Video Processing

Why?

With the internet becoming readily available to many more consumers, the range of networks and devices we are exposed to is increasing rapidly. To ensure that our content can be consumed by as many consumers as possible, it must be adapted and optimized to suit the needs of the consumer.

In order to do so, high-quality input videos must be processed to create multiple versions of the same file, each with a different resolution, bit rate and quality level, a process known as transcoding. Originally, video transcoding was employed in order to save storage space. However, as storage space is cheaper than ever, transcoding is now used to enable content *streaming* across platforms (our focus is on Adaptive HTTP Live Streaming).

How?

To accomplish the entire video processing process in-house, the following technologies were used:

1. Ffmpeg: Cross-platform solution to convert audio and video from one format to the other. Ffmpeg also helps us in preparing the input video for Adaptive HTTP Live Streaming.
2. Ffprobe: Cross-platform solution to probe a media file for meta data like duration, video width/height, audio/video bit-rate, audio sampling rate, audio channels etc. This data helps in adding intelligence to our transcoder to decide the output file parameters.
3. JW Player: Video player that offers built in support for Adaptive HLS Streaming.

There are four major steps involved in the entire video processing process to prepare it for adaptive HTTP Live streaming with JW Player:

1. Probing media file for meta-data

Before actually transcoding the input media file, we must first decide the output media files' parameters, most importantly the video and audio bit-rates, the resolution, and the frame rate. In order to reach the optimum values for these parameters, we extract information from the input file using ffprobe, and perform some calculations on this data.

2. Transcoding

Once we have all the meta-data from the input file, we perform various checks to evaluate the parameters for the output videos.

- Frame Rate

  We have set a base frame rate (currently 15 FPS) which we feel is optimum in most cases. The program selects the frame rate depending on which value is smaller, the base frame or the input video frame rate (doesn't make sense to upscale a video, since this increases file size needlessly without improving video quality).

- Video Bit-rate

  The process to calculate the video bit-rate is a bit more complex than the other calculations, since this factor alone dictates aspects such as the visual appeal and file size of the transcoded video.

  Based on the input file's aspect ratio, we first calculate the output file's width, given a certain vertical resolution (e.g. if the input file is a 16/9 video, and we want to scale it down to 360p, the output file will have a width of 360 * 16/9 = 640p). We do this to ensure that the scaling process doesn't stretch or shrink the video. By multiplying the width with the height, we get the number of pixels per frame; this value multiplied by the frame rate gives us the number of pixels per second. Since the bit-rate is defined in a unit like kilo-bits per second, we need to create an index, henceforth called pixel index, which defines the bits per pixel. We can multiply this with pixels per second to generate the bit-rate.

  The challenge lies in generating a pixel index matrix that can push down file size sufficiently, but not degrade video quality. Upon research, it was discovered that the bare minimum pixel index is 0.03 bits/pixel, which values above 0.1 have no noticeable effect. Keeping this in account, we generated the following matrix:

Resolution	Quality	Bits/Pixel
360p	LOW	0.040
	MEDIUM	0.062
	HIGH	0.089
432p	LOW	0.040
	MEDIUM	0.062
	HIGH	0.086
540p	LOW	0.040
	MEDIUM	0.063
	HIGH	0.083
720p	LOW	0.040
	MEDIUM	0.062
	HIGH	0.080
1080p	LOW	0.040
	MEDIUM	0.061
	HIGH	0.089

  Using these values for the pixel index, we generated the optimum video bit rate. One last check is performed to see if this value is lower than the input video's bit-rate. If the input file's bit rate is already low enough (i.e. it is already a very low quality video), we do not use the calculated bit rate and instead proceed to lower other parameters of the input file.

- Audio Bit-rate

Resolution	Quality	Audio Bit rate (kbps)
360p	LOW	64
	MEDIUM	64
	HIGH	64
432p	LOW	64
	MEDIUM	64
	HIGH	64
540p	LOW	64
	MEDIUM	96
	HIGH	96
720p	LOW	128
	MEDIUM	128
	HIGH	128
1080p	LOW	256
	MEDIUM	256
	HIGH	256

The program simply looks into this matrix and pulls out the corresponding bit rate. This value is compared with the input file's audio bit-rate; if it is lower, we use this value in the transcoding process (doesn't make sense to upscale audio, since this increases file size needlessly without improving audio quality).

- Resolution
  Deciding the set of output resolutions is fairly simple. We have a set of defined resolutions (360p, 432p, 540p, 720p, 1080p) that we intend to support. The program simply converts the input video into all the resolutions below it. For example, an input video of 720p is converted into 540p, 432p and 320p. Additionally, if the input file has a pixel index higher than 0.1, a lower quality file of the same resolution is also created.

  Once we have all the parameters of the output video, we simply need to invoke the ffmpeg command with the parameters we have calculated to transcode the file. Earlier, the program would invoke one ffmpeg process for each output resolution (3 processes if 360p, 432p and 540p outputs were to be generated). However, it was discovered that this was very CPU intensive and an overall slow process. To fix this, we used ffmpeg's built in support for generating multiple outputs from the same input file. Now, only one process is fired, regardless of the number of output resolutions.

3. Segmentation

To enable a video for HLS, it needs to be broken down into smaller segment files (`.ts files`), and all the segments need to be indexed via a play list file (`.m3u8 files`). Fortunately, ffmpeg has built in support for generating the segments and the play list file using the `-stream_segment` muxer (shorthand `-ssegment`) and `-segment_list`. This process is repeated for each output resolution. However, to enable *Adaptive* HLS, we need to provide JW Player with a combined `.m3u8` play list that contains the information for the other play list files. This `master.m3u8` is generated manually by the program since ffmpeg doesn't support this feature yet. This file can be supplied to JW Player to enable Adaptive HLS.

Earlier, the transcoding and segmentation processes were split, i.e. the system would first produce the transcoded mp4 files (1 ffmpeg process for multiple resolution outputs), and then further segment those files (1 ffmpeg process for each resolution file produced earlier). This was proving to be very time consuming, since the ffmpeg process had to go through the files multiple times. To streamline this process, the segmentation and transcoding was performed at the same time. Not only does this all work in one process, it was also much faster. These segments could later be concatenated (using `-concat`, built into ffmpeg) into one mp4 file if that file is needed for storage or fallback purposes (Note: the concatenation process is instantaneous, since all the segment are already in the mp4 format. Thus, no encoding/decoding needs to be done and the files are just stitched together).

4. Video preview generation

Once we have the video segments and the playlist files, we can now generated additional files that enhance the viewing experience. After the transcoding and segmentation processes are complete, the program moves onto generate two image files: a poster image file to display before the user clicks on the play button in JW Player, and video preview images to show a thumbnail of the playing video above the seek bar.

- Poster Image

  Generating the poster image is fairly straightforward; the ffmpeg command seeks to a random part of the video and grabs the first frame at that duration and uses it as the poster.

- Preview Images

  Generating the preview images requires extracting the last frame of every 5 second portion of the video. Once the images are extracted, a index file (`.vtt file`) is created to tell JW Player which image to display at which portion of the video. Ffmpeg doesn't generate this file automatically and hence it must be generated by our program manually.

  One issue faced during this process was that the process was producing too many image files and it could cause problems when going large scale. Hence, instead of producing a large number of images, one image was produced with the different preview images stitched together to produce one image strip. The index file has to be altered to reflect this; `.vtt files` support image strips by allowing users to define the x and y coordinates of the top-left corner of the image.

Step-by-step

1. Using ffprobe, check if the input file has an audio stream. If it does, use ffprobe to extract both video and audio metadata, otherwise just extract video metadata.
2. Calculate output video bit rate, output audio bit rate and output frame rate.
3. Resolve the different resolutions of the output files.
4. Perform transcoding and segmenting. Simultaneously, create master.m3u8 file.
5. Perform preview and poster generation. Simultaneously, create the .vtt index file.

Data

Upon testing, it was discovered that probing takes a fixed amount of time regardless of the input file bit rate or duration – 0.5s for probing the audio stream and 0.5s for probing the video stream.

The relationship between input file duration and bit rate, and the time taken to perform various parts of the video processing can be seen below:

Input file duration (ms)	Input file bitrate (kbps)	Preview + poster (ms)	Transcode + segment (ms)	Total (ms)
61590	145	1658	13070	14728
43750	493	2028	16584	18612
57200	518	2722	21831	24553
1260200	567	30952	548155	579107
71730	1466	3946	33105	37051
358160	5724	31271	612045	643316
30090	5803	4610	31817	36427
72120	11394	6588	124441	131029
82640	12074	7339	140317	147656

To further illustrate the effect of changing bit rate and file duration, further testing was performed. The table below shows the effect of increasing file duration on the time taken to process a video. We can observe that the time taken to transcode and segment a file requires the same **proportion** of the input file duration (i.e. time is increasing at a constant rate) , whereas the time for poster generation is increasing at a decreasing rate (seen by the gradual decrease in the value of P+P Time/Input Time).

Input file duration (ms)	Input file bitrate (kbps)	Preview + poster (ms)	Transcode + segment (ms)	Total (ms)	P+P Time/Input time	T+S Time/Input Time
60000	542	2271	23789	26060	3.79%	39.65%
120000	477	3670	45219	48889	3.06%	37.68%
180000	501	5102	67876	72978	2.83%	37.71%
240000	475	6779	90283	97062	2.82%	37.62%
300000	498	7907	111260	119167	2.64%	37.09%
600000	583	15049	239498	254547	2.51%	39.92%
900000	590	22584	358066	380650	2.51%	39.79%
1200000	562	29697	481618	511315	2.47%	40.13%

The table below shows the effect of increasing bit rate on the time taken to process a video. We can observe that the time taken to transcode and segment a file is heavily influenced by the bit rate, seen by the increasing proportion of the input file duration, whereas the time for poster generation is near-about the same, indicating that it is not heavily influenced by a changing bit-rate.

Input file duration (ms)	Input file bitrate (kbps)	Preview + poster (ms)	Transcode + segment (ms)	Total (ms)	P+P Time/Input time	T+S Time/Input Time
71730	99	3713	24626	28339	5.18%	34.33%
71730	197	3966	28523	32489	5.53%	39.76%
71730	296	3853	33998	37851	5.37%	47.40%
71730	394	3948	43518	47466	5.50%	60.67%
71730	493	3962	44192	48154	5.52%	61.61%

Summary:

- Probing – Unaffected by file duration or file bit rate (Constant at 0.5s for each stream)
- Transcoding+Segmentation – Increases with both file duration and file bit rate
- Preview+Poster – Increases with file duration

Distributed, Parallel, Scalable Transcoding

The system described above works perfectly well for transcoding one file manually. But to incorporate this system into a large project, it needs to work faster and handle much, much more than single file. A potential system that can be used in large-scale operations is outlined below.

- Queuing Mechanism

  All incoming files will be queued in a queuing mechanism. The Master Node will pick the first file at the head of the queue to transcode and segment.

- Master Node

  The node that acts as a controller between the queue and the worker nodes. The master node will take the file at the head of the queue and store the it into the shared file space. It will then perform the probe part of the transcoding process described above and store the meta-data into a database that can be accessed by all the worker nodes. Once the probing is complete, the master node will delegate the transcoding and segmenting onto the worker nodes. While the worker nodes are processing their respective chunks of the file, the master node will perform the video preview generation, and store the generated files into the shared file space. Once all the nodes raise their respective flag, the master node will perform the combining process and produce the play lists and `master.m3u8`.

- Worker Nodes

  Nodes to perform the transcoding and segmentation. These nodes will work on their chunks of the input file and store the output in the shared file space. Once the node has completed its work, it will raise a flag to signal to the master node that it has completed its task. Since all the nodes will create separate play lists for their chunks of the file, they need to be combined to produce one play list for each resolution (done by the master node). All the generated files are stored in the shared file space.

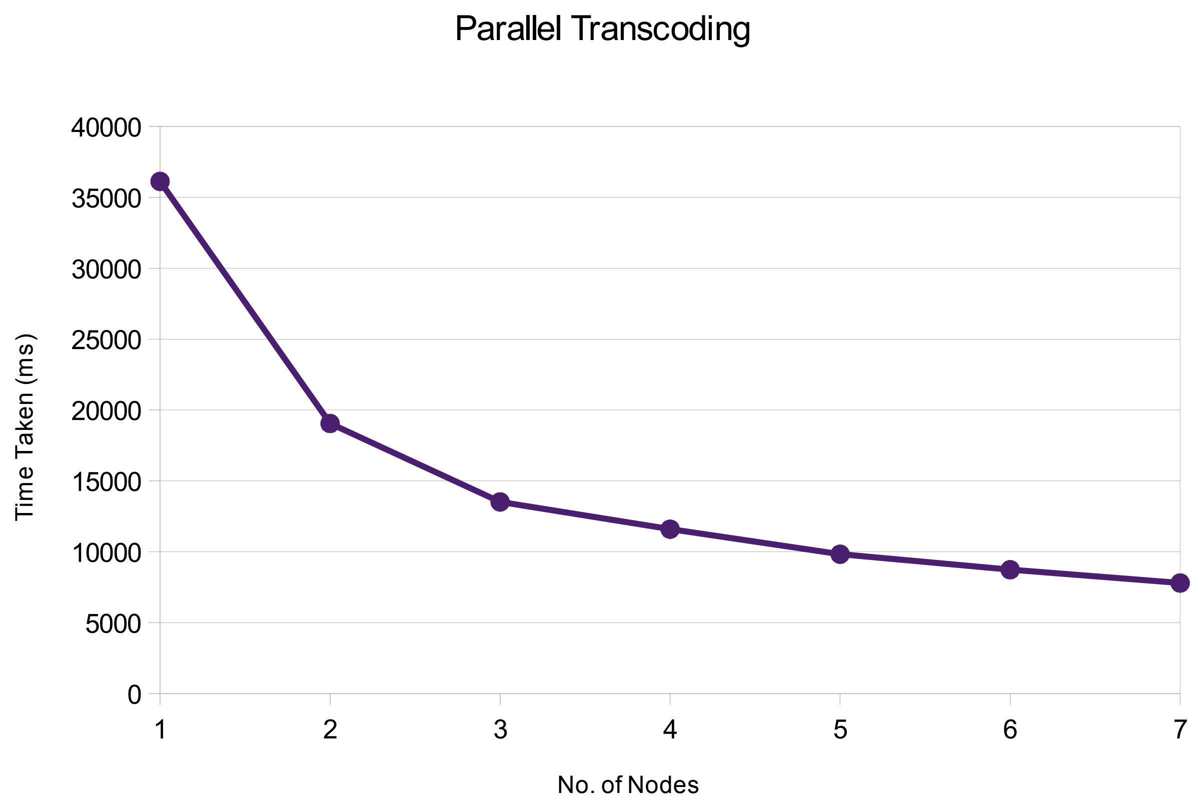
- Shared File Space

  A shared file space where all the input and output files can be stored.

<u>Sample run</u>

A sample run to illustrate the concept of parallel transcoding was done; the results are below:

No. of Nodes	Time Taken (ms)
1	36122
2	19049
3	13511
4	11591
5	9825
6	8731
7	7792



The same program described in the earlier sections was altered to support a distributed and scalable video processing solution.

Distributed Video Processing (v1.0)

The first version (v1.0) of the distributed processing program used 3 Linux servers, 1 acting as the master node and the other 2 acting as the worker nodes. There was no network file system (NFS) to function as the shared file space, so all the files were sourced and stored in the master node. This system also doesn't incorporate a queuing system and hence is only applicable for when a single file is to be processed. Although this doesn't really target the original problem of transcoding many files, it lays the foundation for the distribution and processing described above. The system worked like this:

1. Using ffprobe, the master node checks if the input file has an audio stream. If it does, ffprobe is used to extract both video and audio metadata, otherwise just video meta data is extracted .
2. The output video bit rate, output audio bit rate and output frame rate are calculated.
3. The master node deduces how the video is split up into chunks based on the number of worker nodes available. This is achieved by using the fast seek command (`-ss`) of ffmpeg. Technically, the process of creating 'chunks' is only virtual and the input file isn't broken into smaller files. Instead, the same input file is used by the worker nodes, and they begin processing their 'chunks' by seeking to some part of the file and then performing the transcoding and segmentation for a given duration. E.g. if 3 worker nodes are available to process a 3 min video, the first node seeks to 0 min (virtually no seek) and processes 1 min of the video. The second node seeks to the 1st minute and processes 1 min of the video. Similarly, the third node seeks to the 2nd minute and

processes 1 min of the video. The seek process is instantaneous and hence no extra processing time is required to do so.

4. The master node resolves the different resolutions of the output files. It also simultaneously, creates the master.m3u8 file.
5. The input files are sent to the worker nodes via Secure Copy (SCP).
6. Using SSH, the master node sends commands to worker nodes to begin transcoding and segmentation of their chunk. This is where the seek and 'chunk time' are added to indicate to each node where it needs to start and end it's processing.
7. While worker nodes are processing, the master node generates the preview and poster images, while simultaneously creating the .vtt index file.
8. Monitor the worker nodes, checking for when they finish their processing.
9. Once all processing is complete, the master node retrieves chunks and play lists from the worker nodes using SCP.
10. The different play list files are combined to generate one play list file for each resolution.

Issues

- A lot of time is wasted in sending and receiving files. This can be avoided easily by implementing a NFS.
- This system does not have any fault tolerance; if one of the worker nodes were to fail in doing its job, there would be no way to recover.
- The master node remains stationary (i.e. without any job) for a large portion of the processing. The master node could also be used in the processing, but a proper scheduling mechanism needs to be implemented. Also, this issue will probably not matter when a queue with a large number of files requires processing.
- Currently, the master node simply splits the input file into n number of chunks (where n is the number of nodes) and processes the video. It does not take into account the actual time it would take for the video to be transcoded and the number of videos waiting in the queue. E.g. if a queue contains two videos, the first being extremely long, and the other being extremely short; the shorter video would have to wait needlessly for the longer video to finish processing before it can be processed, even though the processing for the shorter video will be much, much faster. To avoid this, intelligence needs to be added into the master node to use the appropriate number of nodes for a video to empty the queue the fastest. In the above example, 1 node can be used for the shorter video to reduce the waiting time.