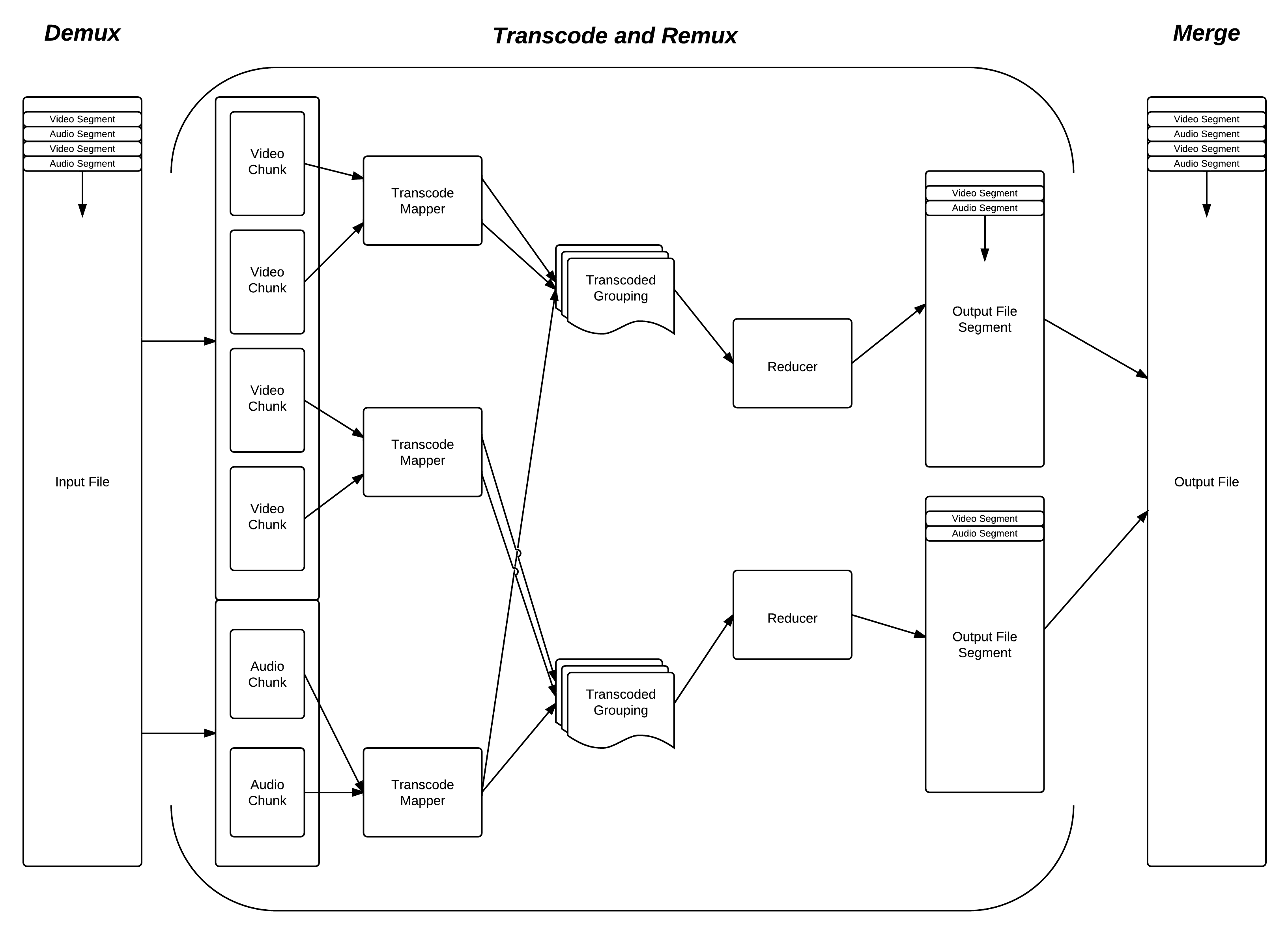
FFmpeg-MR Initial Design

# Overview



# High Level Phases

## Demux

The first stage of processing a transcode job is to ‘demux’ the file that will have been given as a source, into its constituent streams, so that we can place each stream into HDFS. We take this opportunity to calculate the best points in the streams for splitting them into chunks (video streams often reference other parts of the file, before and after the current frame to save space).

We output the data into HDFS using a Hadoop storage format called a SequenceFile. This allows us to store data in key value pairs that describe where all of the split points are, and also have the file be able to split across the HDFS file system using its built-in ‘sync points’. It also makes the data trivial to read in using SequenceFileInputFormat.

------

*We could skip this step, and just calculate the points in the video stream that we are OK to split and leave the data multiplexed in the container format we were given.*

*However, we do this de-multiplexing to make the overall operation simpler to understand as we pass data around, and to allow it to be easier for use to perform jobs that might duplicate streams (mostly audio streams – e.g. 6 channel AC3 to 2 channel AAC). This also helps us to make the C-JNI-Java API (see later sections) simpler as it doesn’t have to know about the context of the whole job, only the specific data it has to decode, and then encode into a new format in each map. Calculating the points in the file where there are multiple streams of video gets even more complicated (e.g. multi-angle DVDs) and this again is another reason for splitting the streams out.*

*At the end of the operation, if the task is sufficiently compute intensive (which should be the case if we are using ffmpeg-mr) then the extra work required to merge together the different parts of the streams into one (or more) files that are multiplexed is expected to be negligible.*

*ALSO – MUCH BETTER REASON: by doing this we don’t have to re-encode the split point GOPs or do anything too mad until will are already doing our encode in the Mappers. We would have to do some pretty nasty splitting to the input, so that we could just go and re-encode anyway (or do close, over egged splits, and then trim at the other end, but this would be fantastically hard to get right when we merged everything back together again… especially if we changed the frame rate!)*

----

Initially this phase will not be a MapReduce job and will run on the machine that the job is submitted to (the ‘task-tracker/name-node’). This may be turned into a MapReduce job if it turns out that extra performance gains can be made here (and we have the means to do a parallel read of the source file system efficiently – e.g Amazon S3 and not a normal file system).

## Transcode

This section forms the ‘Map’ of the MapReduce phase.

The MapReduce job will use SequenceFileInputFormat to read in the file that was generated that contains the split points for the original data. Map tasks match the splits that were generated when we placed the file into HDFS in the demux phase. These splits closely match the block size of HDFS (by default this is 64Mb, but we might go higher to help with performance). It will also try to ensure data-locality where possible (keeping the map task where the data is in HDFS).

Each map task will then read in the data from HDFS which will be passed across the JNI boundary to the custom C API which will then use libavcodec (and others), along with some data about what it is about to encode, and what format to encode to, to perform the actual transcode job on this chunk of data. The C code will pass the data back to java. We may also experiment with using JNI Pointer objects to improve performance if the Hadoop API can be retrofitted to understand them.

The map task will return a standard map-style key-value object, with the key as a combination of the timestamp that the data begins at, and the stream, and the value as the data.

This (along with all the other map task) will then be sorted into an order, and sent to a new worker for the reduce phase.

## (Shuffle/Sort)

We will use a custom Hadoop ‘partitioner’ that keeps the output from the map tasks sorted, rather than shuffling them on a hash of their key (not doing this would result in a randomly ordered video!).

This is the stage where internally Hadoop will allocate a number of the complete map tasks to reducers so that they can begin to remux the streams back into an output file.

## Remux

This section forms the ‘Reduce’ of the MapReduce phase.

Each reducer will receive an ordered list of ‘chunks’ of data (usually the HDFS block size or thereabouts) with a timestamp and stream ID to multiplex into the desired container format. It may receive many groups of these and should continue to output them to the same file in HDFS as they arrive.

The reducers will also use ffmpeg via JNI and a custom C API to pass data back and forth for processing.

## (Merge)

Depending on the number of reducers we use (this will be configurable so that we can see what has the best performance characteristics and why) we may need to take the chunked versions of the outputs of the Remux phase, and simple put them into one file. This will require further use of ffmpeg, JNI, and a custom C API to produce the final output file that will then be written to the output file system using Java.

With only one reducer, the final file will already be complete.

This file will move the output back to Amazon S3 if we are running in Elastic MapReduce, and may run concurrently with the job on the machine the job is submitted to so that it can begin to process the data as soon as the reducers finish.

## ffmpeg Interfaces

The ffmpeg interface is a custom C library written with a subset of the ffmpeg functionality, such that it is simple for the Java code to interact with. The library can be split up into 4 main parts: - file introspection, de-multiplexing, transcoding and multiplexing.

The file introspection API will be used to get container format/codec information about the file into Java, so it can be compared with the requested output. This is an initial pre-process stage that is used to calculate the actual execution plan to get our desired output. This might mean that we don’t have to re-encode the video as it is already in the correct format, or that we will keep one of the audio streams intact and simply add to it with a new one.

The de-multiplexing API will be used to take the actual raw file, and split it up into its raw streams. It will also return points that the stream can be split up safely (in the case where some points are unsafe – e.g. temporally compressed video). It will achieve this by either a custom method for specific codecs (like H.264), or by simply decompressing the stream using libavcodec and using its key\_frame flag.

The transcoding API will take raw stream data from HDFS through Java and convert it into a new stream that uses a different codec. It will require that it be told the original and desired stream types, as all header information will have been lost by this point. It will support video and audio through the same API. It will be a synchronous API where it accepts and correctly split array of bytes of data, and returns an array of the same length back to the caller, in the new format. Some performance optimisation will be attempted here in JNI to make the data pass across the boundary with the lowest overhead possible.

The multiplexing API will be used to put raw streams into the desired container format. It will need to know the target container, along with all of the stream types and their order, and then accept raw data from each in that order. The reducer will deal with ensuring this order is correct and that for each segment of data (that is a collection of streams that all start with the same timestamp and are of the same length) it has put all of the streams into the multiplexer before completing the segment. This in a similar way to the transcoding API will be synchronous and return the data that has been multiplexed back to the Java application as a direct buffer, which can then be flushed to HDFS by Hadoop.