Unlimited seeking during video transcoding

Fredrik Grape
Abstract

Unlimited seeking during video transcoding

Fredrik Grape

High definition, long duration videos can contain complex scenes full of rich detail. As the details get richer and the duration grows, the cost of transcoding invariably increases. For Swedish law enforcement, a greatly detailed video can prove vital in aiding the investigation of a crime, by clearly identifying objects, or the face of a perpetrator or victim. Modern day surveillance cameras are capable of capturing videos with high frame rate and great resolution, but the interesting scenes usually make up only a small fraction of the recorded material. Finding those scenes quickly means being able to search through the entire video unhindered.

To aid in this process, a novel web application was developed during this thesis that allows users to browse videos on a server, load them into a video player and seek across the entire video while it is being transcoded, regardless of the original video format. Experimental results identify the CPU as the primary bottleneck, and a preferable FFmpeg transcoding preset (called veryfast) from a set of candidate presets. Regarding usability, the application runs mostly without issues, but long buffering times will sometimes plague the user experience. Yet, this thesis provides an important step towards achieving seamless and simultaneous seeking and transcoding.
## Contents

1 Introduction .................................................. 5

2 Background ...................................................... 6

2.1 Digital videos ................................................. 6

2.2 Videos within Swedish law enforcement ..................... 7

2.3 Requirements .................................................. 8

2.4 Video transcoding software .................................... 9

2.4.1 FFmpeg ....................................................... 9

2.4.2 A word on FFmpeg wrappers .......................... 10

2.4.3 JCodec ....................................................... 10

2.5 HLS ............................................................. 11

2.5.1 Creating HLS files with FFmpeg ....................... 11

2.5.2 hls.js ......................................................... 13

2.6 HTML5 video and MSE ........................................ 13

2.7 Alternative streaming protocols ............................. 13

2.7.1 MPEG-DASH ................................................ 14

2.7.2 Smooth Streaming ....................................... 15

2.7.3 HTTP Dynamic Streaming ............................. 15

3 Method ............................................................ 16

3.1 System overview .............................................. 16
1 Introduction

Videos are used by the Police on a regular basis. They can be captured by officers in the field to detail the events of a crime scene, or they can be sent to them from elsewhere to aid in the investigation of a crime. The videos they deal with can come from basically anywhere, captured by devices ranging from cellphones and body cameras, to high definition around-the-clock surveillance cameras.

The current workflow regarding videos within Swedish law enforcement harkens back to an older, less centralized era of technology. Videos are handled on an individual basis, with people having to burn CD-ROMs or carry USB drives, and find a way themselves to convert videos to a format that can be played on their machines.

A more streamlined approach would be preferable, such as one that allows law enforcement investigators to use their existing web applications to access, view and work with videos.

To address this issue, this thesis presents a web application that lets users browse the available videos on the server and view them on-demand. The application automatically transcodes videos to a compatible format before playback, and divides them into segments that can be sent to the client one-by-one. In this manner, playback can begin almost immediately as the user presses play, regardless of the original video format.

The application consists of two versions, called Stable and SNAPSHOT. The Stable version provides a comfortable viewing experience, free from buffering or errors, and lets users freely play, pause and seek up until the highest number of transcoded segments. This version is best suited for shorter duration videos (one minute or less) which can be transcoded quickly.

For longer duration videos, such as those captured by surveillance cameras recording non-stop, transcoding times will quickly grow. For these types of videos, investigators will find themselves waiting a long time before they are able to browse the entire video. This is where the SNAPSHOT version comes in. It allows the user to seek to any part of the video, even if that part has not
been transcoded yet. Doing so will immediately start another transcoding session starting from the current time the user seeked to. A detailed description of both versions can be found in the Section 3.

Experimental results can be found in Section 4 showing CPU and RAM usage while running the application, and a comparison of how bitrates and transcoding times vary with different FFmpeg presets. The results show that the CPU is the primary bottleneck, with FFmpeg using roughly 80% of CPU power during transcoding, but only 3% RAM. Out of the presets considered (see Section 4.3) the veryfast preset consistently produced the lowest bitrate of videos after transcoding, and the transcoding time using this preset was at worst merely 24% of the duration of the original video. Using the veryfast preset came at no discernible cost of image quality to the transcoded video, making it the preferable preset.

A usability analysis of both versions in Section 4 reveals that while the Stable version works without issues, an entirely satisfying user experience could not be delivered with the SNAPSHOT version. Users may load a video and seek forward to any part of it, but they do so at the cost of hefty buffering times. This is followed by a discussion in Section 5 and conclusions in Section 7.

2 Background

2.1 Digital videos

Digital videos often come in container formats (e.g. MP4), encapsulating video and audio streams, subtitles, headers and metadata. There are many different container formats and any given video player only supports a subset of these. On top of this, the video streams themselves within the container may all have different encodings, which require the right codec to decode and encode.

To solve this issue, software has been developed for transcoding videos, which are capable of unwrapping, decoding, compressing, encoding and wrapping video files with different encodings and formats. This is the usual process of converting a video from one format to another. Unfortunately, similarly to
video players, any given transcoder only supports a subset of all encodings and formats - a problem which is only exacerbated by the existence of homemade encoders and formats that require specific tools to work with [2]. An investigation into some of the available software can be found in Section 2.3.

Another limitation with digital videos is their size, which affect both the cost of storage and the time it takes to transcode them. Methods to compress video files range from trivial approaches, such as lowering resolution or the amount of frames in the video, to more sophisticated ones like filtering out certain wavelengths [3] and comparing how pixels change from frame to frame [4, p. 157].

2.2 Videos within Swedish law enforcement

Evidentiary videos (i.e., videos used as evidence) are innately the same as any other, but they have to be given special treatment. A video received or created by the police is meant to aid the investigation of a crime, and should therefore be stored without being modified. The reason for this is twofold:

1. To ensure its validity as evidence.

2. To maintain all the original information in the file for conservation.

One better not risk invalidating evidence by tampering with the original file. That being said, videos used in investigations have to be stored for conservation, and the Swedish national archives requires, among other things, videos to be of a specific file format [5]. This does mean that at some point the video has to be transcoded, but only after an investigation is finished.

This becomes problematic when designing a system for working with videos through the browsers currently used by the Swedish police, since they have to be of the right file format to be viewed. One obvious solution is to create copies of the original files and convert them to the right format, storing both files. That way the original file is kept intact.

The issue with such an approach is the cost of storing all these extra
files, especially since videos can be very large. According to Seagate a "high quality" surveillance video with MPEG-4 encoding at 30 frames per second requires 1TB of storage per 8 days of continuous recording [6], and the high quality they list is nowhere near the 4k resolution some of the Swedish surveillance cameras produce.

### 2.3 Requirements

Before building new tools, an investigation was done into what tools for transcoding videos are currently available. Based on this investigation, FFmpeg [7] was chosen to work with.

The investigation was done using a list of criteria, that was produced to lay out in detail what the transcoding tool should fulfill. In particular, the main criteria identified were:

1. It is open source.
2. It is being actively maintained by the developers.
3. It is well documented.
4. It supports the HTTP Live Streaming (HLS) protocol.
5. It supports an H.264 codec.
6. It has a generally good reputation.

H.264 was chosen because it is as close to a universal codec standard as can be found, with 81% of all encodings being done using H.264 in 2017 [8], showing an increase in usage from 72% in 2015 [9].

The reason for choosing HLS is similar. 74% of premium video publishers were using HLS in 2017 [8], showing a small increase in usage from 71% in 2015 [9].

Criteria 1 to 5 were investigated by searching through the web page of the respective tool.
Criteria 6 was based on a subjective analysis of discussions on Stackoverflow [10] and the perceived popularity of the tool in articles, books and as observed on the Internet.

2.4 Video transcoding software

2.4.1 FFmpeg

FFmpeg was initially released towards the end of 2000 [11] and discussions about how to use it for various problems date back several years. Multiple new questions come in daily and discussions are healthy [12]. It is heavily referenced in "Video Encoding by the Numbers", a book used extensively as a video encoding guide throughout this project.

It is published under the GNU Lesser General Public License (LGPL) version 2.1 or later [13], receives frequent updates [14] and has extensive documentation [15].

It is highly user friendly, requiring at its simplest just a short command line to transcode a video from one format to another. For example, to transcode a video input.mov of format MOV to output.mp4 of format MP4, one can simply use the following command:

```
ffmpeg -i input.mov output.mp4
```

It also allows users to easily change settings by adding switches to the command line. For example, one can tell FFmpeg to explicitly use its instance of an H.264 codec (x264), and to use a slower setting that provides better compression (quality per filesize) [16] at the cost of transcoding speed, by using the following command:

```
ffmpeg -i input.mov -c:v libx264 -preset slow output.mp4
```

FFmpeg also makes it easy to output HLS files, by including switches such as:

```
-f HLS -hls_time 6 -hls_list_size 0
```
which, if included with a few other switches, will output an HLS video with a segment length of 6 seconds. The switches used to encode to HLS will be further explained in section 2.5.1.

Lastly, since it is possible to transcode videos with FFmpeg using just one command line, it is also possible to build a simple PHP encoder using the function `shell_exec()` [17].

### 2.4.2 A word on FFmpeg wrappers

Numerous wrappers around FFmpeg exist that provide APIs in various languages, such as Video Converter [18] (Python), Humble Video [19] (Java) and Handbrake.js [20] (JavaScript over node.js). Using a wrapper was initially thought to be more user friendly than working with the pure command line version of FFmpeg, but that idea was abandoned after discovering how easy the command line version was to work with.

### 2.4.3 JCodec

JCodec is a pure java implementation of video and audio codecs. It is free software distributed under FreeBSD license. The earliest trace of JCodec was found when the project moved to GitHub in 2012. Updates come in a few times per year, the latest being Aug 29, 2018.

Being written in Java, decoding with JCodec will typically be an order of magnitude slower than native implementations like FFmpeg. The encoded quality and bitrate for the H.264 encoder can also be expected to be much worse than other known encoders like x264, due to little work having been done in this area. However, decode quality is allegedly at industry level.

There is virtually no documentation, and the developers of JCodec refer to Stackoverflow for information. Among the supported formats is MPEG TS, and since JCodec provides an H.264 encoder, it might be possible to encode to HLS [21].
Discussion are sparse with about a dozen questions coming in per year, and they gain little traction. Furthermore, JCodec seems to be mainly used in Android devices. That being said, discussions regarding JCodec seem healthy [22].

2.5 HLS

HLS is an HTTP-based media streaming communications protocol, developed by Apple [23]. It works by sending videos in smaller segments to the client, which means a user can start viewing the video as soon as the first segment has been downloaded, rather than having to wait for the entire video to download first. HLS is utilizes adaptive bitrate (ABR) streaming [24], which is a method for encoding single videos into multiple streams of varying qualities, that a client can switch between seamlessly depending on the bandwidth available. Alternative ABR streaming technologies will be discussed in section 2.7.

HLS videos can consist of a single file or multiple individual files. Accompanying the video files is a playlist containing information about the length and number of the video segments and where to find them. When working with individual files, the playlist lists the length of each segment in seconds. When working with a single file, the playlist lists the length of each segment in both seconds and byte ranges. The video file extensions used by HLS is either MPEG-2 Transport Stream (TS) or fragmented MP4 (fMP4).

2.5.1 Creating HLS files with FFmpeg

FFmpeg has built in support for encoding to HLS [25]. Outputting a single file, controlling only the segment length, can look something like this:

```
ffmpeg -i aerial.mov -f HLS -g 60 -keyint_min 60 -sc_threshold 0 -hls_time 2 -hls_list_size 0 -hls_flags single_file aerial.m3u8
```
Below are short descriptions of the switches used:

-i aerial.mov  the input video.

-f HLS  specifies that the output format should be HLS.

-g 60  the number of frames in one group of pictures.

-keyint_min 60  sets the minimum space between each key frame.

-sc_threshold 0  tells FFmpeg to not detect scene changes.

-hls_time 2  sets the segment length to 2 seconds.

-hls_list_size 0  when set to 0, all segments will be contained in a single playlist.

-hls_flags single_file  output should be a single file using byte ranges.

aerial.m3u8  the name of the playlist.

Some of these switches require further explanation:

-g 60
This particular video (aerial) has a frame rate of 30 frames per second. This switch sets group of pictures (GOP) to 60, meaning each GOP will contain 60 frames and hence be 2 seconds long. A GOP is a sequence of frames that start with a key frame and includes all frames up to, but not including, the next key frame [4, p. 157]. This sets a lower bound of 2 seconds on the distance between each key frame. When encoding to HLS, key frames must divide evenly into each segment [4, p. 173].

-keyint_min 60
Sets the minimum distance between each key frame, in this case 60 frames or 2 seconds. Combined with the above switch, this forces FFmpeg to insert key frames at exactly every 2 seconds. This ensures key frames divide evenly into each segment since the segment length was chosen to be 2 seconds.

-sc_threshold 0
By default, FFmpeg is able to detect scene changes to dynamically insert key frames where they are considered to be most needed [4, p. 161]. Since
HLS requires key frames to divide evenly into each segment this option is set to 0, which disables it. Theoretically, setting `-g` and `-keyint_min` should be enough to guarantee this, but it is currently unclear if the scene detection option overrides them and it is hence explicitly disabled.

If these options are not set properly, FFmpeg will fail to output HLS files with the desired settings.

### 2.5.2 hls.js

hls.js is a JavaScript library which implements an HLS client. It works by re-wrapping TS files into MP4 fragments. It relies on HTML5 video and MSE for playback and does not need a player of its own as it works directly on top of the HTML5 `<video>` element [26]. hls.js was used to enable playback of the videos used in this project.

When playing back videos, hls.js looks at the HLS playlist to calculate the length of the video. It continually tracks any updates to the playlist.

### 2.6 HTML5 video and MSE

The HTML5 `<video>` element specifies a standard way to embed video in a web page, whereas before one had to use a plug-in such as Flash [27].

Media Source Extensions (MSE) is a specification for a JavaScript interface to play back media data within a browser. Browsers that support MSE can play chunks of videos which enables, among other things, video on-demand [4, p. 246].

### 2.7 Alternative streaming protocols

There exists a number of streaming protocols aside from HLS, and the three major ones will be discussed in this section. Though they may differ in some ways, they all fundamentally function the same. When encoding under any of
these protocols, the end result is a fragmented video divided into several segments, and a manifest file keeping track of them. The differences between them mainly lie in the structure and file extension of the manifest, the video format, video player and the target platform.

2.7.1 MPEG-DASH

MPEG-DASH was developed following a call from MPEG to agree on an HTTP streaming standard. There are several major backers of DASH, including Adobe, Samsung, Netflix and others. It is aimed at making ABR streaming support seamless on all devices [28].

In 2015, only 9% of Streaming Media readers were using DASH as their streaming technology, while 41% used HLS. However, by 2020 41% predicted they would be using DASH [29]. According to Encoding.com [30], 10% [9] of their customers were encoding to DASH in 2015, rising to 22% [8] in 2017.

DASH works much the same as HLS. Indeed, a DASH client can play videos with the same file extension as HLS (TS or fMP4) provided that the manifest can be correctly parsed.

A DASH manifest is called a Media Presentation Description (MPD) and is an XML document containing information about media segments [31]. It is more complex in structure than the HLS playlist, but basically functions the same. It points the video player to the segments that make up the video so they can be played back to back.

FFmpeg does not contain native support to encode to DASH, but since FFmpeg provides tools to build a custom made segmenter and is natively capable of outputting TS or fMP4 files, it might be possible to use FFmpeg to encode to DASH using a segmenter custom made for outputting an MPD along with the video segments.

Otherwise, tools like MP4Box [32] and Bento4 [33] can be used to fragment existing MP4 files into smaller segments for streaming with DASH. However, they both mostly work with ISO base media file formats [34].
2.7.2 Smooth Streaming

Microsoft’s Smooth Streaming was one of the first ABR streaming protocols to be developed.

A Smooth Streaming manifest is called Movie Metadata and is accompanied both by video fragments and an index file called Movie Fragment Random Access, which is what allows seeking within the video. The fragments themselves contain both metadata and the actual video content [35].

In 2015, only 8% of Streaming Media readers were using Smooth Streaming, and that number was predicted to fall to 6% by 2020. According to Encoding.com, 19% of their customers were encoding to Smooth Streaming in 2015, which they mainly attribute to the XBOX One evolving to more than a gaming console [9]. In 2017, that number dropped to 4% [8]. They predict the deprecation of Smooth Streaming and do not see any new workflow planning around it.

2.7.3 HTTP Dynamic Streaming

HTTP Dynamic Streaming (HDS) is Adobe’s take on ABR streaming for Flash video [36].

12% of Streaming Media readers were using HDS in 2015, and that number was predicted to drop to 9% in 2020. Encoding.com makes no mention of HDS in their reports, aside from stating in their 2016 report that Flash video is on life support [9]. In general, the overall consensus while reading about digital videos seems to be that Flash is becoming a thing of the past.
3 Method

3.1 System overview

For this thesis, a web application was developed that allows users to browse videos on a server and watch them in their browser. The application automatically calls the server to start transcoding the video that the user chooses to watch. As the video is transcoded, it is simultaneously segmented using the HLS segmenter provided by FFmpeg, allowing segments to be sent one-by-one to the client as soon as they are complete. The segments are received on the client side by hls.js, and the HTML5 video player takes care of playback and provides all the standard controls such as pause, play and seek. The application can be seen in its entirety in Figure 1.

3.2 Demo videos

Three demo videos were used during development. Thumbnails can be seen in Figure 2. They were deliberately chosen to vary in duration and complexity, and were all encoded using an H.264 codec. The videos are described below, and technical details about them can be found in Table 1.

- aerial.mov: A high resolution, short duration, slow moving video of the sky above the clouds.
- earth.mov: A medium duration animation of the earth spinning around its axis, with little frame-to-frame change and a lot of constant black space.
- fantasticbeasts.mp4: A longer duration trailer for the movie Fantastic Beasts and Where to Find Them (simply referred to as "Beasts" in this thesis), with lots of scene changes and motion.
3.3 Private server and web application

A private server was set up on a laptop\cite{37} using the operating system elementary OS 0.4 (Loki), which is a Linux distribution based on Ubuntu\cite{38}. It was served over HTTP using Nginx\cite{39}, and the web-page was built using HTML5 and JavaScript. PHP was used on the server side to build an encoder and utility scripts. The library hls.js was used to allow playback of HLS videos. Chrome was mainly used as the browser during development, but the application was also tested on Firefox.

Two versions of the application were developed, a simpler one called Stable that has some interactive capabilities during transcoding and a bleeding-edge one called Snapshot that allows users to seek forward in a video to parts that
Table 1: Technical details of the three demo videos.

<table>
<thead>
<tr>
<th>Video</th>
<th>Duration [mm:ss]</th>
<th>Resolution</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aerial</td>
<td>00:11.05</td>
<td>2560x1442</td>
<td>29.97</td>
</tr>
<tr>
<td>Earth</td>
<td>00:30.57</td>
<td>1920x1080</td>
<td>30</td>
</tr>
<tr>
<td>Beasts</td>
<td>02:36.12</td>
<td>1920x796</td>
<td>23.96</td>
</tr>
</tbody>
</table>

have not been transcoded yet. These are described next.

### 3.4 Stable version

The **Stable** version allows users to choose a video to view, via a dropdown menu that lists all available videos on the server. When a video is chosen to watch, a request is sent to the server to start transcoding the video, and playback can commence as soon as the first segment is finished and the playlist has been created. Initially, the playlist only contains information about the first segment, but as soon as the next segment is finished it is appended to the playlist.

The video will only appear to be as long as the playlist currently makes it out to be. For example, in Figure 1 the full video is actually around 11 seconds long, but only 8 seconds of the video at that time has been transcoded, so that is all the playlist shows and hence what the video player knows about. Since hls.js continually tracks updates to the playlist, the video will appear to be longer and longer as more segments are added to the playlist, until the full video has been successfully transcoded. Only then can users interactively seek across the full video.

The logical flow of the **Stable** version is as follows:

1. The user chooses a video to watch from a dropdown menu that lists all available videos on the server. The video is loaded by clicking the **Stable** button, which calls the function `playVideoSTABLE()`.
2. All transcoded videos are removed.
3. After 2 seconds, the function `convertVideoSTABLE()` is called.

4. A request is sent to a PHP script on the server, called `encoder_stable.php`, which calls FFmpeg with a set of predefined settings and holds the script until the first segment is complete and the playlist is created.

5. If the request returns OK, the url to the playlist is saved and the function `startPlayback()` is called. Otherwise, nothing happens.

6. Using hls.js, an HLS object is created and bound to the playlist url. When everything is ready, playback can start. The user can now watch and seek across the video up until the currently highest number of completed segments.

Next follows a description of each of the functions mentioned above,
one-by-one.

3.4.1 playVideoSTABLE()

Checks the value of the dropdown menu to find out which video the user wants to view. It then sends a request to the server to remove all transcoded files, and finally calls convertVideoSTABLE() after a delay of 2 seconds. The delay is there to give the server time to remove the transcoded files before anything else happens.

3.4.2 convertVideoSTABLE()

Sends a GET request for the script encoder_stable.php along with the name of the video. This is done using an XMLHttpRequest object, an asynchronous HTTP request and a function that is triggered on a "ready state change" event of the request. When the ready state is set to done and the status is OK, the url to the playlist is set and startPlayback() is called.
3.4.3 encoder_stable.php

Gets the name of the video to transcode from the client, and uses it to call FFmpeg to start the transcoding. The settings used are:

- **c:v libx264** to specify the H264 codec.
- **preset veryfast** for a fast but less efficient encoding.
- **s 640x480** for a constant resolution of 640x480.
- **r 30** for a constant framerate of 30.
- **g 60**
- **keyint_min 60**
- **sc_threshold 0**
- **f HLS**
- **hls_time 2**
- **hls_list_size 0**

Finally ">/dev/null 2>&1" is added at the end of the command line to write all FFmpeg output and error messages to an external file and allow the script to continue with the transcoding in the background.

The script is then put on hold until the first segment is complete and it can locate the playlist.

3.4.4 startPlayback()

Creates an Hls object, from the hls.js library and binds it to the HTML5 video element. The Hls object then tries to load the video from the source provided by the url set earlier. Playback will then automatically start once the playlist is parsed. As more segments are completed and sent to the client, they will be added to an array so that the loaded segments can be tracked.
3.5 **SNAPSHOT version**

The **SNAPSHOT** version is based on the **Stable** version and basically functions the same. The difference is that in the **SNAPSHOT** version a mock playlist is handmade before the transcoding even begins. This mock playlist looks at the length of the original video the user wanted to watch and creates an approximation of what the final playlist would look like after a successfully completed transcoding. This is made to let the video player understand how long the full video will be, which is a crucial step in allowing users to interact with the entire video during the transcoding process.

The logical flow of the **SNAPSHOT** version is as follows:

1. The user chooses a video the same way as in the **Stable** version. It is loaded by clicking the **SNAPSHOT** button, which calls the `playVideo()` function.
2. All transcoded videos are removed.
3. A request is sent to the server to create a mock playlist based on the original video.
4. The playlist url is set.
5. A request is sent to the server to start transcoding the video.
6. After 2 seconds, `startPlayback()` is called. From here, things are set up the same as in the **Stable** version.

As soon as playback starts, the video will appear to be as long as the original video. Since the player gets its information about the segments from the playlist, it will think all the segments are already complete, and try to load them. When it tries to load a segment that is not complete yet, and fails to find it, it will try again after 2 seconds. This will continue until all segments are actually complete and have been loaded.

Next follows a description of the most important functions used in this version, one-by-one.
3.5.1 playVideo()

Starts the same as its stable counterpart, and then calls the function createMockPlaylist() which sends a GET request to the server for a PHP script called create_mock_playlist.php. Unsurprisingly, this script creates the mock playlist. It is a simple script that looks at the length of the original video, creates a file and writes text to it to mimic the appearance of an HLS playlist.

It then calls the function convertVideo(0), which works the same as its stable counterpart, but also sends information about which segment to start the transcoding from. At start up, the start segment is set to 0, which is the first one. This function sends a GET request for the PHP script encoder.php, explained in Section 3.5.2.

After that, this information is gained when the user seeks in the video. An event listener is added when the web page is first loaded, that listens to the "seeking" event. When the user seeks, an event is triggered, and the current time of the video (after seeking) is divided by 2 (since the segment length is 2), rounded down and stored in a variable startSegment. The script now checks if this segment is already loaded or not. If it is, nothing happens. If it is not loaded, convertVideo(startSegment) is immediately called to send a request to the server to start another FFmpeg thread, starting from startSegment.

3.5.2 encoder.php

Gets the name of the video to be transcoded and which segment to start the transcoding at from the client. Before the transcoding starts however, it checks to see if that segment is currently being transcoded. While the client keeps track of which segments have already been loaded, encoder.php keeps track of segments that have not been loaded yet but is already in the process of being transcoded. This is to avoid starting multiple transcoding procedures of the same segment. If that happens, the script returns.

Otherwise, the script is put on hold until create_mock_playlist.php is finished creating the mock playlist. FFmpeg is then called with a set of
predefined settings. These settings are exactly the same as in the **Stable** version, with a few additions:

- `-ss ".${startTime}"` time to start transcoding.
- `-hls_flags temp_file` gives segments a temporary name until they finish.
- `-start_number ".${startSegment}"` gives segments the correct number.

Finally, an output must be specified to encode with FFmpeg, and when encoding to HLS that output is the playlist. To avoid overwriting the mock playlist, the actual playlist created by FFmpeg is output to a different folder than the segments. The segments are placed in the same folder as the mock playlist.

A more thorough description of some of the above settings is warranted:

- **hls_flags temp_file**
  When encoding to HLS, the standard behavior of FFmpeg is to start by creating an empty file for each segment that is currently being transcoded with the same name as the completed segment. This proved problematic while using hls.js together with the mock playlist (normally a segment is only added to the playlist once it is complete). If the client tries to load a segment, and finds it, it will be loaded regardless if the file is complete or not.
  
  To get around that problem, segments are instead initiated with the file extension ".tmp", and the extension is removed only once the segment is complete. The client does not look for these files since they are not in the playlist, so segments are only loaded once their names change.

- **start_number ".${startSegment}"**
  When a user seeks in the video, the video player will look for the segment corresponding to the current time. If the user seeks to 10 seconds into the video, the video player will look for segment 5, and hence FFmpeg must start that transcoding instance at segment number 5.

The same settings as in the **Stable** version are used to run FFmpeg in the background.
4 Results

4.1 High-level system analysis

With the settings used in Section 3.4.3, the Stable version runs without either buffer stalling or errors on all demo videos that were used. User may choose a video to watch, select the Stable version and sit back and enjoy watching it. While watching, they may freely seek ✓, pause ✓ and play ✓ (and adjust the audio volume and toggle full screen) without risk of getting an error or having to wait for the video to buffer.

That being said, the original goal of being able to seek to any part of a video during transcoding has not exactly been met in either version. In the SNAPSHOT version, users can seek to any part of the video, but doing so makes it far more prone to buffer stalling than the Stable version, where users are limited to seeking to parts of the video that have already been transcoded.

Shown next in Section 4.2 is the impact both versions have on CPU and RAM usage, followed by an analysis in Section 4.3 of compression versus transcoding speed as the codec presets are varied. This section concludes with a thorough look at the usability and challenges facing the SNAPSHOT version in Section 4.4.

4.2 System evaluation

To measure performance while running both the Stable and SNAPSHOT version, the Python library psutil [40] was used. The versions were each run once using the video fantasticbeasts.mp4, while a Python script system_monitor.py using psutil ran for 240 seconds, collecting data about the CPU and memory usage every second. Figures 4 and 5 show how this data varies over time.

Before collecting the data, the application web-page was opened and fantasticbeasts.mp4 was chosen in the dropdown menu, as shown in Figure 3. All other applications and windows were carefully closed to ensure as few background processes were running as possible.
The script `system_monitor.py` was then started, followed by about 20 seconds of leaving the computer untouched. This is the idle time that can be seen at the beginning of the graphs in Figures 4 and 5 where both CPU and memory usage are at their lowest.

When those 20 seconds were up, the button choosing which version to run was pressed, after which the computer was again left untouched until the full 240 seconds had passed and `system_monitor.py` terminated. Figure 4 shows the results of doing this with the Stable version, and Figure 5 for the SNAPSHOT version.

The highest values for total CPU and memory usage are recorded right after pressing the button. This is when the video is being played back and transcoded simultaneously, and it holds true for both versions. Coincidentally, this is the only time that FFmpeg has a non-zero CPU and memory usage, since before pressing the button no FFmpeg thread existed, and as soon as the transcoding finishes this thread terminates.

The peak resource usage is followed by a slight drop in total memory usage, and a significant drop in total CPU usage as the FFmpeg thread terminates at around the 110 seconds mark. The FFmpeg thread exists no more after this point, but some resources are still being used to play back the video, which is now completely transcoded.

At around 180 seconds, the video finishes playing and total CPU usage drops down to its initial (lowest) value. The computer is idle once again. Curiously, total memory usage still remains higher than at the start, which may be because the computer has temporarily stored the video in memory for future playback.
The graphs show that the two versions have almost the exact same impact on performance. The only significant difference between the two versions is the total memory usage, which peaks at 54% in the Stable version compared to 41% in the Snapshot version. However, since in both versions FFmpeg takes up roughly 3% of memory and they both use the same video player, some unrelated task running in the background is likely the cause of this discrepancy.

4.3 Compression versus transcoding speed

To achieve a satisfactory user experience while watching a video as it is being transcoded, it is important to find a balance between transcoding speed and
compression. Essentially, if the video can not be transcoded faster than it can be played back at normal speed, it can not be watched from start to end without buffering. Likewise, if too much of compression is sacrificed for transcoding speed, the bitrates of the transcoded videos will increase and more bandwidth will be required to deliver them to the client.

After briefly watching through the videos, no discernible difference in quality was found between any video transcoded using different presets.

In Section 4.3.1 the resulting bitrates of the transcoded test videos using the different x264 presets are presented. In Section 4.3.2 the transcoding times for the same operation is given.
Table 2: Bitrates of the original test videos, and the bitrates of each of them transcoded using the x264 presets. Bitrates are given in Kb/s. The lowest values are colored green, and the highest red.

<table>
<thead>
<tr>
<th>Video</th>
<th>Original</th>
<th>Ultra 1</th>
<th>Super 2</th>
<th>Veryfast 3</th>
<th>Faster 4</th>
<th>Fast 5</th>
<th>Medium 6</th>
<th>Slow 7</th>
<th>Slower 8</th>
<th>Veryslow 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aerial</td>
<td>203</td>
<td>1070</td>
<td>278</td>
<td>139</td>
<td>186</td>
<td>206</td>
<td>205</td>
<td>198</td>
<td>197</td>
<td>186</td>
</tr>
<tr>
<td>Earth</td>
<td>156</td>
<td>861</td>
<td>290</td>
<td>103</td>
<td>133</td>
<td>143</td>
<td>146</td>
<td>143</td>
<td>144</td>
<td>140</td>
</tr>
<tr>
<td>Beasts</td>
<td>673</td>
<td>1527</td>
<td>800</td>
<td>562</td>
<td>630</td>
<td>650</td>
<td>633</td>
<td>615</td>
<td>607</td>
<td>578</td>
</tr>
</tbody>
</table>

4.3.1 Effects of x264 presets on transcoded bitrates

As can be seen in Figure 6, which shows bitrates of the transcoded videos versus the original video, the leftmost preset (which is Ultrafast) provides by far the highest bitrate. Table 2 also shows this, where the highest bitrate of the transcoded videos is colored red.

Moving on to the other presets, the bitrates all take a large dive, with notable minima achieved using the Veryfast preset, as can be seen in Figure 6. This also becomes clear when looking at Table 2, where the lowest bitrates are all colored green, showing that the Veryfast preset unanimously achieves the lowest bitrate among the test videos. In all cases, bitrates start rising after that, to begin dropping again towards the Veryslow preset.

4.3.2 Effects of x264 presets on transcoding speed

The graphs in Figure 7 present the duration of the original test videos versus the transcoding time of those videos using the different x264 presets. These graphs show that the first preset, Ultrafast (which provided the highest bitrate), is not necessarily the fastest. This can also be seen in Table 3, where the fastest transcoding times are colored green.

Notably, using the Veryfast preset (which provided the lowest bitrate), resulted at worst in a transcoding time that is merely 24% of the duration of the original video, meaning playback without buffering during transcoding is
Figure 6: The bitrates of the transcoded videos using different x264 presets (black) versus the bitrate of the original video (red). The codec presets are ordered from 1 representing Ultrafast to 9 representing veryslow (values and enumeration of presets in Table 2).

In all cases, transcoding times rise sharply at the Veryslow preset. Interestingly enough, looking at Figure 7 graph (b), it can be seen that even the slowest preset still manages to be faster than the duration of the original video. This could potentially be attributed to the complexity of the video. As described in Section 3.2, this video (earth.mov) contains little frame-to-frame change and a lot of constant black space.
Figure 7: The transcoding time using different x264 presets (black) versus the duration of the original video (red). The codec presets are ordered from 1 representing Ultrafast to 9 representing veryslow (values and enumeration of presets in Table 3).

4.4 Usability of the SNAPSHOT version

This version works mostly without issues when simply watching a video from start to finish without interacting with it. This is especially true for aerial.mov, which is a very short video, that can be consistently played back without buffer stalling (at least using the same settings as in Section 3.4.3).

Interesting issues arise however when trying to seek to a part of the video that has not been transcoded yet. These issues are described next.
<table>
<thead>
<tr>
<th>Video</th>
<th>Original</th>
<th>Ultra 1</th>
<th>Super 2</th>
<th>Veryfast 3</th>
<th>Faster 4</th>
<th>Fast 5</th>
<th>Medium 6</th>
<th>Slow 7</th>
<th>Slower 8</th>
<th>Veryslow 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aerial</td>
<td>11.05</td>
<td>1.67</td>
<td>1.29</td>
<td>1.64</td>
<td>2.73</td>
<td>3.35</td>
<td>4.18</td>
<td>5.41</td>
<td>7.91</td>
<td>15.73</td>
</tr>
<tr>
<td>Earth</td>
<td>30.55</td>
<td>3.73</td>
<td>3.28</td>
<td>3.97</td>
<td>6.47</td>
<td>7.53</td>
<td>7.75</td>
<td>9.29</td>
<td>13.68</td>
<td>25.95</td>
</tr>
<tr>
<td>Beasts</td>
<td>156.14</td>
<td>25.91</td>
<td>26.10</td>
<td>37.13</td>
<td>61.68</td>
<td>73.41</td>
<td>83.44</td>
<td>111.67</td>
<td>191.78</td>
<td>389.66</td>
</tr>
</tbody>
</table>

Table 3: Durations of the original test videos, as seen under the "Original" column. The rest of the columns show the time it took to transcode that video using the respective setting. All values are given in seconds. The (three) instances where the transcoding time exceeds the original duration are colored red. The fastest transcoding times are colored green.

### 4.4.1 Seeking to unfinished parts of the video

When a user seeks forward the client automatically sends a request to the server to start transcoding at the segment corresponding to the current time the user seeked to. Another FFmpeg thread is then started.

Smooth playback can not be guaranteed when doing this. Even in the case where only one additional FFmpeg thread is started a long buffering time will be encountered. Other curious issues may also occur, such as the video not resuming playback or appearing to increase in duration.

Additionally, the player will stop loading segments at locations close to where the user was before seeking, and will not resume loading them. An example of this is given below:

**Example:** Assume the user watches a video from the start, and segments up until 30 seconds into the video have been loaded. If the user then seeks to 2 minutes into the video, the player will instead start loading segments from that mark, and there will be a gap between 30 seconds and 2 minutes where no segments have been loaded.

The same thing happens in the **Stable** version, although in that version it does not prevent the segments in that gap from being loaded when the user wants to watch that part of the video. In the **SNAPSHOT** version however, those
Figure 8: CPU and memory usage over time while running the SNAPSHOT version and starting 8 concurrent FFmpeg threads. Shown are both the total CPU (a) and memory usage (b) for the whole computer, and for just the first FFmpeg thread in (c) and (d).

segments are never loaded, and the user will only experience indefinite buffering while trying to watch that part. This may be due to how hls.js loads segments into the video player.

4.4.2 Starting multiple FFmpeg threads

The long buffering time encountered when seeking forward can be attributed to the computer slowing down when more FFmpeg threads are started. If about 10 or more FFmpeg threads are started simultaneously, the computer will slow down to the point of being unusable.
Figure 8 shows a system evaluation while running 8 concurrent FFmpeg threads. Shown here is the total CPU and memory usage, and the CPU and memory usage for the first thread that was started. The data was collected by letting the script `system_monitor.py` run for 1000 seconds. The video was fast-forwarded 7 times, each time to a part of the video that had not been transcoded yet, thereby starting a total of 8 FFmpeg threads. This was done to simulate a real world situation where a user might seek forward several times until they find what they are looking for.

It can be seen here that total CPU usage lies steadily at almost 100% while FFmpeg is running, while total memory usage goes from 35% to about 60% at its peak. Total memory usage can be seen rising and dropping sharply, which can likely be attributed to the FFmpeg threads starting and terminating.

Shown in the two bottom graphs are the CPU and memory usage of the first FFmpeg thread that was started. As could also be seen in Figure 4 and Figure 5, memory usage is still only 3% for a single thread. However, since this FFmpeg thread now has to share the CPU with 7 other threads, CPU usage is initially much lower and only rises as the other FFmpeg threads terminate and more CPU power is freed. No data was collected about the remaining 7 threads.

Segments created by a thread further ahead will always be overwritten by threads that come after it. An example of this is given below:

**Example:** Assume a user chooses a video to watch with the **SNAPSHOT** version. A new FFmpeg thread (thread 1) is then created, starting at segment 0. The user then immediately seeks to 20 seconds into the video, which creates another FFmpeg thread (thread 2), starting at segment 10. thread 2 will keep transcoding segments up until the end of the video. thread 1 will do the same, and will in the process of doing so overwrite all the segments created by thread 2. In the end, segments 0-9 are transcoded once, while segments >10 are transcoded twice, meaning unnecessary work has been done.

It is easy to imagine that such a scenario can quickly get out of hand if the user seeks forward multiple times, creating multiple new threads whose work will be overwritten.
Even if one were to solve this issue, by for example making an FFmpeg thread stop if it finds a segment that already exists, there is still the problem of making segments created by multiple different threads to sync up. Consider the example below:

**Example:** An FFmpeg encoder is set up to transcode a video in sets of 5 segments. Set 1 contains segments 0-4, set 2 segments 5-9, and so on. When a set is completed, the thread that worked on those segments is killed and a new thread is started to work on the next set. All segments created this way are added to the same folder, and a mock playlist is created the same way as described in Section 3.5.1.

If that playlist is loaded into a player, it would try to play the segments back to back as described in the playlist, just like a regular HLS video. However, the transition between segments that have been created by two different FFmpeg threads will not be smooth, resulting in frozen frames, a green screen or other playback errors. This leaves space for interesting future work, since better management of the FFmpeg threads could drastically improve performance.

An analysis of multithreading specific issues is omitted, since it would be beyond the scope of this thesis.

## 5 Discussion

### 5.1 Results

While the system evaluation in Section 4.2 provides useful information about how much CPU power the application requires compared to RAM, one must remember that (i) the computer that the tests were run on is likely significantly less powerful than what is available to the Police, and (ii) it was conducted in an environment without interaction with the computer, which may not accurately reflect a real world scenario. Nevertheless, it seems clear that the primary bottleneck here is the CPU since the FFmpeg thread alone shown in Figure 4 and 5 uses almost all available CPU power, but only a fraction of RAM.
The data presented in Figure 8 presents another system evaluation that may closer reflect how the application would be used in real life, since it simulates a user seeking forward until they find what they are looking for, starting 8 concurrent FFmpeg threads in the process. The graphs presented here reinforce the idea that the CPU is the primary bottleneck, especially looking at graph (c), where one can see the first FFmpeg thread being given much less CPU power to work with as more FFmpeg threads are started. One can also infer from this figure how hopelessly slow the application is in such a scenario. Looking back at the figures in Section 4.2 it can be seen that FFmpeg runs for about 70 seconds until it is finished transcoding the video, and that the video itself finishes playing roughly 160 seconds after starting the application. However, while running 8 concurrent FFmpeg threads, the first FFmpeg thread (which transcodes the entire video from start till finish) requires closer to 450 seconds of work before it finishes, which is almost 3 times the duration of the video itself. This may cause the user experience to suffer severely, and indeed while conducting the test the video would not resume playing (after having fast forwarded 7 times) until almost the full 450 seconds had passed.

The bitrates presented in Figure 6 may seem surprising. One would expect the slower settings to result in better compression as more speed is sacrificed, but it is not entirely clear why the veryfast preset gives the lowest bitrate. A closer look at these presets reveals they are just collections of FFmpeg settings being tweaked, and a closer analysis of these settings would need to be conducted before drawing any real conclusion. The same transcoding test as in Section 4.3.1 should also be carried out on many more videos. Even though the videos used for this thesis differed quite a lot in their quality and complexity, 3 videos is still a relatively small sample size.

5.2 Method

Owing in part to a lack of experience in web development, the choice of using PHP for the back-end code and Chrome as the web browser stands in contrast to the frameworks in use by Swedish law enforcement today. They mainly work with Java on the back-end, and all their applications are developed for IE11 and Edge, neither of which the application developed for this thesis were
tested on. The choice of frameworks used during this thesis were mostly made based on perceived popularity, so as to quickly find information and limit the time required to set up the basic things necessary to run a web application. A more experienced web developer could perhaps have limited himself/herself to the relevant frameworks. That being said, all the tools in the application do support IE11 and Edge. The back-end code consists mostly of conditionals and shell commands, and could likely be converted to Java code without issue.

The decision to use HLS over any other streaming protocol was a rather easy one to make, given its popularity and FFmpeg having native support for transcoding to HLS. Smooth streaming and HDS can in all likelihood be disregarded entirely, since the Police will not be using either Flash video or the XBOX One. MPEG-DASH is likely something to keep a lookout for in the future though, given its rising popularity and backing by titans of the industry.

5.3 Wider context

Law enforcement serves a critical role in our society, and aiding the police as a software developer in their daily work is a meaningful task. Videos used as evidence need a standardized way of being dealt with to escape the cumbersome procedures of today. Enabling them to work with videos in their web applications will be a huge step towards a more effective work flow. Both the research and the work carried out in this thesis will serve Swedish law enforcement well when they decide to implement support for videos. The research provides crucial insight into what is required for a good viewing experience of streaming video, and the Stable and SNAPSHOT versions of the application that was developed for this thesis could serve as guidelines during their own development process.

6 Related work

Krishnamoorthi et al. [42], [43] have done work on branched video, where users can select their own path through the video by choosing options
presented to them while watching. Such videos are inherently interactive which ties into the goals of this thesis. In their article they present a video player that provides seamless playback of branched videos, using prefetching of data, buffer management and parallel TCP connections. The techniques they use could prove helpful in delivering a smooth experience to law enforcement working with videos.

Krishnamoorthi et al. [44] have furthermore done research on how to optimally preload videos that are related to the video a user is currently watching, and present a solution that provides careful prefetching and buffer management. An investigator may want to quickly switch between several videos related to the same crime, whereupon providing them a way to quickly switch between these videos becomes important.

Potentially tying into viewing footage from surveillance cameras is the work done by Carlsson et al. [45], where they introduce the concept of a "multi-video stream bundle" that consists of multiple parallel video streams that are synchronized in time, each providing the video from a different camera capturing the same event. To achieve close to seamless switching between video streams they use their own prefetching and buffer management solution. Their work could prove helpful in viewing an event captured by multiple surveillance cameras, where an investigator may want to quickly switch between camera views several times. More work along the lines of prefetching management has been done by Mathias et al. [46], on the topic of 360° video, which is yet another type of interactive video.

Samani et al. [47] discuss jump-cuts in videos produced by surveillance systems, and present a novel method for detecting jump-cuts in real-time, enabling self-diagnostics of surveillance systems. Government agencies, such as law enforcement, who deal with surveillance systems will want to know if their videos or cameras have been tampered with. An algorithm created to autonomously examine thousands of hours of video and detect jump-cuts could prove useful in such cases.

To battle the high energy consumption required to transcode videos, Lee et al. [48] propose algorithms that perform transcoding operations selectively to balance transcoding loads with quality-of-experience. As law enforcement
receives more and more videos, it becomes increasingly important to think about the high computational cost of video transcoding.

Even though videos used by law enforcement are not for entertainment, user satisfaction still serves a critical role. Police officers should want to work with the new systems being developed. Florin Dobrian et al. [49] have found that the percentage of time spent in buffering has the largest impact on user engagement across all types of content. This result is highly relevant considering the heavy buffering times that can be encountered with the SNAPSHOT version. More along the lines of user satisfaction is the work done by Tobias Hoßfeld et al. [50], where they discuss ways to switch between different qualities of the same video depending on available bandwidth to optimize user quality-of-experience. This functionality is not yet implemented in the application presented here, but could make for interesting future work.

Rate estimation techniques are discussed by Saamer et al. [51] where they conduct experiments on, among other things, an adaptive player’s ability to react to changes in the underlying network available bandwidth. More work on rate estimation have also been done by Te-Yuang Huang et al. [52], and a network simulator where one can test e.g. bandwidth limitations is presented by Luigi Rizzo [53].

Omar A. Niamut et al. [54] present the Spatial Representation Description feature (a part of MPEG-DASH), which enables DASH clients to retrieve only those video streams at those resolutions that are relevant to the user experience. Though they do not use HLS, their work could prove helpful in the context of viewing surveillance footage from multiple viewpoints.

Tan et al. [55], [56] propose a way to realize fast-forwarding and fast-reverse in digital videos through video transcoding. Their approach only transcodes the necessary frames; e.g. if a speed-up factor of 4 is desired, only every fourth frame is transcoded and the fast video playback can be achieved by playing the transcoded frames at normal speed. They are especially concerned with realizing these features in systems with limited access to bandwidth or minimum computation resources. Their approach could be useful to Swedish law enforcement to provide a better video viewing experience, especially in conjunction with the seeking technique during transcoding presented in this
thesis.

7 Conclusion

This thesis presents a novel application for watching and interacting with videos during transcoding. It consists of two versions. The first version, called Stable, allows users to watch and seek freely across the video up until the currently highest number of transcoded segments. The second version, called SNAPSHOT, allows users to seek to any arbitrary part of the video regardless if that part has been transcoded or not, but does so at the cost of (at least) long buffering durations.

The Stable version, if elaborated upon to support more formats than MP4 and MOV, could be implemented as is into the systems used by Swedish law enforcement. While it does not enable users to arbitrarily seek forward in a video, it provides a very stable way to view and interact with a video during transcoding. This version could allow investigators to comfortably work with shorter duration videos, e.g. videos taken by police officers on-site with their mobile phones from a crime scene. Experimental results show that using the veryfast preset while transcoding with FFmpeg results in the lowest possible bitrate of the transcoded video, while being fast enough to enable seamless playback of the video during transcoding. The results show that the same thing holds true for the SNAPSHOT version.

The SNAPSHOT version however will require work before it can be implemented in any system. Experimental results show that the CPU is the primary bottleneck for performance if users seek forward multiple times. Running it on more powerful hardware or off-loading work to the GPU may increase performance drastically, but there are inefficiencies current in the program that should be dealt with regardless. The most pressing task to deal with is better management of FFmpeg threads. As it stands, there is no limit to how many threads can be started and any given thread ignores the work done by the other threads.

For future work, limiting thread count and making threads stop when they try
to transcode segments that already exist could be the first steps towards realizing the full potential of the application.
References


[25] FFmpeg. 4.25 segment, stream_segment, ssegment. 
https://www.ffmpeg.org/ffmpeg-formats.html#segment_002c-
stream_005fsegment_002c-ssegment (Accessed September 8, 2019).


http://www.streamingmedia.com/Articles/Editorial/Featured-
Articles/Encoding-2020-Experts-Predict-the-Future-of-Video-


[34] Wikipedia. Iso base media file format. 


[37] HP. Amd a6-7310 apu with amd radeon r4 graphics, 4 cores, 1.5 ghz, 4gb ram.

[38] Wikipedia. elementary os. 


[41] Beandog. Encoding presets for x264. 


[45] Niklas Carlsson, Derek Eager, Vengatanathan Krishnamoorthi, and Tatiana Polishchuk. Optimized adaptive streaming of multi-video stream...


[54] Omar A. Niamut, Cyril Concolato, Emmanuel Thomas, Franck Denoual,
Lucia D’Acunto, and Seong Yum Lim. Mpeg dash srd - spatial

[55] Yap-Peng Tang, YongQing Liang, and Jun Yu. Video transcoding for fast
forward/reverse video playback. Proc. International Conference on Image
Processing (ICIP), 1, 2002.

[56] Yap-Peng Tang and YongQing Liang. A unified transcoding approach to
fast forward and reverse playback of compressed video. Proc. IEEE
Transactions on Consumer Electronics, 49(4), 2003.

[57] Cheekysoft. Crop mp3 to first 30 seconds.
https://stackoverflow.com/questions/43890/crop-mp3-to-first-30-seconds
(Accessed September 8, 2019).

https://en.wikipedia.org/wiki/Digital_container_format
(Accessed September 8, 2019).