Creating an Ecoder Ladder

I've always thought about the concept of an encoding ladder without really knowing what it was and why it was important. I came accross the concept again a few weeks ago when researching the AV1 codec.

Before jumping into details, let's define what an encoding ladder is:

> Your encoding ladder is the set of encoding parameters that you use to create the various files that you deliver adaptively to your web viewers. These encoding parameters can live in your on-premise encoder, in your cloud encoder, or in your online video platform (OVP).
>
> Five Signs Your Encoding Ladder May Be Obsolete

So what ladder should we use.

It depends on the budget and what streams your audience is using.

Figure 1: Apple recommended bitrates from TN 2244

We don't have to use all or any of the encodings.

In my research I came across Five Views of your Encoding Ladder by Jan Ozer where he discusses encoding ladders and how to set them up.

We will discuss both an ideal ladder and a more practical ladder for web content that we want to publish.

The Ideal Ladder

For this ideal ladder we'll skip pricing concerns. In an ideal world we'd take that into consideration as well.

The table below, taken from Ozer's Linked In article, uses Apple's Technical

Note 2244 as the basis for the encoding ladder.

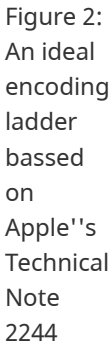

Figure 2: An ideal encoding ladder bassed on Apple''s Technical Note 2244

I’ve always wondered how much we can cut off the bottom end for our American users. I’m thinking about removing either 234p, 270p or both depending our target audience.

I’m also questioning whether we need data rates over 4500. Unless we can derive actual differences between the 4500 and higer data rates I don’t think we need to encode to the highter rates.

This is a good starting point but you’ll have to do a lot of testing with your content and with potential delivery channels before you can pronounce it good and use it to deploy content.

An Optimized Ladder for Web Delivery

Working with video for web delivery can be done either of two ways, we can use a video delivery network or we can upload our content to different providers for delivery.

I’ve chosen to work with these providers:

- Youtube
- Vimeo
- Twitch

Although Twitch was intially geared towards computer gamers’ broadcast my experience has been vastly different in the kind of games that first attracted me to earth.

The World According to Twitch: Winning With Experimentation covers the gaming aspect of the platform along with some explorations of what other content can be used in the platform.

GVC
Usage
Content
Figure 3:
GVC
Usage
Content

The important part of that article, to me, is what it doesn't cover. For some people it wasn't gaming broadcast that attracted people to the platform. Channels like Geek and Sundry were the first Twitch Streams that I became interested in. Later Sean Larkin and Jim Lee that I really saw the potential in the platform. Larkin and Lee both provide some gaming content but their primary content is non-gaming related, Larkin produces Webpack-related software content and Lee creates sketches and drawings of Image and DC characters.

So coming back to the encoding ladder for these platforms we can look at what the requirement for each platform are.

Below I'm providing a condensed version of my Encoding Spreadsheet that I compiled to try and figure out what the encoding ladder should be like

Provider	Size	Aspect Ratio	Container	Rate (FPS)
Youtube 4k	3840x2160	16 x 9	MP4	25
Youtube 2k	2560x1440	16 x 9	MP4	25
Youtube 1080p	1920x1080	16 x 9	MP4	25
Youtube 720p	1280x720	16 x 9	MP4	25
Youtube 480p	854x480	16 x 9	MP4	25
Youtube 360p	640x360	16 x 9	MP4	25
Vimeo 4k	3840×2160	16 x 9	MP4	25
Vimeo 2k	2560×1440	16 x 9	MP4	25

Compiled data about encoding requirements for video on demand clips for Youtube, Vimeo and Twitch.

Provider	Size	Aspect Ratio	Container	Rate (FPS)
Vimeo 1080p	1920×1080	16 x 9	MP4	25
Vimeo 720p	1280 × 720	16 x 9	MP4	25
Vimeo SD	640 × 360	16 x 9	MP4	25
Twitch 1080p	1920x1080	16 x 9	MP4	25
Twitch 720p	1280x720	16 x 9	MP4	25
Twitch 480p	854x480	16 x 9	MP4	25

Looking at the table and their resources on compression:

- Youtube
- Vimeo
- Twitch

We can see the commonalities. For the most part we should be able to create one file to push to Youtube and Vimeo and one additional stream, if needed or wanted, for Twitch.

Creating files with FFMPEG

Using the FFMPEG command line utility we'll do several passes at converting the video to the format we want.

In the first pass we'll worry about bitrate and framerate for a 360p version of the video. FFMPEG's default frame rate is 25 so we can leave it out to make the code simpler or add it to make sure we account for it if/when the default changes.

The initial command to create a 360p version of the video at 25fps looks like this:

```
ffmpeg -i Agent_327_Operation_Barbershop.mkv -b:v 2m -r 25 agent_327_360p
```

Next we'll make sure that the video size is correct. for this we'll use FFMPEG's `-s` attribute along with the size that we want to use, in this case the full attribute will be `-s 640x360`. The modified command looks like this:

```
ffmpeg -i Agent_327_Operation_Barbershop.mkv -b:v 2m -r 25 -s 640x360 agen
```

The following step is to make sure the resulting video clip keeps a 16:9 aspect ratio. For this we use the `-aspect` parameter with the value we want the resulting video to have. The full parameter is `-aspect 16x9`.

```
ffmpeg -i Agent_327_Operation_Barbershop.mkv -b:v 2m -r 25 -s 640x360 -asp
agent_327_360p_sized_ratio.mp4
```

Youtube doesn't like interlaced video so we'll deinterlace it ourselves rather than leave it in the vendor's hands with unpredictable results.

```
ffmpeg -i Agent_327_Operation_Barbershop.mkv -b:v 2m -r 25 -s 640x360 -asp
agent_327_360p_sized_ratio_deinterlace.mp4
```

I think that covers the basic work with video, now let's look at the audio portion of the clip. The two attributes that we want to work with for audio are `-ar` and `-ab`.

The `-ar` attribute controls the audio sampling frequency. Both Youtube and Vimeo want you to leave it at 48khz so we set the value to 48000. Note that for any audio uploaded with a sampling rate over 48kHz, Vimeo will resample your audio to 48 kHz or below.

With `-ab` we get control of the audio bit rate (expressed in bits per second). Here, again, we let Vimeo drive the process as they require a constant rate of 320 kbits per second so we set it to 320k. However, depending on the type of source material you're working with it may be possible to lower this value without losing quality.

```
ffmpeg -i Agent_327_Operation_Barbershop.mkv -b:v 2m -r 25 -s 640x360 -asp
-ar 48000 -ab 320k \
agent_327_360p_sized_ratio_deinterlace_audio.mp4
```

Using the 360p video as a reference we can work the other versions (1080p and 720p) using the command as a reference. The parameters we need to change for each version are:

- `-b:v 2m` to change the video data rate
- `-s 640x360` to change the video dimensions

We can go even further and create a script in either Python or Ruby that will automate the changes. We only need one command for each version that we want to create

DASH

If you have the storage and the bandwidth you may also want to play with converting your videos into DASH content and serve them directly through your web server. In Revisitng Video Encoding: DASH we did this using Google's Shaka Packager for creating the DASH streams and the companion Shaka Player to play the content.

Creating the manifest

For this portion we'll assume that we've created 3 files:, one for each of 360p, 720p and 1080p resolutions. For each of the resolutions we'll create both an audio and a video stream and write all those streams to a manifest file, in this case `agent327.mpd`.

Note that I'm working on MacOS and the packager is installed on my path, that's why I can get away with just running `packager` for the command, otherwise I would have to specify the full path to the application.

The full command looks like this:

```
packager \
input=agent_327_360p.mp4,stream=audio,output=360p_audio.mp4 \
input=agent_327_360p.mp4,stream=video,output=360p_video.mp4 \
input=agent_327_720p.mp4,stream=audio,output=720p_audio.mp4 \
input=agent_327_720p.mp4,stream=video,output=720p_video.mp4 \
input=agent_327_1080p.mp4,stream=audio,output=1080p_audio.mp4 \
input=agent_327_1080p.mp4,stream=video,output=1080p_video.mp4 \
--mpd_output agent327.mpd
```

We will use the manifest as the source of the video we want to play. We will use the Shaka Player to play the video on the web page.

Playing DASH content

Shaka Player is the playback component of the Shaka ecosystem. Also developed by Google and open sourced on Github.

If you're used to HTML5 the way you add DASH video is a little more complicated than you're used to.

First we create a simple HTML page with a video element. In this page we make sure that we add thescripts we need:

- The shaka-player script
- The script for our application

The video element is incomplete on purpose. We will add the rest of the video in the script later on.

```
<!DOCTYPE html>
<html>
  <head><html><!-- Shaka Player compiled library: -->    <script src="path
    <<script src="path/to/shaka-player.compiled.js"><!-- Your application
    <video id="vide<body></script>
  </head>
  <body>
    <video id="video"
      width="640"
      poster="media/SavingLight.jpg"
      controls autoplay></video>
  </body>
</html>
```

We'll break the script into three parts:

- Application init
- Player init
- Error handler and event listener

We initialize the application by installing the polyfills built into the Shaka player to make sure that all the supported players behave the same way and that there won't be any unexpected surprises later on.

The next step is to check if the browser is supported using the built in isBrowserSupported check. If the browsers supports DASH then we initialize the player by calling `initPlayer()` otherwise we log the error to console.

```
var manifestUri = 'media/example.mpd';

function initApp() {
  // Install built-in polyfills to patch browser incompatibilities.
  shaka.polyfill.installAll();

  // Check to see if the browser supports the basic APIs Shaka needs.
  if (shaka.Player.isBrowserSupported()) {
    // Everything looks good!
    initPlayer();
  } else {
    // This browser does not have the minimum set of APIs we need.
    console.error('Browser not supported!');
  }
}
'Browser not supported!'
```

Initializing the player is the meat of the process and will take serval different steps.

We create variables to capture the video element using `getElementById` and the player by assigning a new instance of `Shaka.Player` and attach it to the video element.

We then attach the player to the window object to make it easier to access the console.

Next we attach the error event handler to the onErrorEvent function defined later in the script. Positioning doesn't matter as far as Javascript is concerned.

The last step in this function is to try and load a manifest using a promise. If the promise succeeds then we log it to console otherwise the catch tree of the promise chain is executed and runs the onError function (which is different than onErrorEvent discussed earlier).

```
function initPlayer() {
  // Create a Player instance.
  var video = document.getElementById('video');
  var player = new shaka.Player(video);

  'video'// Attach player to the window to make it easy to access in the J
  window.player = player;

  // Listen for error events.er.addEventListener('error', onErrorEvent);

  // Try 'error'// Try to load a manifest.
  // This is an asynchronous process.
  player.load(manifestUri).then(function() {
    // This runs if the asynchronous load is successful.og('The video has
  }).catch(onError);  // onError is 'The video has now been loaded!'// onl
}
```

The last part of the script is to create the functions for errors (onErrorEvent and onError)

Finally we attach the `initApp()` function to the DOMContentLoaded event.

```
function onErrorEvent(event) {
  // Extract the shaka.util.Error object from the event.
  onError(event.detail);
}

function onError(error) {
  // Log the error.
  console.error('Error code', error.code, 'object', error);
}

document.addEventListener('DOMContentLoaded', initApp);
'Error code'
```

If everything works out OK we should have a video playing on screen and the video will switch to the most appropriate version for your hardware and network conditions.