*NGINX*²⁷ is an asynchronous event driven HTTP server with a small memory footprint, capable of handling many connections at once. REST API requests and website pages are not served directly from the Flask instance. Instead a deployment option called *uWSGI*²⁸ is used which communicates with the Flask WSGI. Whenever a client connects to the REST API or website, NGINX handles the connection logic and optional load balancing, passing along the request to the uWSGI instance via a unix socket;

¹⁷<https://www.python.org/>

¹⁸<http://flask.pocoo.org/>

¹⁹<http://werkzeug.pocoo.org/>

²⁰<http://jinja.pocoo.org/>

²¹<https://flask-restful.readthedocs.io>

²²<http://flask-sqlalchemy.pocoo.org/2.1/>

²³<https://www.sqlalchemy.org/>

²⁴<http://json-schema.org/>

²⁵<https://www.mysql.com/>

²⁶<https://www.postgresql.org/>

²⁷<https://www.nginx.com/>

²⁸<https://uwsgi-docs.readthedocs.io>

the uWSGI instance on its turn communicates with the Flask WSGI instance. The current information flow when a user makes an HTTP request looks like this:

Web Client \Leftrightarrow Web Server \Leftrightarrow Unix Socket \Leftrightarrow uwsgi \Leftrightarrow Flask

The reason the web server makes use of NGINX instead of Flask directly, stems from the fact that Flask was never to be deployed as a web server directly. Flask uses Werkzeug’s development server which as the name suggests was intended for development purposes and doesn’t handle high load well. NGINX on the other hand is a dedicated web server software which is capable of handling thousands of simultaneous connections and scales well.

4.4.3 Inner Workings

Internally, the REST API makes use of a versioning system based on the modular *blueprint* system of Flask. Each version is a Flask blueprint which gets added to the Flask instance at initialisation. A Flask blueprint can be considered as a module which contains functionality and can be added to a Flask instance. A user differentiates between REST API versions by specifying the desired version in the url during a request: `api.bendwit.net/v<x>/<request>`; which at the time of writing only consists of `v1`.

The REST API also requires users to identify themselves with an API key, which can only be created via the website interface (see Section 4.5 for more information) and is specified in an HTTP header with the name `bendwit-api-key` and the API key as value. The REST API logs all actions which take place with the key, these can be found on the website interface.

REST API request properties are passed as JSON arguments in the body of the HTTP request and get validated with a JSON schema. Whenever a POST message body contains faulty or missing properties, the REST API will return an HTTP 400 status code, indicating a bad request, together with a JSON object in the response body containing a list of errors.

A REST API request is always answered with a JSON response object containing information about the request along with an HTTP status code. Every JSON answer object contains a `status` property indicating the status of the request. The combination of the HTTP status code and the JSON response object `status` property are enough to uniquely identify every possible REST API answer.

The following table describes the REST API version 1 request URLs and what request method to use. POST request JSON properties and JSON response objects are described in full detail below the following table. Listing 4.5 showcases a small REST API interaction example using curl²⁹. In this example the user requests the last added media artifacts, creates a new media artifact and requests a HAS preparation on a media artifact using the encodings supplied in the JSON structure.

²⁹<https://curl.haxx.se/>

URL	Request method	Description
/probes/<media_id>/	GET	Retrieve probe data about the specified media_id
/probes/	POST	Create a new probe request
/has_transcodes/<has_id>/	GET	Retrieve the manifest location of the given has transcode id
/has_transcodes/	POST	Create a new adaptive streaming transcode
/has_transcodes/<has_id>	DELETE	Delete the has transcode linked to the given id
/media/	POST	Create a new media_id for the specified media artifact
/media/<media_id>/	GET	Return whether the media_id has been probed and all has_transcode ids the system has on the given media_id
/media/<media_id>/	DELETE	Delete the given media_id with all its has_transcodes and probe information
/media/	GET	Return the latest 20 added media ids
/media/<page>/	GET	Same as /media/, except the first <page> × 20 results are ignored

URL	Request method	REST API Status	HTTP Status
/media/	GET	media.{finished in_progress}	200
<p>The JSON response object consists of a list of the last 20 added media objects, each with the following properties:</p> <ul style="list-style-type: none"> • <code>status</code> • <code>media_id</code> <p>If the media artifact is ready for manipulation, the status will read as finished, else an in_progress status is returned; indicating that the user should wait before requesting a probe or has_transcodes.</p>			
/media/<page>/	GET	media.{finished in_progress}	200
<p>Returns the same response body as the GET request to /media/, except the first <page> × 20 results are ignored. This enables the user to retrieve media artifact ids older than the last 20 ones added.</p>			
/media/	POST	media.{finished in_progress}	201
<p>POST /media/ accepts as message body a JSON object with the following properties:</p> <ul style="list-style-type: none"> • <code>external_url</code> • <code>webhook</code> *30 • Listing A.3 shows the JSON schema for this URL <p>If the JSON object was formatted correctly, the following JSON response body properties are returned along with an HTTP 201 status code:</p> <ul style="list-style-type: none"> • <code>status</code> • <code>media_id</code> 			
/media/<media_id>/	GET	media.{finished in_progress}	200
<p>If <code>media_id</code> exists, the following JSON object is returned:</p> <ul style="list-style-type: none"> • <code>status</code> • <code>media_id</code> • <code>contains_probe</code> A boolean value representing whether the given <code>media_id</code> has been probed in the past • <code>has_transcodes</code> A list of has_transcode ids belonging to the given <code>media_id</code> <p>If the media artifact is ready for manipulation, the status will read as finished, else an in_progress status is returned; indicating that the user should wait before requesting a probe or has_transcodes.</p>			
/media/<media_id>/	GET	media.error	404
<p>If the given <code>media_id</code> does not exist, the REST API will return an error status along with an HTTP 404 status code. The REST API also returns this status in case the given <code>media_id</code> does not belong to the REST API key holder.</p>			
/media/<media_id>/	DELETE	media.deleted	200
<p>The delete request only issues a status property in the JSON response body together with an HTTP 200 status code indicating that the removal was successful.</p>			
/media/<media_id>/	DELETE	media.error	404
<p>If the given <code>media_id</code> does not exist, the REST API will return an error status along with an HTTP 404 status code. The REST API also returns this status in case the given <code>media_id</code> does not belong to the REST API key holder.</p>			

/probes/	POST	probe.in_progress	201
<p>POST /probes/ accepts as message body a JSON object with the following properties:</p> <ul style="list-style-type: none"> • <code>media_id</code> or <code>external_url</code> • <code>webhook</code> * • Listing A.1 shows the JSON schema for this URL <p>If the JSON object was formatted correctly, the following JSON response body properties are returned along with an HTTP 201 status code:</p> <ul style="list-style-type: none"> • <code>status</code> • <code>media_id</code> (In case an <code>external_url</code> property was provided in the POST body JSON object) 			
/probes/	POST	probe.error	404
<p>If the given <code>media_id</code> does not exist, the REST API will return an error status along with an HTTP 404 status code. The REST API also returns this status in case the given <code>media_id</code> does not belong to the REST API key holder.</p>			
/probes/	POST	probe.error	400
<p>If the given <code>media_id</code> has already been probed in the past, the REST API will return an error status along with an HTTP 400 status code.</p>			
/probes/	POST	probe.wait	409
<p>If the given <code>media_id</code> is not ready for manipulation yet, the REST API will return a wait status along with an HTTP 409 status code. It is recommended retry after a minimal waiting period of 5 seconds.</p>			
/probes/<media_id>/	GET	media.{finished in_progress}	200
<p>If <code>media_id</code> exists, the following JSON object is returned:</p> <ul style="list-style-type: none"> • <code>status</code> • <code>payload</code> <p>If the Bendwit platform has finished probing the media artifact, a finished status will be returned along with probe data in the payload property; else an <code>in_progress</code> status will be returned with an empty payload.</p>			
/probes/<media_id>/	GET	media.error	404
<p>If the given <code>media_id</code> does not exist, the REST API will return an error status along with an HTTP 404 status code. The REST API also returns this status in case the given <code>media_id</code> does not belong to the REST API key holder.</p>			
/probes/<media_id>/	GET	media.error	400
<p>If the given <code>media_id</code> has never received a probing request in the past, the REST API will return an error status along with the HTTP 400 status code.</p>			

<code>/has_transcodes/</code>	POST	<code>has_transcode.in_progress</code>	201
<p>POST <code>/has_transcodes/</code> accepts as message body a JSON object with the following properties</p> <ul style="list-style-type: none"> • <code>media_id</code> or <code>external_url</code> • <code>webhook</code> * • <code>type</code> : <code>dash</code> or <code>hls</code> • <code>segment_length</code> (in milliseconds) • <code>encodings</code> <ul style="list-style-type: none"> – <code>h264</code> * – <code>aac</code> * – <code>webvtt</code> * – Encodings accept the parameters explained in Section 4.2.4 • Listing A.2 shows the JSON schema for this URL <p>If the JSON object was formatted correctly, the following JSON response body properties are returned along with an HTTP 201 status code:</p> <ul style="list-style-type: none"> • <code>status</code> • <code>media_id</code> (In case an <code>external_url</code> property was provided in the POST body JSON object) • <code>has_id</code> 			
<code>/has_transcodes/</code>	POST	<code>has_transcode.error</code>	404
<p>If the given <code>media_id</code> does not exist, the REST API will return an error status along with an HTTP 404 status code. The REST API also returns this status in case the given <code>media_id</code> does not belong to the REST API key holder.</p>			
<code>/has_transcodes/</code>	POST	<code>has_transcode.error</code>	409
<p>If the given <code>media_id</code> is not ready for manipulation yet, the REST API will return a wait status along with an HTTP 409 status code. It is recommended retry after a minimal waiting period of 5 seconds.</p>			
<code>/has_transcodes/<has_id>/</code>	GET	<code>has_transcode.{finished in_progress}</code>	200
<p>If <code>has_id</code> exists, the following JSON object is returned:</p> <ul style="list-style-type: none"> • <code>status</code> • <code>manifest_url</code> <p>If the Bendwit platform has finished preparing the media artifact for adaptive streaming, a finished status will be returned along with a manifest url. If the has preparation is still in progress, an in_progress status will be returned with no <code>manifest_url</code> property.</p>			
<code>/has_transcodes/<has_id>/</code>	GET	<code>has_transcode.error</code>	404
<p>If <code>has_id</code> exists, the following JSON object is returned:</p> <ul style="list-style-type: none"> • <code>status</code> • <code>manifest_url</code> <p>If the given <code>has_id</code> does not exist, the REST API will return an error status along with an HTTP 404 status code.</p>			

```

1 # Retrieve the last 20 added media artifacts
2 curl -H "bendwit-api-key: 9245ba7120254b54aea03566fc462694" -X GET https://api.bendwit
   .net/v1/media/
3
4 # Create a new media artifact with a name
5 curl -H "bendwit-api-key: 9245ba7120254b54aea03566fc462694" -X POST -d '{"external_url
   " : "http://www.sample-videos.com/video/mp4/720/big_buck_bunny_720p_1mb.mp4", "
   name" : "Big Buck Bunny Example Video"}' https://api.bendwit.net/v1/media/
6
7 # Request a HAS preparation for the given media_id and with the given encodings
8 curl -H "Content-Type: application/json" -H "bendwit-api-key: 9245
   ba7120254b54aea03566fc462694" -X POST -d '{"media_id":"
   bea6827dcb504f8daff85d3fa9c5061a","type":"dash","segment_length":3000,"encodings
   ":{"h264":[{"source_index":0,"media_height":360,"media_width":640,"h264_profile":'
```

```
baseline", "h264_level": "3.1", "kilobitrate": 500, "framerate": 24}, {"source_index": 0, "media_height": 720, "media_width": 1208, "h264_profile": "baseline", "h264_level": "3.1", "kilobitrate": 2400, "framerate": 24}, {"source_index": 0, "media_height": 1080, "media_width": 1920, "h264_profile": "high", "h264_level": "4.1", "kilobitrate": 6000, "framerate": 24}], "aac": [{"source_index": 1, "channel_layout": "stereo", "kilobitrate": 36}, {"source_index": 1, "channel_layout": "stereo", "kilobitrate": 56}]]}' https://api.bendwit.net/v1/has_transcodes/
```

Listing 4.5: Direct REST API interaction example using curl

4.5 Website

As mentioned earlier in Section 4.4, the website is consolidated into the same codebase as the REST API, making it easy to reuse the same database models, use the same user system model and avoid code duplication. During development, the choice was made to create a simple-hearted website which showed the basic functionality of the Bendwit platform. As explained in Section 4.2, the Bendwit CLI tool forms the core of the Bendwit platform, everything added on top is just a (more) convenient way to interface with the Bendwit CLI tool. That is why the website only showcases the basic functionality: preparing media for adaptive streaming. If so desired, one can for example create a sugarcoated version which includes a payment wall for premium members. At the time of writing, the website has the following responsibilities:

- Manage user accounts
- Manage API keys and their logs
- Provide a user interface with novice abstraction which makes use of the REST API

4.5.1 Choice of Programming Language

Section 4.4.1 already explained the choice of Python and Flask for the REST API and the website. The website uses the following third-party Flask plugins:

- `flask_menu`³¹
- `flask_user`³²

`flask_menu` provides a simple way to define a menu in the view system of Flask so that the menu can be built dynamically based on the available blueprints, an example of this can be seen on the left in the Figures 4.5 and 4.6. `flask_user` provides a fully customisable user and authentication system.

The graphical part of the website makes use of:

- Twitter Bootstrap CSS framework v3.3.7³³
- jQuery javascript framework v3.2.1³⁴

The Twitter Bootstrap CSS framework is used to build a responsive and mobile-friendly website layout. DOM manipulation and Bendwit REST API interaction is done with the jQuery javascript framework.

4.5.2 Inner Workings

The website consists out of two parts. Firstly a part which manages the API keys, explained in the “API Keys” Section and secondly a graphical interface for the REST API that prepares content for adaptive streaming, explained in the “HAS Preparation” Section.

³¹<http://flask-menu.readthedocs.io>

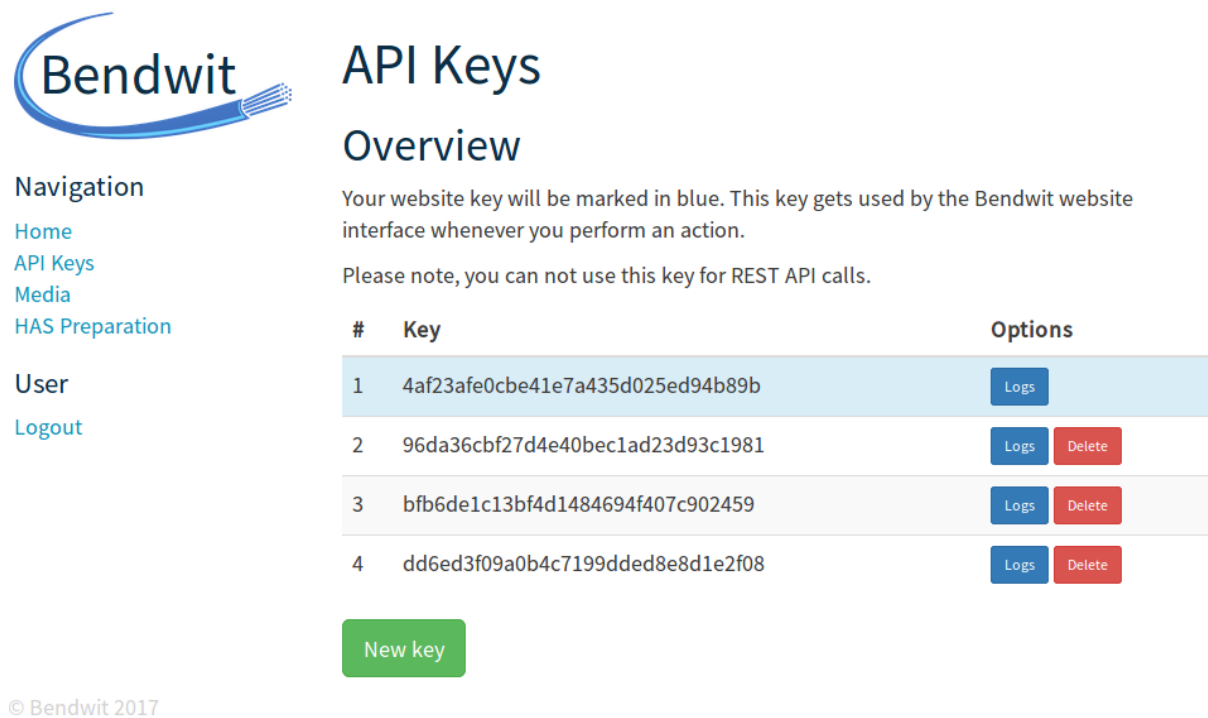
³²<https://pythonhosted.org/Flask-User/>

³³<http://getbootstrap.com/>

³⁴<https://jquery.com/>

API Keys

The REST API makes use of a key based authentication system. The only way to acquire one of such keys is via the website interface (see Figure 4.5). A Bendwit user account has the ability to generate keys and hold up to 5 API keys³⁵ at the same time. When a REST API key gets used, its actions are logged. The log files of individual API keys are accessible via the website interface (see Figure 4.6). There is no security system in place at the time of writing to prohibit the use of an API key if someone other than the owner gets a hold of it; deleting the key and generating a new one is the only option in such a case.



Bendwit

API Keys

Overview

Your website key will be marked in blue. This key gets used by the Bendwit website interface whenever you perform an action.

Please note, you can not use this key for REST API calls.

#	Key	Options
1	4af23afe0cbe41e7a435d025ed94b89b	Logs
2	96da36cbf27d4e40bec1ad23d93c1981	Logs Delete
3	bfb6de1c13bf4d1484694f407c902459	Logs Delete
4	dd6ed3f09a0b4c7199dded8e8d1e2f08	Logs Delete

[New key](#)

© Bendwit 2017

Figure 4.5: Bendwit website API keys page

³⁵A total of 6 API keys if the website API key is taken into the summation. The website API is only used by the website interface when interacting with the REST API, this key is protected against outside use and can not be deleted by the user.



API key logs

Logs for key "4af23afe0cbe41e7a435d025ed94b89b"

Navigation

[Home](#)
[API Keys](#)
[Media](#)
[HAS Preparation](#)

User

[Logout](#)

Message	Date
Media artifact "2fa3eb7733b043559402a7dd9fa4be45" transcode requested, transcode id "0acdda2d0ac1441085eff4cba8179766".	2017-08-19 00:04:58.916750
Media artifact "24f074b9f256463fbb0be965c467bd59" probe requested.	2017-08-19 00:04:43.263491
New media artifact uploaded "24f074b9f256463fbb0be965c467bd59"	2017-08-19 00:04:43.245112
Media artifact "2fa3eb7733b043559402a7dd9fa4be45" probe requested.	2017-08-19 00:02:53.723123
New media artifact uploaded "2fa3eb7733b043559402a7dd9fa4be45"	2017-08-19 00:02:53.711890
Media artifact "3c2e8aad6f5a4a9db269915b5b0105f5" probe requested.	2017-08-19 00:02:21.025688
New media artifact uploaded "3c2e8aad6f5a4a9db269915b5b0105f5"	2017-08-19 00:02:21.005963
Media artifact "fb3b1b167eeb445fb0e21a731c5c898d" transcode requested, transcode id "dc8578cd79cd4a4090d7fe1538edba69".	2017-08-18 23:46:53.570244
Media artifact "Thesis CHills" probe requested.	2017-08-18 23:46:22.216134
New media artifact uploaded "Thesis CHills"	2017-08-18 23:46:22.198711


1	2	3	4	5	...	9	10
---	---	---	---	---	-----	---	----

Figure 4.6: Bendwit website API key logs

Media

The *Media* page consists of a list of all media artifacts linked to the Bendwit user account that is currently logged in. The list shows the latest media artifact entries first, as shown in Figure 4.7. On the *Media* page, users have the ability to upload new media artifacts by pressing the green *New* button. They will then be prompted to supply either an http(s) direct link to a media artifact or pick a media file from their personal dropbox (depicted in Figure 4.8).

Users also have the option to manage their media artifacts from the *Media* page. A media artifact can be deleted by pressing the red *Delete* button next to a media artifact, the user will need to confirm a prompt in order to continue. Unprobed media artifacts also show the option to probe them, this is done by pressing the orange *Request probe* button, which will request a probe for the respective media artifact (depicted in Figure 4.9). By pressing the *Details* button, a details modal will show the user a more detailed description of the media artifact and all has_ids which have this media artifact as input artifact as shown in Figure 4.10. Whenever a user wants to prepare the media artifact for adaptive streaming, all they need to do is press the *HAS prepare* button which will take them to the *Has Preparation* page (see Section “HAS Preparation” for more details).



Media

Overview

New

Navigation

- Home
- API Keys
- Media
- HAS Preparation

User


- Logout

Name	Probe	Options
Thesis Example Video	✗	HAS prepare Details Delete Request probe
Monkey Video	✗	HAS prepare Details Delete Request probe
24f074b9f256463fbb0be965c467bd59	✓	HAS prepare Details Delete
2fa3eb7733b043559402a7dd9fa4be45	✓	HAS prepare Details Delete
3c2e8aad6f5a4a9db269915b5b0105f5	✓	HAS prepare Details Delete

1 2 3 4 5 ... 10 11

© Bendwit 2017

Figure 4.7: Bendwit website Media page



Media

Upload media

New

Navigation

- Home
- API Keys
- Media
- HAS Preparation

User

- Logout

Name

HTTP URL

Set input media

Dropbox File

Choose from Dropbox


Close

3c2e8aad6f5a4a9db269915b5b0105f5	✓	HAS prepare Details Delete
----------------------------------	---	----------------------------

1 2 3 4 5 ... 10 11

© Bendwit 2017

Figure 4.8: Upload media artifact form from the Media page on the Bendwit website



Media

Overview

New

Navigation

- Home
- API Keys
- Media
- HAS Preparation

User

- Logout

Name	Probe	Options
Thesis Example Video	✗	HAS prepare Details Delete Probing...
Monkey Video	✗	HAS prepare Details Delete Request probe
24f074b9f256463fbb0be965c467bd59	✓	HAS prepare Details Delete
2fa3eb7733b043559402a7dd9fa4be45	✓	HAS prepare Details Delete
3c2e8aad6f5a4a9db269915b5b0105f5	✓	HAS prepare Details Delete

[1](#) [2](#) [3](#) [4](#) [5](#) [...](#) [10](#) [11](#)

© Bendwit 2017

Figure 4.9: Media artifact probe request from the Media page on the Bendwit website

Details

Media artifact

ID: adcdb11feee44ccd99e341dd08ee267d
 Name: Not set
 Contains probe: true

HAS prepared content

ID	Name
fb7cd894ef414360a3df9abbff2ffb19	

Probe data

Probe data

Video artifact data

Stream index	Codec name	Width	Height	Aspect ratio	Frame rate
0	h264	1920	1080	16:9	24000/1001

Audio artifact data

Stream index	Codec name	Channel #	Channel layout	Bitrate
1	vorbis	2	stereo	48000

Subtitle artifact data

Figure 4.10: Media artifact details form from the Media page on the Bendwit website

HAS Preparation

The website also provides a user interface for scheduling HAS media preparation jobs. The interface is an abstraction for novice users which can select a premade encoding profile for a certain range of target devices. The Bendwit platform will try to prepare the requested media according to the specified profile on a best-effort basis. If due to a corrupt or edge case media artifact the Bendwit CLI tool finds conflicts, the website interface will force the Bendwit CLI tool to prepare the media anyway on a best-effort basis.


The *HAS Preparation* page consists of several steps. The first step is the selection of a media artifact from the *Media* page by pressing the *Has prepare* button or the upload of a new media artifact on the *HAS Preparation* page itself. Once the media artifact is ready for manipulation, the Bendwit platform will probe the media artifact for basic information, this information is then displayed on the *HAS Preparation* page. The last step is to provide the wanted settings for the HAS representation. The user gets to choose a profile and which streams to include in the end result; the probe information is there to guide the user in their choices. In the final step, the Bendwit platform transcodes the media artifact to different qualities, segments these qualities and creates a manifest file according to the selected HAS type. When the HAS preparation finishes, the user will be shown the manifest location and an example playback of the HAS content. Figures 4.11, 4.12, 4.13, 4.14 and 4.15 depict the steps from start to finish.

As explained earlier, the user chooses a profile which will be used for transcoding the media artifact. A profile represents a premade collection of transcoding settings which will be applied to the input media artifact. The automated setup takes the resolution and aspect ratio into account when applying a profile. A media artifact will never be upscaled and the aspect ratio will be kept; if for example an input media artifact has a 9:16 aspect ratio, the automated setup will transcode the media in such a way that it keeps the vertical aspect ratio, this is done by switching the profile's height and width properties. A profile in its essence is just a collection of properties, this means that future additions or changes are perfectly possible. In the current implementation, the automated setup does not allow the user to make changes in a profile's properties as this requires knowledge about encoding settings and is more suited for the developer API (see Section 4.6). The following profiles are currently implemented:

Desktop Web Browsers - MPEG-DASH profile - 3 second segments	
Video quality 1 (H264)	Height of 360px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 500kbps; H264 baseline profile level 3.1
Video quality 2 (H264)	Height of 360px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 800kbps; H264 baseline profile level 3.1
Video quality 3 (H264)	Height of 720px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 2000kbps; H264 baseline profile level 3.1
Video quality 4 (H264)	Height of 720px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 3000kbps; H264 baseline profile level 3.1
Video quality 5 (H264)	Height of 1080px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 3000kbps; H264 baseline profile level 4.1
Video quality 6 (H264)	Height of 1080px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 4000kbps; H264 baseline profile level 4.1
Video quality 7 (H264)	Height of 1080px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 6000kbps; H264 baseline profile level 4.1
Audio quality 1 (AAC)	Mono channel layout; Bitrate of 36kbps
Audio quality 2 (AAC)	Stereo channel layout; Bitrate of 56kbps
Smartphones - MPEG-DASH profile - 3 second segments³⁶	
Video quality 1 (H264)	Height of 144px (width is automatically scaled accordingly); Framerate of 12fps; Bitrate of 56kbps; H264 baseline profile level 3.1
Video quality 2 (H264)	Height of 360px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 500kbps; H264 baseline profile level 3.1
Video quality 3 (H264)	Height of 720px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 2000kbps; H264 baseline profile level 3.1
Audio quality 1 (AAC)	Mono channel layout; Bitrate of 36kbps
Audio quality 2 (AAC)	Stereo channel layout; Bitrate of 56kbps
Apple Devices - HLS profile - 10 second segments³⁷	
Video quality 1 (H264)	Height of 270px (width is automatically scaled accordingly); Framerate of 15fps; Bitrate of 400kbps; H264 baseline profile level 3.0
Video quality 2 (H264)	Height of 360px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 800kbps; H264 baseline profile level 3.0
Video quality 3 (H264)	Height of 360px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 400kbps; H264 high profile level 4.1
Video quality 4 (H264)	Height of 720px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 5000kbps; H264 high profile level 3.1
Video quality 5 (H264)	Height of 720px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 4000kbps; H264 high profile level 4.1
Video quality 6 (H264)	Height of 1080px (width is automatically scaled accordingly); Framerate of 30fps; Bitrate of 8600kbps; H264 high profile level 4.1
Audio quality 1 (AAC)	Mono channel layout; Bitrate of 36kbps
Audio quality 2 (AAC)	Stereo channel layout; Bitrate of 56kbps

³⁶<https://developer.android.com/guide/topics/media/media-formats.html>

³⁷https://developer.apple.com/library/content/technotes/tn2224/_index.html



Transcode

Define input source artifact

Name

HTTP URL


Dropbox File

Navigation
[Home](#)
[API Keys](#)
[Media](#)
[HAS Preparation](#)

User
[Logout](#)


© Bendwit 2017

Figure 4.11: A user uploads a new media artifact on the HAS Preparation page



Transcode

Probing media artifact


Please wait...

Navigation
[Home](#)
[API Keys](#)
[Media](#)
[HAS Preparation](#)

User
[Logout](#)

© Bendwit 2017

Figure 4.12: The Bendwit platform probes a selected media artifact on the HAS Preparation page



Navigation

[Home](#)
[API Keys](#)
[Media](#)
[HAS Preparation](#)

User

[Logout](#)

Transcode

Probe data

Video artifact data

Stream index	Codec name	Width	Height	Aspect ratio	Frame rate
0	h264	1080	1350	4:5	25/1

Audio artifact data

Stream index	Codec name	Channel #	Channel layout	Bitrate
1	aac	2	stereo	48000

Subtitle artifact data

Stream index	Codec name
No subtitle streams found	

Transcode settings

Name

Adaptive Streaming Type

☒ MPEG-DASH ☐ Apple HLS

Profile

Video streams

☐ Stream 0

Audio streams

☐ Stream 1

Subtitle streams

☐ No options available

[Prepare content for adaptive streaming!](#)

© Bendwit 2017

Figure 4.13: A user fills in the wanted settings for their HAS content on the HAS Preparation page

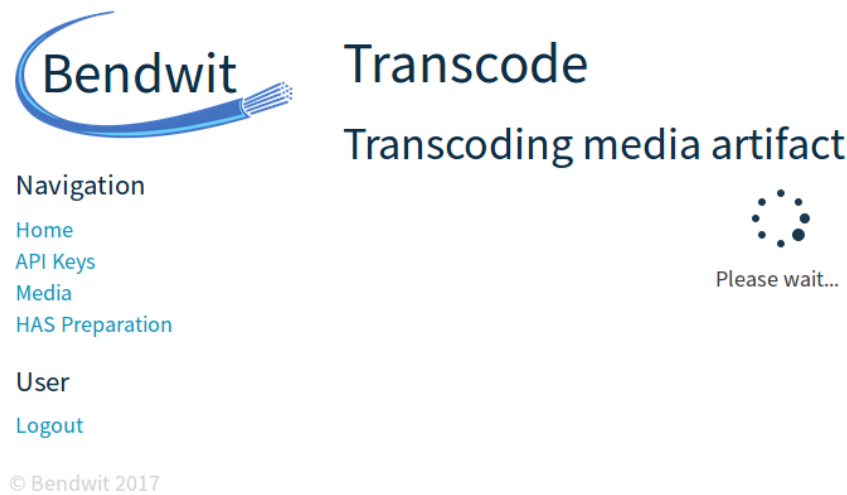


Figure 4.14: The Bendwit platform prepares a selected media artifact for adaptive streaming on the HAS Preparation page

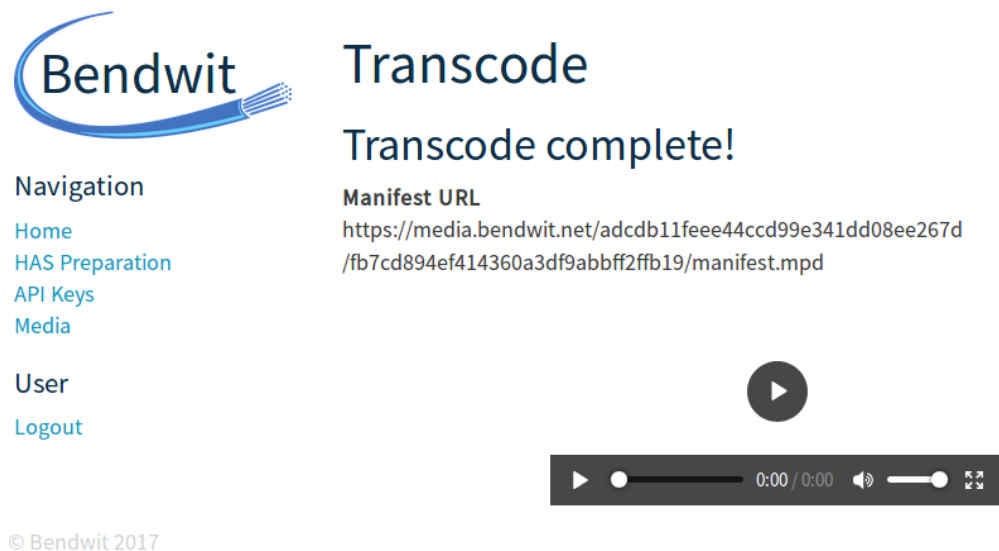


Figure 4.15: A media artifact has successfully been prepared for adaptive streaming on the HAS Preparation page

Media hosting

All media content prepared for adaptive streaming by Bendwit is hosted on `media.bendwit.net`. By default, modern browsers block script access to domains other than the domain the scripts are hosted on for security reasons, this is called the same-origin policy [68]. The way DASH and HLS work, client players such as the Bitmovin HTML 5 player³⁸ and Dash.JS³⁹ fetch media segments via Javascript. This violates the same-origin policy and thus requires cross-domain access control, this is done by enabling HTTP headers for cross-origin resource sharing. The `media.bendwit.net` domain adds the following HTTP headers in order to make cross-domain resource access possible:

- 1 `Access-Control-Allow-Origin: "*"`
- 2 `Access-Control-Expose-Headers: "Server,range"`
- 3 `Access-Control-Allow-Methods: "GET, HEAD, OPTIONS"`

³⁸<https://bitmovin.com/video-player/>

³⁹<https://github.com/Dash-Industry-Forum/dash.js/wiki>

4 Access-Control-Allow-Headers: "origin, range"

The media itself is stored in a directory structure which is known to the Bendwit CLI tool, the job scheduler and the REST API. Every media artifact is stored in a directory named after the `media_id` of the media artifact. The artifact itself is always stored under the filename `raw_media.bendwit` in the root of the media directory. Probe information is also kept in a `probe.json` file in the root directory. Every time the user requests a HAS preparation, a new `has_id` gets generated and a new directory with the same name gets added to the root of the media directory; this folder contains all segments, intermediate encodings and the manifest for streaming. Figure 4.16 depicts the directory structure.

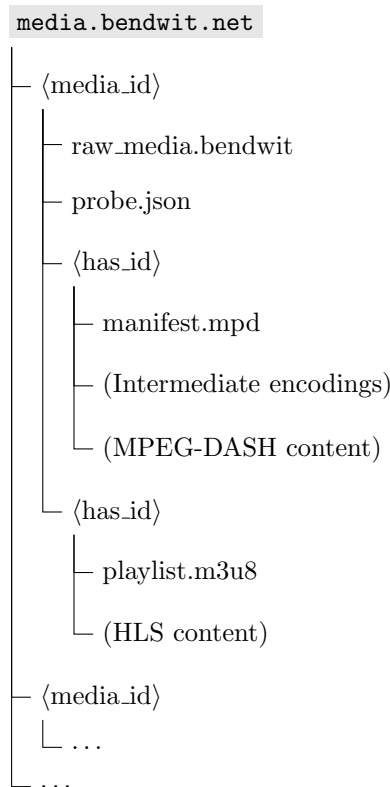


Figure 4.16: `media.bendwit.net` directory structure

4.6 Python Developer API

Another way of interacting with the Bendwit platform is via the Python developer API interface. This developer API is aimed at expert users wanting to simplify the integration of their programming projects with the Bendwit platform. At the time of writing the Python developer API provides two ways of interacting with the platform: a functional way and an object-oriented way. The functional way is very similar to direct communication with the REST API. All Python functional methods raise Python exceptions corresponding to REST API errors, this enables developers to use the Pythonic “ask forgiveness not permission” style of coding. The object-oriented implementation makes use of the functional methods to implement a small system that abstracts some error handling. For example, the developer only has to request probe information from a media artifact, the object-oriented implementation will handle the probe request (if needed) and polling after the probe information. Developers only using the functional methods would need to implement a polling mechanism themselves making use of the functional methods explained in Section 4.6.1. The object-oriented abstraction is explained in Section 4.6.2.

In order to use the Python developer API, one needs to set the Bendwit API key in order to identify themselves with the Bendwit REST API; this can be done by manually filling in the REST API key in the Python developer API source code or by using the `set_api_key(key)` method.

4.6.1 Functional methods

The following tables describe the available methods in the Python developer API at the time of writing, together with their function signature and the Python exceptions they raise. All function names and parameters were chosen in such a way that the function definitions denote what they do. It should be noted that all functional methods work in a halting fashion: the method will block for as long as the REST API call(s) that it encapsulates requires to fulfill the request. If a developer wishes non-blocking behaviour, they are able to build a threaded shell around these functional methods.

<code>create_media_artifact(external_url, name)</code>	
Description	Create a new media artifact from the given <code>external_url</code> and optional <code>name</code> parameters. Upon creation the newly generated <code>media_id</code> is returned.
Exceptions	The following exceptions can be raised: <ul style="list-style-type: none"> • <code>MediaArtifactException</code> whenever media creation fails
<code>delete_media_artifact(media_id)</code>	
Description	Delete an existing media artifact.
Exceptions	The following exceptions can be raised: <ul style="list-style-type: none"> • <code>MediaArtifactException</code> whenever media deletion fails
<code>get_media_artifact_data(media_id)</code>	
Description	Get media artifact information; this includes whether the media artifact has been probed before and all its HAS prepared content IDs.
Exceptions	The following exceptions can be raised: <ul style="list-style-type: none"> • <code>MediaArtifactException</code> if the <code>media_id</code> does not exist
<code>get_media_list(page)</code>	
Description	Get the last 20 media artifacts added to the API key's user account. If a page number is supplied, the first $\langle \text{page} \rangle \times 20$ results are ignored.
Exceptions	The following exceptions can be raised: <ul style="list-style-type: none"> • <code>MediaArtifactException</code> if the media list could not be retrieved
<code>create_probe_request(media_id)</code>	
Description	Create a probe request for the supplied <code>media_id</code> .
Exceptions	The following exceptions can be raised: <ul style="list-style-type: none"> • <code>MediaArtifactException</code> if an invalid <code>media_id</code> was supplied • <code>ProbeWaitException</code> if the media artifact is not ready for manipulation yet • <code>ProbeDuplicateException</code> if the media artifact has already received a probe request in the past
<code>get_probe_data(media_id)</code>	
Description	Retrieve the probe data for the supplied <code>media_id</code> . The return value of this method is the probe payload in Python dictionary format.
Exceptions	The following exceptions can be raised: <ul style="list-style-type: none"> • <code>MediaArtifactException</code> if an invalid <code>media_id</code> was supplied • <code>ProbeInProgressException</code> if the media artifact probe has not been fully processed yet by the Bendwit platform • <code>ProbeDoesNotExistException</code> if the media artifact has never received a probe request before

<code>create_has_preparation(media_id, has_type, segment_length, encodings)</code>	
Description	Create a HAS preparation request for the supplied <code>media_id</code> . The return value of this method is the HAS id for request being made. <code>encodings</code> follows the same format as the “encodings” property of the REST API HAS preparation request explained in Section 4.4.3.
Exceptions	The following exceptions can be raised: <ul style="list-style-type: none"> • <code>MediaArtifactException</code> if an invalid <code>media_id</code> was supplied • <code>HasPreparationWaitException</code> if the media artifact is not ready for HAS preparation yet

<code>get_has_manifest(has_id)</code>	
Description	Retrieve the manifest URL for the supplied <code>has_id</code> .
Exceptions	The following exceptions can be raised: <ul style="list-style-type: none"> • <code>HasIdException</code> if an invalid <code>has_id</code> was supplied • <code>HasPreparationInProgressException</code> if the HAS preparation request has not been fully processed yet

4.6.2 Object-Oriented Abstraction

As briefly mentioned in Section 4.6, the object-oriented abstraction makes use of the functional methods described in Section 4.6.1. The only difference between the functional methods and the object-oriented approach, is that the object-oriented approach abstracts some polling work and gathers everything in one class called `Media`. Whenever a developer wants to perform a request on a certain `media_id`, they can create a new `Media` object and for example directly call methods like `get_probe()` which will return the media artifact’s probe data. A media object can be created from an already existing `media_id` or by supplying the `Media` constructor with an `external_url` (and optional `name`) parameter which will create a new media artifact. The following methods are available on `Media` objects:

- `get_probe()` will retrieve the media artifact probe data. In case the media has not been probed yet, this will be taken care of automatically. This method returns the probe its payload.
- `prepare_has(encoding_creator_object)` will send out a HAS preparation request and keep polling until the Bendwit platform finishes the HAS preparation. The `encoding_creator_object` is explained below. This method returns the manifest URL of the HAS preparation request.

The object-oriented approach also provides an abstraction for the creation of encodings. By creating a new `EncodingCreator` object, a developer can create encodings by using methods instead of dictionary/JSON structures. This is done with the following methods incorporated in the `EncodingCreator` class:

- `add_h264_encode()`
- `add_aac_encode()`
- `add_webvtt_encode()`

All these methods accept the same properties as specified in the Section 4.4.3. Instead of supplying a dictionary, these methods accept parameters, see Listing 4.6 for an example.

```

1 from developer_api import developer_api_v1 as api
2
3 # Set the API key
4 api.set_api_key("9245ba7120254b54aea03566fc462694")
5
6 # Create a new media artifact object
7 media = api.Media(external_url="http://www.sample-videos.com/video/mp4/720/
    big_buck_bunny_720p_1mb.mp4", name="Developer API code - Example test")
8
9 # Request and print probe information

```

```
10 print(media.get_probe())
11 # Output: {'video': [{'avg_frame_rate': '25/1', 'height': 720, 'display_aspect_ratio':
    '16:9', 'codec_name': 'h264', 'width': 1280, 'index': 0}], 'subtitle': [], 'audio
    ': [{'sample_rate': '48000', 'codec_name': 'aac', 'index': 1, 'channel_layout':
    '5.1', 'channels': 6}]}
12
13 # Create some encoding configurations: 2 video qualities and 1 audio quality
14 encode = api.EncodingCreator(api.EncodingCreator.EncodingType.MPEG_DASH, 3000)
15 encode.add_h264_encode(source_index = 0, framerate=24, h264_profile="baseline",
    h264_level="3.1", media_height=360, kilobitrate = 600)
16 encode.add_h264_encode(source_index = 0, framerate=24, h264_profile="baseline",
    h264_level="3.1", media_height=720, kilobitrate = 2400)
17 encode.add_aac_encode(source_index = 1, channel_layout = "stereo", kilobitrate = 56)
18
19 # Prepare the media artifact for adaptive streaming and retrieve the manifest URL
20 print(media.prepare_has_encode())
21 # Output: https://media.bendwit.net/8e939ff3a88c47d68596e98a0352d2e8/
    c85ca3dc93234cd4a79832f0bf18795c/manifest.mpd
```

Listing 4.6: Developer API example of object-oriented usage

Chapter 5

Evaluation - Time Study

As a way of evaluating the Bendwit platform, we shall conduct a small time study. The purpose of the Bendwit platform is to provide an automated way of preparing media content for adaptive streaming, which is achieved by abstracting the whole process in different layered systems as described in Chapter 4. Because of this approach, some overhead is to be expected when Bendwit is compared to a manual process of preparing media for adaptive streaming.

The way we will conduct this time study is by setting up four test cases (see Section 5.1 for more information). A test case consists of a media artifact which will be prepared for adaptive streaming. In order to find out the Bendwit overhead, we will use a script which sends a HAS preparation request to the REST API along with a webhook in the request. Two timestamps are logged, one when the request is sent and a second one when the webhook registers a call from the Bendwit platform. By subtracting these two timestamps, we get the total time it took the Bendwit platform to fulfill the request. This time is then compared to a manual preparation setup. The manual test case uses the same commands as the Bendwit platform does to keep things fair at the encoding step. The media artifact is already locally available for the Bendwit platform to use, uploading times are not represented in these test cases. A bash script¹ will be used to facilitate the transcoding, segmentation and manifest creation steps; at start and finish a timestamp will be logged. By subtracting these two timestamps we find the time it takes for the manual process to prepare the media artifact for adaptive streaming.

5.1 Test Cases

Four test cases will be used to evaluate the Bendwit platform. These test cases will each be run 10 times. For these four test cases, two media artifacts will be used. The first media artifact will be a 30 seconds slice from the Sintel open movie project², the second media artifact will be the full Big Buck Bunny project³ which is 9 minutes 56 seconds in length. Both these media artifacts will be prepared for adaptive streaming in the MPEG-DASH and HLS formats. Both media artifacts contain H264 video streams. In terms of audio, the Sintel media artifact contains a Vorbis encoded audio stream and the Big Buck Bunny project contains an AAC encoded audio stream. Both media artifacts are 1080p content and will be transcoded to the following qualities in all test cases:

- Video stream
 1. Low Quality
 - Resolution 640×360
 - Framerate of 24fps
 - Bitrate of 500kbps

¹By using a Bash script, we ensure no extra overhead when invoking the manual FFmpeg transcoding commands.

²<https://durian.blender.org/>

³<https://peach.blender.org/download/>

- H264 codec with the following settings:
 - * Baseline profile
 - * Level 3.1
- 2. Medium Quality
 - Resolution 1280×720
 - Framerate of 24fps
 - Bitrate of 2400kbps
 - H264 codec with the following settings:
 - * Baseline profile
 - * Level 3.1
- 3. High Quality
 - Resolution 1920×1080
 - Framerate of 24fps
 - Bitrate of 6000kbps
 - H264 codec with the following settings:
 - * High profile
 - * Level 4.1
- Audio stream
 1. Low Quality
 - Stereo channel layout
 - Bitrate of 36kbps
 - AAC codec
 2. Normal Quality
 - Stereo channel layout
 - Bitrate of 56kbps
 - AAC codec

Listings B.1 and B.2 show the different JSON objects that were sent as a POST body to the `/has_transcodes/` REST API. Listings B.3 and B.4 on the other hand show the raw FFmpeg and MP4Box commands required to manually prepare the content for adaptive streaming.

The expectation is that both media artifacts in all test cases will show a clear difference in timing because of the expected overhead the Bendwit platform introduces. The HLS test cases are expected to show bigger time differences due to the way Bendwit creates the master m3u8 playlist (described in Chapter 4).

5.2 Findings

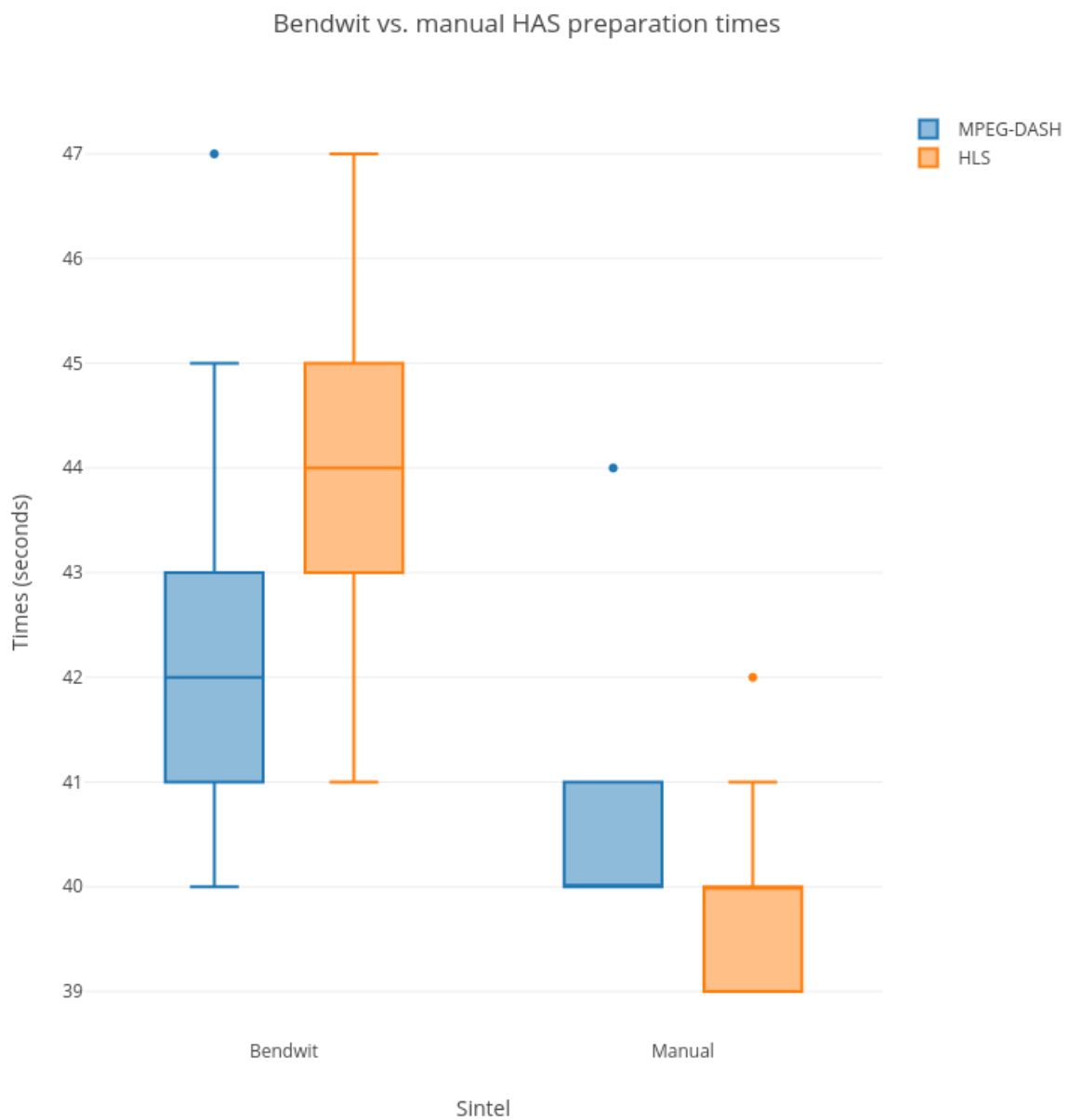


Figure 5.1: Sintel test case: Bendwit preparation times versus manual preparation times boxplots

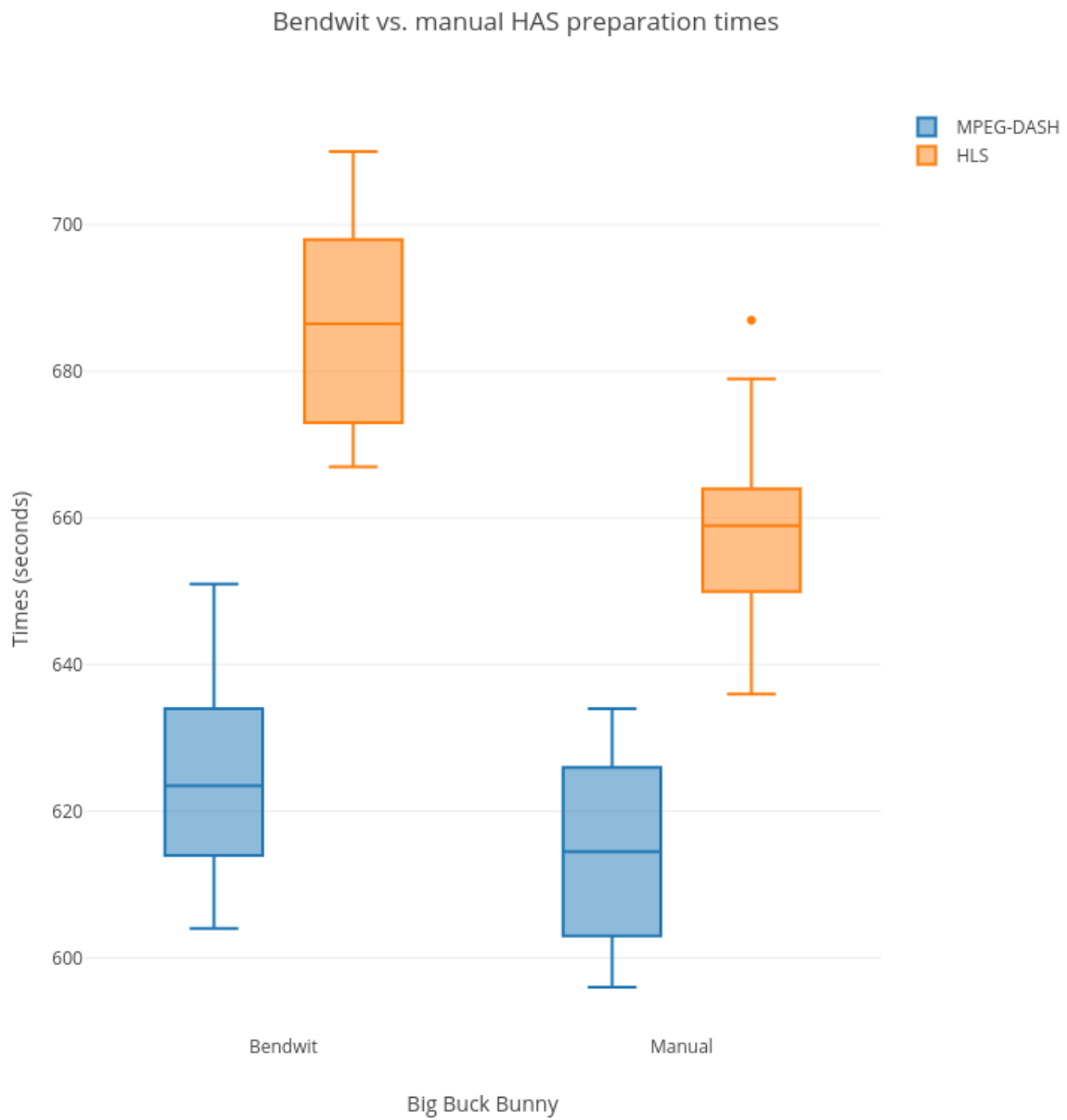


Figure 5.2: Big Buck Bunny test case: Bendwit preparation times versus manual preparation times boxplots

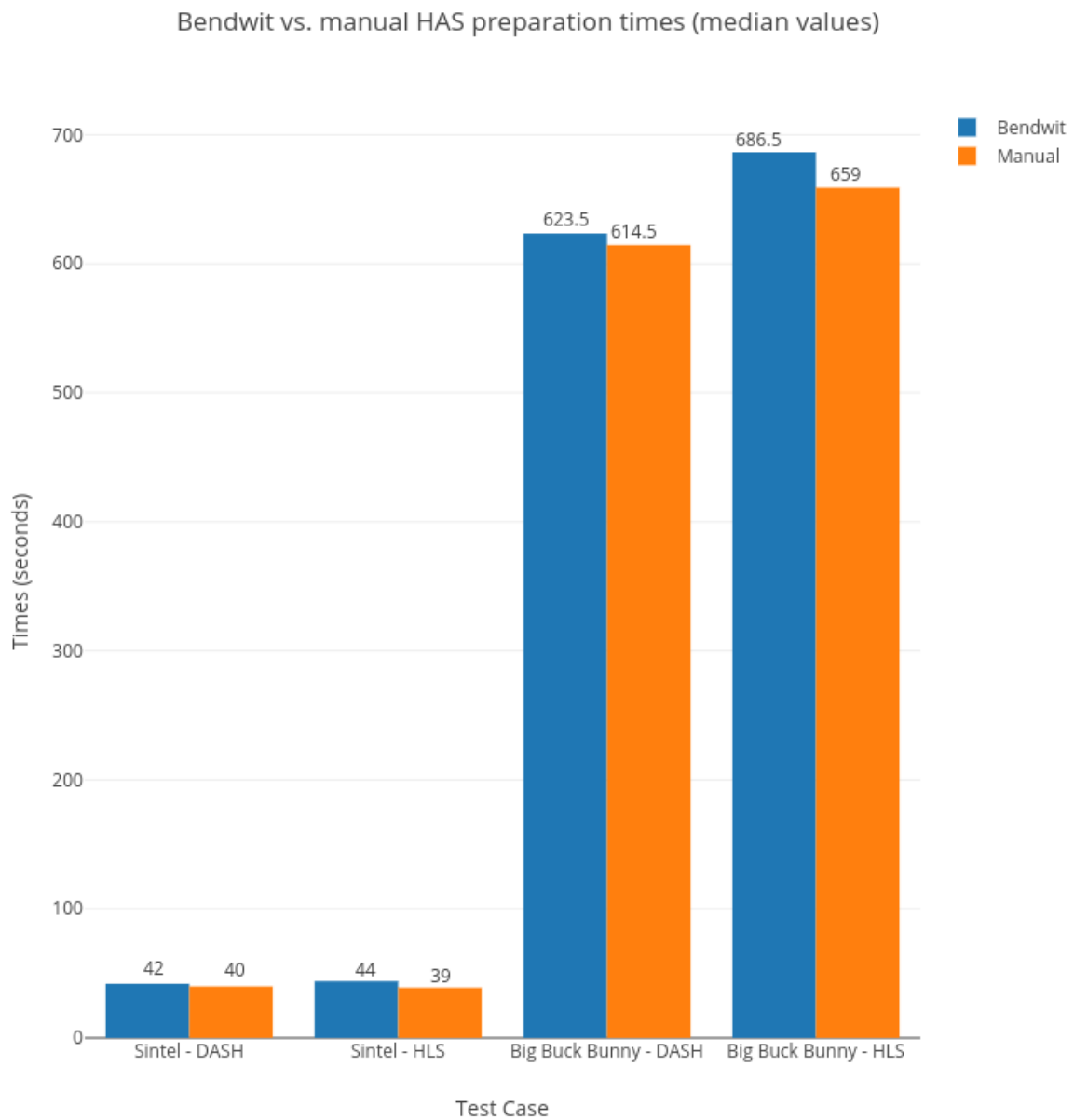


Figure 5.3: Bendwit HAS preparation versus manual HAS preparation time comparison

The findings match the hypothesis in the sense that all test cases show a clear difference in HAS preparation time. This indicates that the Bendwit platform introduces some overhead as expected. This overhead stems from the intra and extra configuration conflict management and source artifact probing built into the Bendwit CLI tool as well as the communication between the user, the REST API, the job scheduler, the Bendwit CLI tool and the user supplied webhook.

Chapter 6

Conclusion

To conclude this thesis, this chapter will give a small recapitulation of all the principles and things learned from the perspective of the author. When I started out with this thesis, I did not know a single thing about the topic. I had seen the evolution from Flash to HTML 5 media players and noticed that client players started handling video playback in a different way, but never really focused on what happened behind the scenes. Little did I know, this would once become the subject of my bachelor thesis.

At the very beginning, I assumed to do some research into what adaptive streaming entails and then spend the bulk of my time implementing a platform that prepares media content for adaptive streaming. Obviously, I was very wrong with this assumption. In order to understand what adaptive streaming is, I had to go back and look into many other subjects like media container formats, codecs, streaming protocols, ... I quickly came to the conclusion that the world of media is a very broad one. That is why I picked out the more popular codecs that are used for streaming and examined those in detail. I wrote down my findings in Chapters 2 and 3. When writing these chapters, I felt I could go a lot more in depth into certain topics; the H264 codec for example is a very complex one with support for a lot of features which were not all described in this thesis. At certain moments during the research I felt lost in the abundance of information I found, which made me take a step back and work in a different fashion. Instead of researching every piece of information I found on media, I started working in a way that led me towards my end goal: creating a platform for preparing media content for adaptive streaming. This is the reason why all the information in Chapters 2 and 3 are primarily focused on adaptive streaming.

Once I had a basic understanding of adaptive streaming, I went on to trying different tools for manipulating and segmenting media. I quickly found the most commonly used tools FFmpeg, FFprobe and MP4Box which led to the creation of the Bendwit CLI tool. The CLI tool uses the previously mentioned tools to create different media qualities from a specified input media artifact and outputs a DASH or HLS manifest, depending on what the user requests. Instead of creating a HAS preparation platform in a monolithic codebase, I opted for a layered approach with separation of concerns in mind. This resulted in a *REST API* which sends requests to the *Bendwit CLI tool* via a *Job Scheduler*, Bendwit users interact with the Bendwit platform directly via the REST API or via the *Website* and/or *Developer API* interfaces. This way, future platform changes could be made to the Website, REST API or Job Scheduler without the need of touching the media preparation codebase in the Bendwit CLI tool. This choice also resulted in a standalone adaptive streaming preparation tool which can be used by expert users directly via a CLI without the overhead of using a complete platform via a REST API. All the different layers of which Bendwit consists are described in Chapter 4 along with how to use them and how they were designed.

When looking at the findings described in Chapter 5, I can conclude that the Bendwit platform performs well given the circumstances in which it was developed. Media encoding services typically require strong hardware setups capable of transcoding media at fast rates. This is something I did not have access to during the development, it did on the other hand force me to develop a platform which is capable of working on lower end devices such as consumer laptops, which on its own is a very fun challenge as resources such as processing power, memory and hardware codec support are limited.

Comparing Bendwit to the competitors described in Section 1.3, the Bendwit platform offers a similar style of preparing media for adaptive streaming. I deliberately did not look at how other platforms implemented their REST API's and developer API's at the beginning of this thesis and came with a very similar setup for Bendwit. The way Bendwit differentiates itself with the earlier described competitors, is with the novice user interface the website provides. Bendwit provides predefined quality sets which can be used easily by any Bendwit user to prepare their media for adaptive streaming. Other platforms do not provide this abstraction and rely on the fact that users need to have a technical know-how about media codecs and adaptive streaming implementations.

Reflecting back on the time spent on this thesis and its implementation, I can say it has been quite the adventure. One I can recommend to any bachelor student, knowing that they will gain a lot of knowledge and experience on the subject of adaptive streaming. I went in with zero knowledge and came out with an understanding of how digital media works, how adaptive streaming looks like these days, what codecs and options to use in which situation or why certain media files do not play on certain devices and so much more. It has truly been an eye opener to something most people think they understand or even take for granted. The bachelor thesis has even sparked my interest to look further into multimedia studies. The only regret I have, looking back, is that I did not have enough time to realise all the ideas I had concerning the implementation. I enjoyed my time programming the platform a lot, but like many programmers I was naive to the idea that no problems and bugs would arise during the development. If asked for advice concerning a bachelor thesis, I would say to not underestimate the total package, plan enough time on your research and write down every idea you have. Planning the whole setup is crucial and will give you insight in advance which you wouldn't have when jumping in head first. I planned every part of the Bendwit separately, which is something I regret in hindsight. If I had planned every element of Bendwit at the same time, I would have had a better overview of the workings and communication required to make the Bendwit layers work with each other. I also recommend newcomers to the topic of adaptive streaming to thoroughly research the topic, find out the whys and hows and play around with the HAS implementations themselves. Only by doing so will you discover how HAS truly works. The world of HAS is still in its infancy; many bugs exists in HAS client players, the segmentation tools still do not support everything, ... only by knowing how HAS works will you be able to reason and figure out why certain setups fail or why they work in the first place.

Chapter 7

Future work

As briefly mentioned in Chapter 6, due to time constraints, not every idea made the final implementation. As it currently stands, the Bendwit platform could use improvements which would make it more robust and faster. This chapter will shortly describe some improvements which would benefit Bendwit.

7.1 Containerized Nodes

Currently the job scheduler simply spawns dedicated Bendwit CLI instances for probing and transcoding media. This idea is ideal for a small scale solution, but does not scale that well. Instead of spawning dedicated Bendwit CLI instances, containerized nodes could be used. For example Docker¹ containers could contain everything needed to run Bendwit CLI instances, these could be made available on multiple servers which are controlled by the job scheduler on the master server. The idea can even be expanded by letting NGINX do the loadbalancing and spread the REST API requests over multiple job schedulers on different systems. The system would then be scalable and balance according to the user request load.

7.2 Bendwit CLI Performance Boost

During the development of the Bendwit CLI tool, an HP Probook 650 G1 laptop² was used. This laptop only has CPU video encoding and decoding support. Due to this constraint no dedicated hardware encoding and decoding support was built into the Bendwit CLI tool. If access to a powerful graphics card, like the Nvidia GeForce GTX 1080³, becomes available; support for hardware encoding and decoding configuration classes could be added to the Bendwit CLI tool, making it perform a lot faster.

Due to the same hardware constraints, the Bendwit CLI tool was created in such a way that it performs every transcode on a new FFmpeg instance instead using the FFmpeg multi-core ability of performing multiple transcodes at the same time. Effectively transcoding all qualities on a one-by-one basis. If access to dedicated transcoding hardware becomes available, support for multithreaded transcoding jobs could be added. Extending on that idea, a balancing system could be introduced that senses how much load a system can take and thus scale the multithreaded transcoding of the different qualities according to this.

¹<https://www.docker.com/>

²<http://store.hp.com/us/en/mdp/probook-650>

³<https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080/>

7.3 Interactive REST API

At the time of writing, the REST API works in a request and process fashion. For example, a user requests a HAS preparation by supplying all wanted information in one request to the REST API; the Bendwit platform then performs this preparation and the user polls regularly for a status update or waits on the optional webhook report. Instead of supplying all HAS preparation data in one request, this process could be made more interactive. A user could for example work in multiple steps: declare a new HAS preparation job, add some encodings to the job via multiple REST API calls and finally declare that the job configuration is done so that the Bendwit platform can prepare the media for adaptive streaming. During this process, the REST API could report faulty encoding configurations, conflicts, . . . to the user making it more interactive; in the current implementation, the REST API only reports problems after a full HAS preparation request has been submitted.

Appendices

Appendix A

JSON Schemas

A.1 Probes Schema

```
1 {
2   "type": "object",
3   "properties": {
4     "external_url": {
5       "type": "string"
6     },
7     "media_id": {
8       "type": "string"
9     },
10    "webhook": {
11      "type": "string"
12    }
13  },
14  "oneOf": [
15    {
16      "required": [
17        "external_url"
18      ]
19    },
20    {
21      "required": [
22        "media_id"
23      ]
24    }
25  ]
26 }
```

Listing A.1: `/probes/` JSON schema

A.2 HAS Transcodes Schema

```
1 {
2   "type": "object",
3   "properties": {
4     "external_url": {
5       "type": "string"
6     },
7   },
8 }
```

```

7      "media_id": {
8        "type": "string"
9      },
10     "webhook": {
11       "type": "string"
12     },
13     "type": {
14       "type": "string",
15       "enum": [
16         "dash",
17         "hls"
18       ]
19     },
20     "segment_length": {
21       "type": "integer",
22       "minimum": 1000
23     },
24     "encodings": {
25       "type": "object",
26       "properties": {
27         "h264": {
28           "type": "array",
29           "minItems": 1,
30           "items": {
31             "type": "object",
32             "properties": {
33               "source_index": {
34                 "type": "integer",
35                 "minimum": 0
36               },
37               "media_height": {
38                 "type": "integer",
39                 "minimum": 0
40               },
41               "media_width": {
42                 "type": "integer",
43                 "minimum": 0
44               },
45               "kilobitrate": {
46                 "type": "integer",
47                 "minimum": 0
48               },
49               "aspectratio": {
50                 "type": "string"
51               },
52               "framerate": {
53                 "type": "integer",
54                 "minimum": 0
55               },
56               "h264_profile": {
57                 "type": "string",
58                 "enum": [
59                   "baseline",
60                   "main",
61                   "high",
62                   "high10",
63                   "high422",
64                   "high444"

```



```

65         ]
66     },
67     "h264_level": {
68         "type": "string",
69         "enum": [
70             "1",
71             "1b",
72             "1.1",
73             "1.2",
74             "1.3",
75             "2",
76             "2.1",
77             "2.2",
78             "3",
79             "3.1",
80             "3.2",
81             "4",
82             "4.1",
83             "4.2",
84             "5",
85             "5.1",
86             "5.2"
87         ]
88     }
89 },
90 "required": [
91     "source_index",
92     "framerate"
93 ],
94 "dependencies": {
95     "h264_profile": "h264_level",
96     "h264_level": "h264_profile"
97 }
98 }
99 },
100 "aac": {
101     "type": "array",
102     "minItems": 1,
103     "items": {
104         "type": "object",
105         "properties": {
106             "source_index": {
107                 "type": "integer",
108                 "minimum": 0
109             },
110             "channel_layout": {
111                 "type": "string",
112                 "enum": [
113                     "mono",
114                     "stereo",
115                     "2.1",
116                     "5.1",
117                     "7.1"
118                 ]
119             },
120             "kilobitrate": {
121                 "type": "integer",
122                 "minimum": 0

```

```

123         }
124     },
125     "required": [
126         "source_index"
127     ]
128 }
129 },
130 "webvtt": {
131     "type": "array",
132     "minItems": 1,
133     "items": {
134         "type": "object",
135         "properties": {
136             "source_index": {
137                 "type": "integer",
138                 "minimum": 0
139             }
140         },
141         "required": [
142             "source_index"
143         ]
144     }
145 }
146 }
147 }
148 },
149 "allOf": [
150     {
151         "oneOf": [
152             {
153                 "required": [
154                     "external_url"
155                 ]
156             },
157             {
158                 "required": [
159                     "media_id"
160                 ]
161             }
162         ]
163     },
164     {
165         "required": [
166             "type",
167             "segment_length",
168             "encodings"
169         ]
170     }
171 ]
172 }

```

Listing A.2: `/has_transcodes/` JSON schema

The “dependencies” indicate that when one property is specified, the other one must also be specified. Which in this case enforces that H264 profile and level will both be specified.

A.3 Media Schema

```
1 {
2   "type": "object",
3   "properties": {
4     "external_url": {
5       "type": "string"
6     },
7     "webhook": {
8       "type": "string"
9     }
10  },
11  "required": [
12    "external_url"
13  ]
14 }
```

Listing A.3: `/media/` JSON schema

Appendix B

Evaluation Configurations

B.1 Sintel Open Movie REST API Configuration

```
1 {
2   "media_id":"bea6827dcb504f8daff85d3fa9c5061a",
3   "type":"dash",
4   "segment_length":3000,
5   "webhook":"http://nethelix.org/bendwit/timestamp_webhook.php",
6   "encodings":{
7     "h264":[
8       {
9         "source_index":0,
10        "media_height":360,
11        "media_width":640,
12        "h264_profile":"baseline",
13        "h264_level":"3.1",
14        "kilobitrate":500,
15        "framerate":24
16      },
17      {
18        "source_index":0,
19        "media_height":720,
20        "media_width":1280,
21        "h264_profile":"baseline",
22        "h264_level":"3.1",
23        "kilobitrate":2400,
24        "framerate":24
25      },
26      {
27        "source_index":0,
28        "media_height":1080,
29        "media_width":1920,
30        "h264_profile":"high",
31        "h264_level":"4.1",
32        "kilobitrate":6000,
33        "framerate":24
34      }
35    ],
36    "aac":[
37      {
38        "source_index":1,
39        "channel_layout":"stereo",
```

```

40         "kilobitrate":36
41     },
42     {
43         "source_index":1,
44         "channel_layout":"stereo",
45         "kilobitrate":56
46     }
47 ]
48 }
49 }

```

Listing B.1: The Sintel open movie project DASH REST configuration

```

1 {
2     "media_id":"bea6827dcb504f8daff85d3fa9c5061a",
3     "type":"hls",
4     "segment_length":3000,
5     "webhook":"http://nethelix.org/bendwit/timestamp_webhook.php",
6     "encodings":[
7         "h264":[
8             {
9                 "source_index":0,
10                "media_height":360,
11                "media_width":640,
12                "h264_profile":"baseline",
13                "h264_level":"3.1",
14                "kilobitrate":500,
15                "framerate":24
16            },
17            {
18                "source_index":0,
19                "media_height":720,
20                "media_width":1280,
21                "h264_profile":"baseline",
22                "h264_level":"3.1",
23                "kilobitrate":2400,
24                "framerate":24
25            },
26            {
27                "source_index":0,
28                "media_height":1080,
29                "media_width":1920,
30                "h264_profile":"high",
31                "h264_level":"4.1",
32                "kilobitrate":6000,
33                "framerate":24
34            }
35        ],
36        "aac":[
37            {
38                "source_index":1,
39                "channel_layout":"stereo",
40                "kilobitrate":36
41            },
42            {
43                "source_index":1,
44                "channel_layout":"stereo",
45                "kilobitrate":56
46            }
47        ]
48    }
49 }

```

```

47     ]
48   }
49 }

```

Listing B.2: The Sintel open movie project HLS REST configuration

B.2 Sintel Open Movie Manual Commands

```

1  ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
   vf scale=w=640:h=360 -pix_fmt yuv420p -r 24 -profile:v baseline -level 3.1 -
   x264opts keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -
   map_chapters -1 -pass 1 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a
   /2052327cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_2.mp4 -b:v 500k -
   maxrate 500k -bufsize 1000k -f mp4 /dev/null
2
3  ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
   vf scale=w=640:h=360 -pix_fmt yuv420p -r 24 -profile:v baseline -level 3.1 -
   x264opts keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -
   map_chapters -1 -pass 2 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a
   /2052327cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_2.mp4 -b:v 500k -
   maxrate 500k -bufsize 1000k /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
   cb2c24d9a9cf8ed5424b463c2/video_transcode_2.mp4
4
5  ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
   vf scale=w=1280:h=720 -pix_fmt yuv420p -r 24 -profile:v baseline -level 3.1 -
   x264opts keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -
   map_chapters -1 -pass 1 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a
   /2052327cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_3.mp4 -b:v 2400k -
   maxrate 2400k -bufsize 4800k -f mp4 /dev/null
6
7  ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
   vf scale=w=1280:h=720 -pix_fmt yuv420p -r 24 -profile:v baseline -level 3.1 -
   x264opts keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -
   map_chapters -1 -pass 2 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a
   /2052327cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_3.mp4 -b:v 2400k -
   maxrate 2400k -bufsize 4800k /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
   cb2c24d9a9cf8ed5424b463c2/video_transcode_3.mp4
8
9  ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
   vf scale=w=1920:h=1080 -pix_fmt yuv420p -r 24 -profile:v high -level 4.1 -x264opts
   keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -map_chapters
   -1 -pass 1 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
   cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_4.mp4 -b:v 6000k -maxrate 6000k
   -bufsize 12000k -f mp4 /dev/null
10
11 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
   vf scale=w=1920:h=1080 -pix_fmt yuv420p -r 24 -profile:v high -level 4.1 -x264opts
   keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -map_chapters
   -1 -pass 2 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
   cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_4.mp4 -b:v 6000k -maxrate 6000k
   -bufsize 12000k /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
   cb2c24d9a9cf8ed5424b463c2/video_transcode_4.mp4
12
13 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -map_chapters
   -1 -vn -dn -c:a aac -b:a 36k -ac 2 /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
   cb2c24d9a9cf8ed5424b463c2/audio_transcode_0.mp4
14

```

```

15 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -map_chapters
    -1 -vn -dn -c:a aac -b:a 56k -ac 2 /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
    cb2c24d9a9cf8ed5424b463c2/audio_transcode_1.mp4
16
17 MP4Box -dash 3000 -profile dashavc264:live -url-template -bs-switching no -rap -frag-
    rap -out /media/bea6827dcb504f8daff85d3fa9c5061a/2052327cb2c24d9a9cf8ed5424b463c2/
    manifest -segment-name $RepresentationID$/segment_$Number$ /media/
    bea6827dcb504f8daff85d3fa9c5061a/2052327cb2c24d9a9cf8ed5424b463c2/
    audio_transcode_0.mp4#audio /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
    cb2c24d9a9cf8ed5424b463c2/audio_transcode_1.mp4#audio /media/
    bea6827dcb504f8daff85d3fa9c5061a/2052327cb2c24d9a9cf8ed5424b463c2/
    video_transcode_2.mp4#video /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
    cb2c24d9a9cf8ed5424b463c2/video_transcode_4.mp4#video /media/
    bea6827dcb504f8daff85d3fa9c5061a/2052327cb2c24d9a9cf8ed5424b463c2/
    video_transcode_3.mp4#video

```

Listing B.3: The Sintel open movie project manual MPEG-DASG HAS preparation commands

```

1 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
    vf scale=w=640:h=360 -pix_fmt yuv420p -r 24 -profile:v baseline -level 3.1 -
    x264opts keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -
    map_chapters -1 -pass 1 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a
    /2052327cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_2.mp4 -b:v 500k -
    maxrate 500k -bufsize 1000k -f mp4 /dev/null
2
3 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
    vf scale=w=640:h=360 -pix_fmt yuv420p -r 24 -profile:v baseline -level 3.1 -
    x264opts keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -
    map_chapters -1 -pass 2 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a
    /2052327cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_2.mp4 -b:v 500k -
    maxrate 500k -bufsize 1000k /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
    cb2c24d9a9cf8ed5424b463c2/video_transcode_2.mp4
4
5 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
    vf scale=w=1208:h=720 -pix_fmt yuv420p -r 24 -profile:v baseline -level 3.1 -
    x264opts keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -
    map_chapters -1 -pass 1 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a
    /2052327cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_3.mp4 -b:v 2400k -
    maxrate 2400k -bufsize 4800k -f mp4 /dev/null
6
7 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
    vf scale=w=1280:h=720 -pix_fmt yuv420p -r 24 -profile:v baseline -level 3.1 -
    x264opts keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -
    map_chapters -1 -pass 2 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a
    /2052327cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_3.mp4 -b:v 2400k -
    maxrate 2400k -bufsize 4800k /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
    cb2c24d9a9cf8ed5424b463c2/video_transcode_3.mp4
8
9 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
    vf scale=w=1920:h=1080 -pix_fmt yuv420p -r 24 -profile:v high -level 4.1 -x264opts
    keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -map_chapters
    -1 -pass 1 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
    cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_4.mp4 -b:v 6000k -maxrate 6000k
    -bufsize 12000k -f mp4 /dev/null
10
11 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -c:v libx264 -
    vf scale=w=1920:h=1080 -pix_fmt yuv420p -r 24 -profile:v high -level 4.1 -x264opts
    keyint=72:min-keyint=72:scenecut=-1 -an -sn -dn -preset ultrafast -map_chapters
    -1 -pass 2 -passlogfile /media/bea6827dcb504f8daff85d3fa9c5061a/2052327

```

```

cb2c24d9a9cf8ed5424b463c2/pass_log_video_transcode_4.mp4 -b:v 6000k -maxrate 6000k
-buFSIZE 12000k /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
cb2c24d9a9cf8ed5424b463c2/video_transcode_4.mp4
12
13 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -map_chapters
-1 -vn -dn -c:a aac -b:a 36k -ac 2 /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
cb2c24d9a9cf8ed5424b463c2/audio_transcode_0.mp4
14
15 ffmpeg -y -i /media/bea6827dcb504f8daff85d3fa9c5061a/raw_media.bendwit -map_chapters
-1 -vn -dn -c:a aac -b:a 56k -ac 2 /media/bea6827dcb504f8daff85d3fa9c5061a/2052327
cb2c24d9a9cf8ed5424b463c2/audio_transcode_1.mp4
16
17 ffmpeg -i /media/bea6827dcb504f8daff85d3fa9c5061a/84e31414510d4867abb8a8b2bff131cc/
audio_transcode_1.mp4 -codec copy -map 0 -bsf:v h264_mp4toannexb -f segment -g 72
-segment_time 3.000 -segment_list /media/bea6827dcb504f8daff85d3fa9c5061a/84
e31414510d4867abb8a8b2bff131cc/1/out.m3u8 /media/bea6827dcb504f8daff85d3fa9c5061a
/84e31414510d4867abb8a8b2bff131cc/1/segment%d.ts
18
19 ffmpeg -i /media/bea6827dcb504f8daff85d3fa9c5061a/84e31414510d4867abb8a8b2bff131cc/
audio_transcode_0.mp4 -codec copy -map 0 -bsf:v h264_mp4toannexb -f segment -g 72
-segment_time 3.000 -segment_list /media/bea6827dcb504f8daff85d3fa9c5061a/84
e31414510d4867abb8a8b2bff131cc/2/out.m3u8 /media/bea6827dcb504f8daff85d3fa9c5061a
/84e31414510d4867abb8a8b2bff131cc/2/segment%d.ts
20
21 ffmpeg -i /media/bea6827dcb504f8daff85d3fa9c5061a/84e31414510d4867abb8a8b2bff131cc/
video_transcode_2.mp4 -codec copy -map 0 -bsf:v h264_mp4toannexb -f segment -g 72
-segment_time 3.000 -segment_list /media/bea6827dcb504f8daff85d3fa9c5061a/84
e31414510d4867abb8a8b2bff131cc/3/out.m3u8 /media/bea6827dcb504f8daff85d3fa9c5061a
/84e31414510d4867abb8a8b2bff131cc/3/segment%d.ts
22
23 ffmpeg -i /media/bea6827dcb504f8daff85d3fa9c5061a/84e31414510d4867abb8a8b2bff131cc/
video_transcode_4.mp4 -codec copy -map 0 -bsf:v h264_mp4toannexb -f segment -g 72
-segment_time 3.000 -segment_list /media/bea6827dcb504f8daff85d3fa9c5061a/84
e31414510d4867abb8a8b2bff131cc/4/out.m3u8 /media/bea6827dcb504f8daff85d3fa9c5061a
/84e31414510d4867abb8a8b2bff131cc/4/segment%d.ts
24
25 ffmpeg -i /media/bea6827dcb504f8daff85d3fa9c5061a/84e31414510d4867abb8a8b2bff131cc/
video_transcode_3.mp4 -codec copy -map 0 -bsf:v h264_mp4toannexb -f segment -g 72
-segment_time 3.000 -segment_list /media/bea6827dcb504f8daff85d3fa9c5061a/84
e31414510d4867abb8a8b2bff131cc/5/out.m3u8 /media/bea6827dcb504f8daff85d3fa9c5061a
/84e31414510d4867abb8a8b2bff131cc/5/segment%d.ts

```

Listing B.4: The Sintel open movie project manual HLS HAS preparation commands

Bibliography

- [1] *White paper: Cisco VNI Forecast and Methodology, 2015-2020*. URL: <http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html> (visited on 05/04/2017).
- [2] *State of the internet*. URL: <https://www.akamai.com/us/en/about/our-thinking/state-of-the-internet-report/> (visited on 04/28/2017).
- [3] Greg Sterling. *Report: Mobile Search Queries 29 Percent Of Total But Growth Modest*. URL: <http://searchengineland.com/report-mobile-search-queries-29-percent-of-total-but-growth-modest-217501> (visited on 04/28/2017).
- [4] *Supported Encoding for Formats*. URL: <https://bitmovin.com/encoding-documentation/supported-formats-encoding/> (visited on 08/14/2017).
- [5] *Bitmovin Video Player*. URL: <https://bitmovin.com/video-player/> (visited on 08/14/2017).
- [6] *API Reference*. URL: <http://coconut.co/docs/api/config/variables> (visited on 08/14/2017).
- [7] *MPEG-DASH*. URL: <http://coconut.co/docs/api/config/dash> (visited on 08/14/2017).
- [8] *HTTP Live Streaming*. URL: <http://coconut.co/docs/api/config/hls> (visited on 08/14/2017).
- [9] *RTP: A Transport Protocol for Real-Time Applications*. URL: <https://tools.ietf.org/html/rfc3550> (visited on 04/28/2017).
- [10] Rafael Osso. *Handbook of Emerging Communications Technologies: The Next Decade*. 1 september 1999. URL: <https://books.google.be/books?id=5fms2DW7mMUC&pg=PA42>.
- [11] KyoungSoo Park Sangwook Bae Dahyun Jang. “Why Is HTTP Adaptive Streaming So Hard?” In: (2015). URL: <https://www.ndsl.kaist.edu/~kyoungsoo/papers/apsys2015.pdf>.
- [12] Iain E. Richardson. *Video Codec Design: Developing Image and Video Compression Systems*. 2002. ISBN: 978-0-471-48553-7.
- [13] *NVIDIA CUDA Video Decoder*. URL: <http://docs.nvidia.com/cuda/video-decoder/index.html> (visited on 04/28/2017).
- [14] *AMD Unified Video Decoder (UVD)*. URL: https://www.amd.com/Documents/UVD3_whitepaper.pdf (visited on 04/28/2017).
- [15] *State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs)*. URL: <https://tools.ietf.org/html/rfc5128#section-3.3> (visited on 04/28/2017).
- [16] *Hypertext Transfer Protocol – HTTP/1.1*. URL: <https://tools.ietf.org/html/rfc2616#section-1.4> (visited on 04/28/2017).
- [17] *HTTP over TLS*. URL: <https://tools.ietf.org/html/rfc2818#section-2.3> (visited on 04/28/2017).
- [18] *Streaming vs. Progressive Download*. URL: https://www.unique-media.tv/support/28/Introduction/Streaming_vs_Progressive_Download (visited on 04/29/2017).
- [19] *MPEG — The Moving Picture Experts Group website*. URL: <http://mpeg.chiariglione.org/> (visited on 04/29/2017).
- [20] *Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats*. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/c065274_ISO_IEC_23009-1_2014.zip (visited on 04/30/2017).

- [21] Stefan Lederer. *Why YouTube & Netflix use MPEG-DASH in HTML5*. URL: <https://bitmovin.com/status-mpeg-dash-today-youtube-netflix-use-html5-beyond/> (visited on 04/29/2017).
- [22] *DASH-IF Interoperability Points*. URL: <http://dashif.org/wp-content/uploads/2016/12/DASH-IF-IOP-v4.0-clean.pdf> (visited on 04/29/2017).
- [23] *Microsoft Smooth Streaming*. URL: <https://www.iis.net/downloads/microsoft/smooth-streaming> (visited on 04/30/2017).
- [24] *Adobe HTTP Dynamic Streaming Specification*. URL: <http://www.images.adobe.com/content/dam/Adobe/en/devnet/hds/pdfs/adobe-hds-specification.pdf> (visited on 04/30/2017).
- [25] *MPEG-4 Advanced Video Coding*. URL: <http://mpeg.chiariglione.org/standards/mpeg-4/advanced-video-coding> (visited on 04/29/2017).
- [26] Christopher Müller and Christian Timmerer. "A VLC Media Player Plugin enabling Dynamic Adaptive Streaming over HTTP". In: (2011). URL: <http://www-itec.uni-klu.ac.at/bib/files/p723-muller.pdf>.
- [27] *HTTP Live Streaming*. URL: <https://developer.apple.com/streaming/> (visited on 04/30/2017).
- [28] *HTTP Live Streaming draft-pantos-http-live-streaming-22*. URL: <https://tools.ietf.org/html/draft-pantos-http-live-streaming-22> (visited on 04/30/2017).
- [29] *Best Practices for Creating and Deploying HTTP Live Streaming Media for Apple Devices*. URL: https://developer.apple.com/library/content/technotes/tn2224/_index.html (visited on 04/30/2017).
- [30] *IIS Smooth Streaming Technical Overview*. URL: http://www.bogotobogo.com/VideoStreaming/Files/iis8/IIS_Smooth_Streaming_Technical_Overview.pdf (visited on 08/02/2017).
- [31] *Smooth Streaming*. URL: <https://www.iis.net/downloads/microsoft/smooth-streaming> (visited on 08/02/2017).
- [32] *IIS Smooth Streaming Client Manifest Format*. URL: [https://msdn.microsoft.com/en-us/library/ee673436\(v=vs.90\).aspx](https://msdn.microsoft.com/en-us/library/ee673436(v=vs.90).aspx) (visited on 08/02/2017).
- [33] *Adobe HDS Basic FAQs*. URL: <http://www.adobe.com/products/hds-dynamic-streaming/faq.html> (visited on 08/03/2017).
- [34] *Adobe Flash Video File Format Specification Version 10.1*. URL: http://download.macromedia.com/f4v/video_file_format_spec_v10_1.pdf (visited on 08/03/2017).
- [35] *HTTP Dynamic Streaming - An Overview*. URL: <https://www.encoding.com/http-dynamic-streaming-hds/> (visited on 08/03/2017).
- [36] Anthony T. S. Ho; Shujun Li. *Handbook of Digital Forensics of Multimedia Data and Devices*. John Wiley & Sons, 2015. ISBN: 9781118757079. URL: <https://books.google.be/books?id=pDUODAAAQBAJ&pg=PT146#v=onepage&q&f=false> (visited on 07/31/2017).
- [37] *What is Matroska?* URL: <https://www.matroska.org/technical/whatis/index.html> (visited on 07/31/2017).
- [38] *FFmpeg naming and logo*. URL: <http://ffmpeg.org/pipermail/ffmpeg-devel/2006-February/010315.html> (visited on 05/01/2017).
- [39] *FFmpeg's future and resigning as leader*. URL: <http://ffmpeg.org/pipermail/ffmpeg-devel/2015-July/176489.html> (visited on 05/01/2017).
- [40] *The FFmpeg/Libav situation*. URL: <http://blog.pkh.me/p/13-the-ffmpeg-libav-situation.html> (visited on 07/31/2017).
- [41] *Debate libav-provider ffmpeg*. URL: <https://wiki.debian.org/Debate/libav-provider/ffmpeg> (visited on 05/01/2017).
- [42] *Supported File Formats, Codecs or Features*. URL: http://ffmpeg.org/general.html#Supported-File-Formats_002c-Codecs-or-Features (visited on 05/01/2017).
- [43] *ffmpeg Documentation*. URL: <http://ffmpeg.org/ffmpeg.html> (visited on 05/01/2017).
- [44] *Instructions to playback Adaptive WebM using DASH*. URL: <http://wiki.webmproject.org/adaptive-streaming/instructions-to-playback-adaptive-webm-using-dash> (visited on 05/01/2017).
- [45] *ffprobe Documentation*. URL: <https://ffmpeg.org/ffprobe.html> (visited on 05/01/2017).

- [46] *ffprobe Documentation - Writers*. URL: <https://ffmpeg.org/ffprobe.html#Writers> (visited on 05/01/2017).
- [47] *GPAC - About us*. URL: <https://gpac.wp.imt.fr/home/about/> (visited on 05/01/2017).
- [48] *MP4Box*. URL: <https://gpac.wp.imt.fr/mp4box/> (visited on 05/01/2017).
- [49] *GPAC sourceforge repository*. URL: <https://sourceforge.net/p/gpac/code/2147/> (visited on 05/01/2017).
- [50] *AVC/H.264 FAQ*. URL: <http://www.mpegla.com/main/programs/AVC/Pages/FAQ.aspx> (visited on 07/31/2017).
- [51] *H.264 answers Google's open codec with forever free license*. URL: https://www.theregister.co.uk/2010/08/26/mpegla_v_google/ (visited on 07/31/2017).
- [52] *H.264 Video Codec Adopted for Next Generation DVDs*. URL: <https://www.apple.com/newsroom/2004/06/23H-264-Video-Codec-Adopted-for-Next-Generation-DVDs/> (visited on 07/31/2017).
- [53] *Timeline: Apple milestones and product launches*. URL: <https://www.reuters.com/article/us-apple-timeline-idUSTRE72170T20110303> (visited on 08/01/2017).
- [54] *Advanced video coding for generic audiovisual services*. URL: https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-H.264-201610-S!PDF-E&type=items (visited on 08/01/2017).
- [55] *Reduce Bandwidth Consumption by GOP Settings*. URL: http://www2.acti.com/support_old/Package/%7B6060C79F-2A5D-40A4-8837-16B835E3364.PDF (visited on 08/01/2017).
- [56] *B-Frames*. URL: <http://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node258.html> (visited on 08/01/2017).
- [57] *WebM Bitstream Specification License*. URL: <https://www.webmproject.org/license/bitstream/> (visited on 08/02/2017).
- [58] Fabio Sonnati. *Thoughts on VP8*. URL: <https://sonnati.wordpress.com/tag/vp8/> (visited on 08/02/2017).
- [59] *Inside WebM Technology: The VP8 Alternate Reference Frame*. URL: <http://blog.webmproject.org/2010/05/inside-webm-technology-vp8-alternate.html> (visited on 08/02/2017).
- [60] *RTP Payload Format for VP8 Video draft-ietf-payload-vp8-17*. URL: <https://tools.ietf.org/html/draft-ietf-payload-vp8-17> (visited on 08/02/2017).
- [61] *VP8 Data Format and Decoding Guide*. URL: <https://tools.ietf.org/html/rfc6386> (visited on 08/02/2017).
- [62] Jan Ozer. *First Look: H.264 and VP8 Compared*. URL: <http://www.streamingmedia.com/Articles/Editorial/Featured-Articles/First-Look-H.264-and-VP8-Compared-67266.aspx> (visited on 08/02/2017).
- [63] *AAC License Fees*. URL: <http://www.via-corp.com/us/en/licensing/aac/licensefees.html> (visited on 08/31/2017).
- [64] Karlheinz Brandenburg. "MP3 and AAC explained". In: (1999). URL: https://www.iis.fraunhofer.de/content/dam/iis/de/doc/ame/conference/AES-17-Conference_mp3-and-AAC-explained_AES17.pdf.
- [65] *ISO/IEC 13818-7:2006*. URL: <https://www.iso.org/standard/43345.html> (visited on 08/01/2017).
- [66] *Encoding specifications for music videos*. URL: <https://support.google.com/youtube/answer/6039860?hl=en> (visited on 08/01/2017).
- [67] *EBU-TTD support in GPAC*. URL: <https://gpac.wp.imt.fr/2014/08/23/ebu-ttd-support-in-gpac/> (visited on 07/30/2017).
- [68] *Same-origin policy*. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy (visited on 08/14/2017).
- [69] Claire Lauer. "Contending with Terms: "Multimodal" and "Multimedia" in the Academic and Public Spheres". In: (). URL: <https://web.archive.org/web/20140327001010/http://dmp.osu.edu:80/dmac/supmaterials/lauer.pdf>.
- [70] *DASH Support in MP4Box*. URL: <https://gpac.wp.imt.fr/mp4box/dash/> (visited on 05/01/2017).
- [71] *JCT-VC - Joint Collaborative Team on Video Coding*. URL: <http://www.itu.int/en/ITU-T/studygroups/2017-2020/16/Pages/video/jctvc.aspx> (visited on 07/31/2017).

- [72] Jay Yarrow. *Google Bullied Out Of Another \$26.5 Million In Cash By On2 Shareholders*. URL: <http://www.businessinsider.com/google-adds-cash-to-its-on2-acquisition-to-close-the-deal-2010-1?IR=T> (visited on 08/02/2017).