co-oCCur

A high-speed subtitle synchronization tool

CCExtractor Development

Project Proposal

Google Summer of Code, 2019

TABLE OF CONTENTS

Basic Information

Name: Suyash Bajpai
Major: Electronics & Telecommunication Engineering
Institute: Government Engineering College, Raipur
Email: 7suyashbajpai@gmail.com
Github: @sypai
Slack: sypai
Phone: (+91) 7772800787
Postal Address: M.I.G - 62, Sector - 4
Pandit Deen Dayal Upadhyay Nagar
PIN - 492010
Raipur, Chhattisgarh, INDIA
Timezone: Indian Standard Time (UTC +5:30)

---

co-oCCur

AIM

To build tools for the high-speed synchronization of subtitles with the audiovisual content.

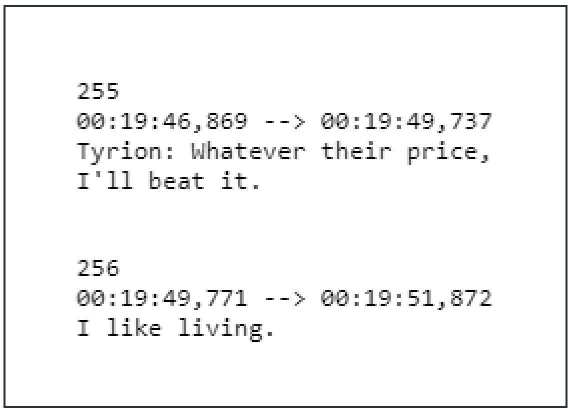
- **Tool A**: Synchronization of subtitles between two versions of the same audiovisual content, using audio fingerprinting annotations.
- **Tool B**: Synchronization of subtitles, using the *burned-in* subtitles of the base audiovisual content and a modified version of the base content, comparing the timing window (constructing two binary strings) of the modified audio and the subtitles.

INTRODUCTION

Closed Captions (CC) and subtitles enhance audiovisual content by providing speech information and description of representative events in a textual format. Captioning is especially used as an aid for people with hearing loss or deafness, but its use is definitely not limited to that domain. For instance, it is frequently the case that captions are necessary to watch a TV show in a noisy surrounding or when one is not familiar with the language or accent available in the audio streams.

The most popular subtitle file format, the SRT (*SubRip Text*) consists of:

1. A number that corresponds to the subtitle in the sequence
2. Time codes that correlate to a subtitle appearing and disappearing
3. Textual payload, the actual subtitle
4. A blank line to show the beginning of a new subtitle



```
255
00:19:46,869 --> 00:19:49,737
Tyrion: Whatever their price,
I'll beat it.


256
00:19:49,771 --> 00:19:51,872
I like living.
```

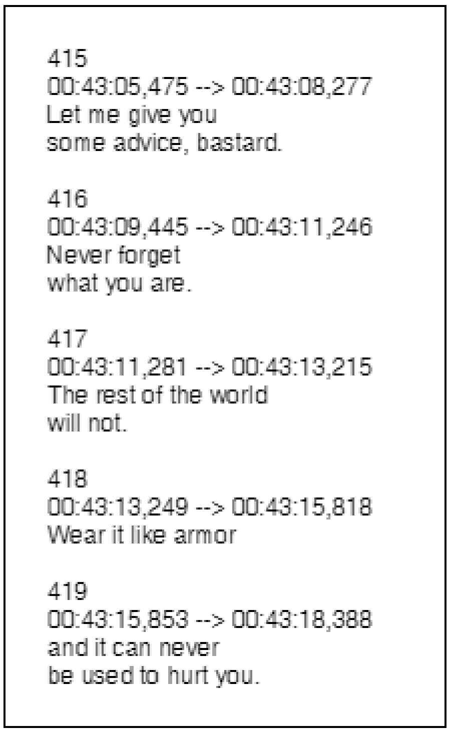
In this example, dialogue #255 appears at **`00:19:46`** and remains on the screen for three seconds and disappears at **`00:19:49`**.

For an ideal subtitle file, the subtitles are perfectly aligned with the base audiovisual content. In other words, the audio and the corresponding subtitle **co-oCCUr**.

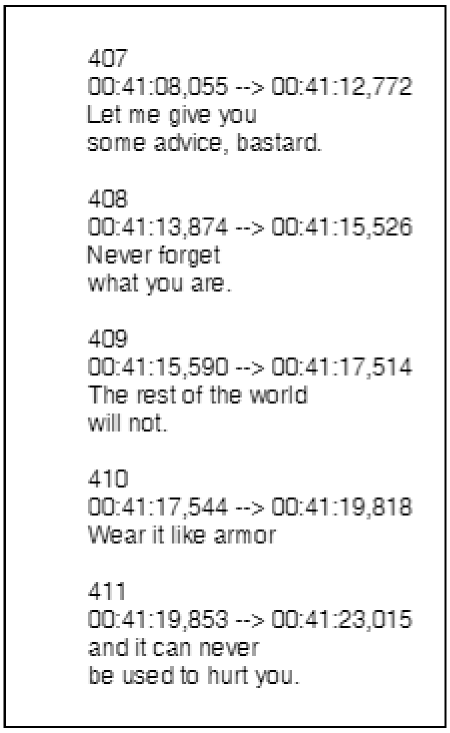
The subtitles and captions we use, originate from different sources. Their lifecycle can be analyzed from different perspectives.
In professional post-production, there are special teams working on the creation of subtitles alongside the audiovisual content. These subtitles are *burned-in* in a container format with other data streams and then distributed. These subtitles are created in a very controlled workflow, major synchronization problems, are not expected during consumptions.

Alternatively, there are some non-professional enthusiasts who put in efforts for the benefit of the rest of us and create subtitle files, mostly for online TV series' and movies. These subtitles find a home in online databases in a vast number of languages. The subtitle documents present online are often distributed separately from the reference audiovisual content and multiple versions are available for the same audiovisual material. Thus users may face difficulties in finding the version that offers satisfactory synchronization. The existence of multifarious versions of subtitle documents may reflect the presence of various editions of the original audiovisual content that may or may not include an opening intro, scenes from previous or next episodes, commercials, different frame settings and so on.

Consequently, the caption entries may be displayed on the screen earlier or later than the corresponding audio and video events. Even if the temporal offset is of a couple of seconds, misalignment of the caption entries will cause a sustained exasperation to the user.



```
415
00:43:05,475 --> 00:43:08,277
Let me give you
some advice, bastard.

416
00:43:09,445 --> 00:43:11,246
Never forget
what you are.

417
00:43:11,281 --> 00:43:13,215
The rest of the world
will not.

418
00:43:13,249 --> 00:43:15,818
Wear it like armor

419
00:43:15,853 --> 00:43:18,388
and it can never
be used to hurt you.
```



```
407
00:41:08,055 --> 00:41:12,772
Let me give you
some advice, bastard.

408
00:41:13,874 --> 00:41:15,526
Never forget
what you are.

409
00:41:15,590 --> 00:41:17,514
The rest of the world
will not.

410
00:41:17,544 --> 00:41:19,818
Wear it like armor

411
00:41:19,853 --> 00:41:23,015
and it can never
be used to hurt you.
```

The above figure shows parts of two different subtitle files of the same episode of *Game of Thrones*, differing in timestamps and duration, for the same caption entries. The one on the left is the synchronized subtitle document.

Thus the misalignment of the subtitle files is the underlying problem that this project aims to solve so that the viewer does not have any burden before the *fun* starts (this is what matters).

THE “WHY” BEHIND IT ALL?

Why “CCExtractor Development”?

Being a non-native English speaker, I remember how subtitles were the only way for me to watch movies and when I look back subtitles have helped me improve my comprehension skills in English.

“Magical” appearance of subtitles on the screen had always fascinated me and I always wanted to know how this happens.

I came across CCExtractor Development while going through the GSoC archive and voila! I knew this is what I was looking for. Delving in, I am amazed by the journey of CCExtractor from a personal-project to something being used by the world as the standard tool for everything related to subtitles. I cannot think of a better platform than “CCExtractor” to spend my summer with. The project is engrossing and who to better implement it with than the “magicians” behind the de-facto subtitle tool.

Moreover, right from the first “Hey!”, the CCExtractor community has been exceedingly helpful. Have a query, the answer is just a slack message away!

I would love to be a part of CCExtractor Development, in GSoC and in future, and work with them towards their goal “*CCExtractor is to subtitles what ffmpeg is to video: The only tool you'll ever need*”.

Why “co-oCCur”?

Downloading a subtitle document and playing it alongside an episode of your favourite TV show, is not rocket science, but it sure can feel that way sometimes. Getting the subtitle document that gives satisfactory synchronization on the first attempt is like *hitting the jackpot.* I know the annoyance that comes in with misaligned subtitles and is a very general problem.
Working on a “high-speed synchronization tool” that can solve the sync problem is a perfect choice to spend a summer on. To be able to implement it myself will provide me with great learning opportunities that I can get nowhere else.

The project requires getting acquainted with various techniques, technologies and libraries that will help me gain a diverse set of skills.
The project involves “audio fingerprinting” which blows away my mind. The technique as a whole and the various tools that go behind the fingerprinting algorithm make the perfect match. It involves Digital Signal Processing which is a part of my majors in Electronics & Telecommunication.

PROPOSED DELIVERABLES

1. co-oCCur, A high-speed subtitle synchronization tool with following functionalities:
    a. Tool A
    b. Tool B
2. Dactylogram: An audio fingerprinting library
3. Subtitle Parser & Subtitle Editor
4. Setup scripts
5. A minimal command line interface.
6. Tests for various logical units.
7. Detailed Documentation.
8. Fortnightly blog posts on developmental advances of the project.

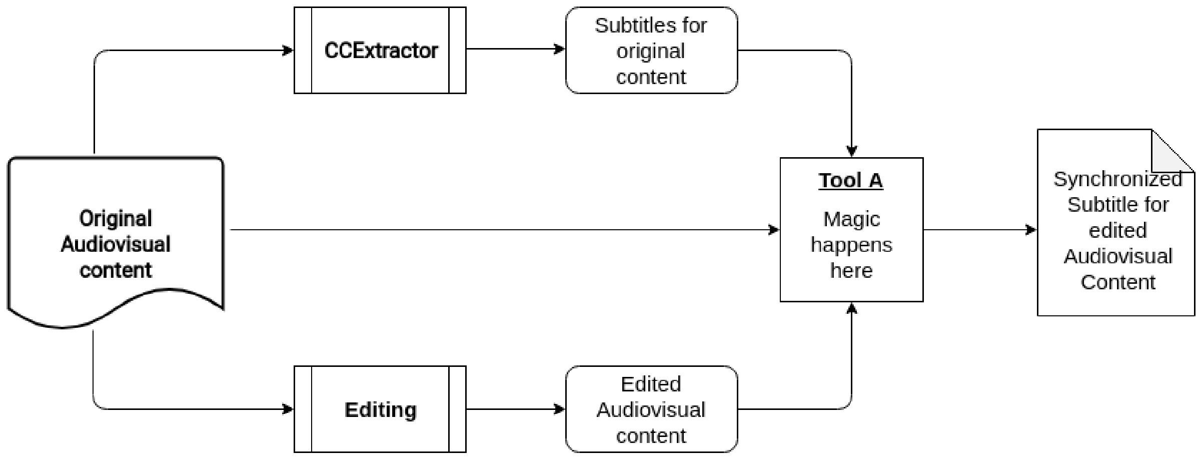
BRIEF WORKING

The **Aim** and the **Introduction** sections of this proposal have set the stage, clearly elucidating the problem that this project aims to solve. The project **co-oCCur** will comprise of two tools, tackling different cases.

Some subtitle documents are extracted directly from DVDs and Blu-Rays, whereas others are created using audiovisual content recorded from broadcast TV. The most common misalignment problem between different subtitle documents is a ***constant temporal offset*** that produces an equal shift of the caption entries.
This is the case, for instance, if one subtitle document is generated using an audiovisual content including commercials, whereas another document is created using a version of the same content but without any commercials.

The working of the tools is briefly explained below which gives an outline of how the two tools will be used to solve the problems of misalignment.

Tool A
This tool will be used to synchronize subtitles using audio fingerprinting annotations.


CCExtractor
Subtitles for original content
Original Audiovisual content
Tool A
Magic happens here
Synchronized Subtitle for edited Audiovisual Content
Editing
Edited Audiovisual content

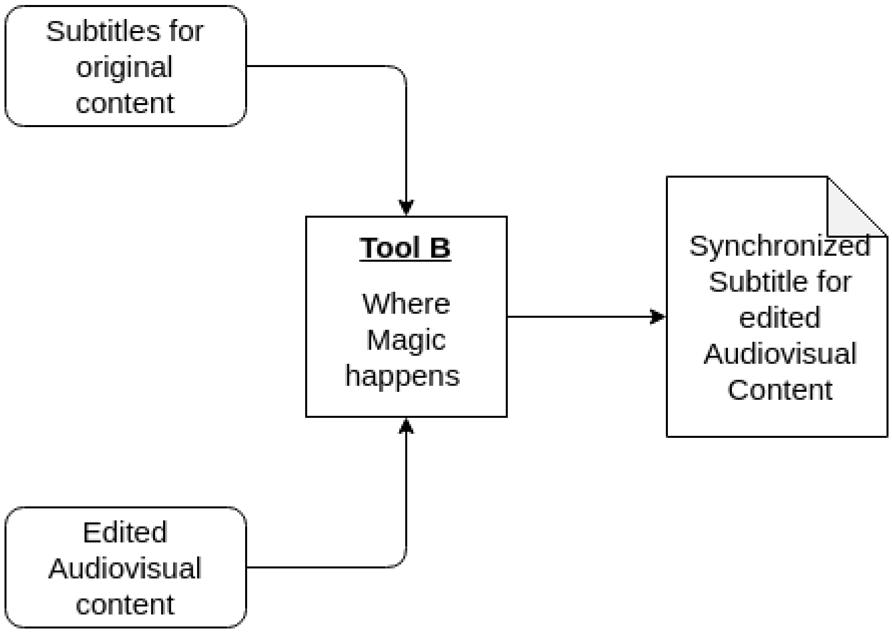
It will take as input the original audiovisual content, the edited audiovisual content and the subtitles document of the original audiovisual content extracted using **CCExtractor**. It will produce as output, an edited subtitle document which will **co-oCCur** with the edited version of the audiovisual content.
At a superficial level, the workflow of this tool will consist of the following steps:

1. Extraction of audio from the original audiovisual content using FFmpeg.
2. Extraction of subtitles from the original video using CCextractor.
3. Extraction of fingerprints anchors from the audio obtained from (1).
4. Insertion of fingerprint anchors extracted from (3) into the subtitle file obtained from (2) at the corresponding timestamps.
5. Extraction of audio from the modified audiovisual content using FFmpeg.
6. Calculate audio fingerprints from the modified audio (5) at the offsets where fingerprint anchors have been inserted into subtitle document.
7. Comparing audio fingerprints of the modified audio (6) with the fingerprint anchors from subtitle document (4).

```
if (calculated fingerprints match reference fingerprints):
    Subtitles co-oCCur with the modified media file.
else:
    Subtitle entries are dynamically adjusted to co-oCCur with
    the modified audiovisual content.
```

Tool B

This tool will match audio to the corresponding subtitles using the timings of both.


Subtitles for original content
Tool B
Where Magic happens
Synchronized Subtitle for edited Audiovisual Content
Edited Audiovisual content

Since the media file has been modified the timing of each subtitle entry needs to be changed in order to synchronize with the corresponding audio and video events. It will take as input the modified audiovisual content and the subtitle document for the original audiovisual content. It will produce as output, an edited subtitle document which will **co-oCCur** with the edited version of the audiovisual content.

Tool B will work in steps as shown below:

1. Extraction of audio from the original audiovisual content using FFmpeg.
2. Break the extracted audio into very small frames.
3. Parsing the subtitle file and discretizing by time into small frames of equal windows as audio.
4. Using Voice Activity Detection in the frames obtained from (2) and forming a binary string.
5. Using a subtitle parser to check if there is a subtitle entry present or not, thus another binary string obtained.

   ‘1’: Audio/subtitle is present in the frame
   ‘0’: No audio/subtitle in the frame
6. Align the two binary strings by matching 0’s with 0’s and 1’s with 1’s.
7. We score these alignments and the best-scoring alignment decides the offset that has to be propagated to all subtitle entries, for them to co-oCCur with the modified video.

DETAILED WORKING

co-oCCur as has been mentioned will consist of two tools:
Tool A and **Tool B.**
As is the parent program, **CCExtractor**, this project will also be entirely developed using C++.

Tool A

Synchronization of subtitles between two versions of the same audiovisual content. This tool will consist of two modules:

1. Enhancing the original subtitle document by inserting representative audio fingerprint annotations extracted from the original audio.
2. Adjustment of misaligned subtitle entries based on the comparison of audio fingerprints of modified audio at the offset of anchors present in enhanced subtitle file obtained from (1).

The input to this tool is:

A. The original audiovisual content (`#in_video`)
B. The subtitles for the original content extracted using CCExtractor (`#in_CC`)
C. The modified audiovisual content (`#edited_video`)

Audio Extraction

FFmpeg will be used to extract audio files from audiovisual contents. For further processing of the audio files, the audio needs to be uncompressed, raw PCM (16-bit signed int). It should be mono (single channel) sampled at 16 KHz (enough to cover the human speech frequency range).

Using FFmpeg it can be done by just a single command:

```
ffmpeg -i #in_video -acodec pcm_s16le -ac 1 -ar 16000 #in_audio.wav
```

- -i : input;
- -acodec : stream;
- -ac : set number of audio channels
- -ar : set audio sampling rate (in Hz)

Outcome:

`#in_video` --> `#in_audio`

`#edited_video` --> `#edited_audio`

Thus we will have two audio files extracted.

1) Enhancement of the Subtitle document

The first module in this tool deals with the enhancement of the original subtitle document by inserting into it representative audio fingerprints.

Audio fingerprint Generation (AFExtractor in the workflow)

An **audio fingerprint** (also referred to as an acoustic fingerprint) is a compact representation of some audio that encapsulates information that is specific to the audio that it represents. The role of an audio fingerprint is to capture the signature of a piece of sound, such as a song, that allows it to be differentiated from other sounds.

This sub-module will be focused on the development of audio fingerprint generating library. I will use dejavu by willdrevo's and this article as a reference for the development of ***Dactylogram*** (that's what we'll call it).

- Audio fingerprints returned from this library would be represented either as **base64 encoded strings** or **32-bit integer arrays**.

  The **32-bit integer** arrays are also called "raw fingerprints" and they represent the internal structure of the fingerprints. This is what we will get as output. For the correlation between two audio signatures, while adjusting subtitle later, this is the form which will be used.

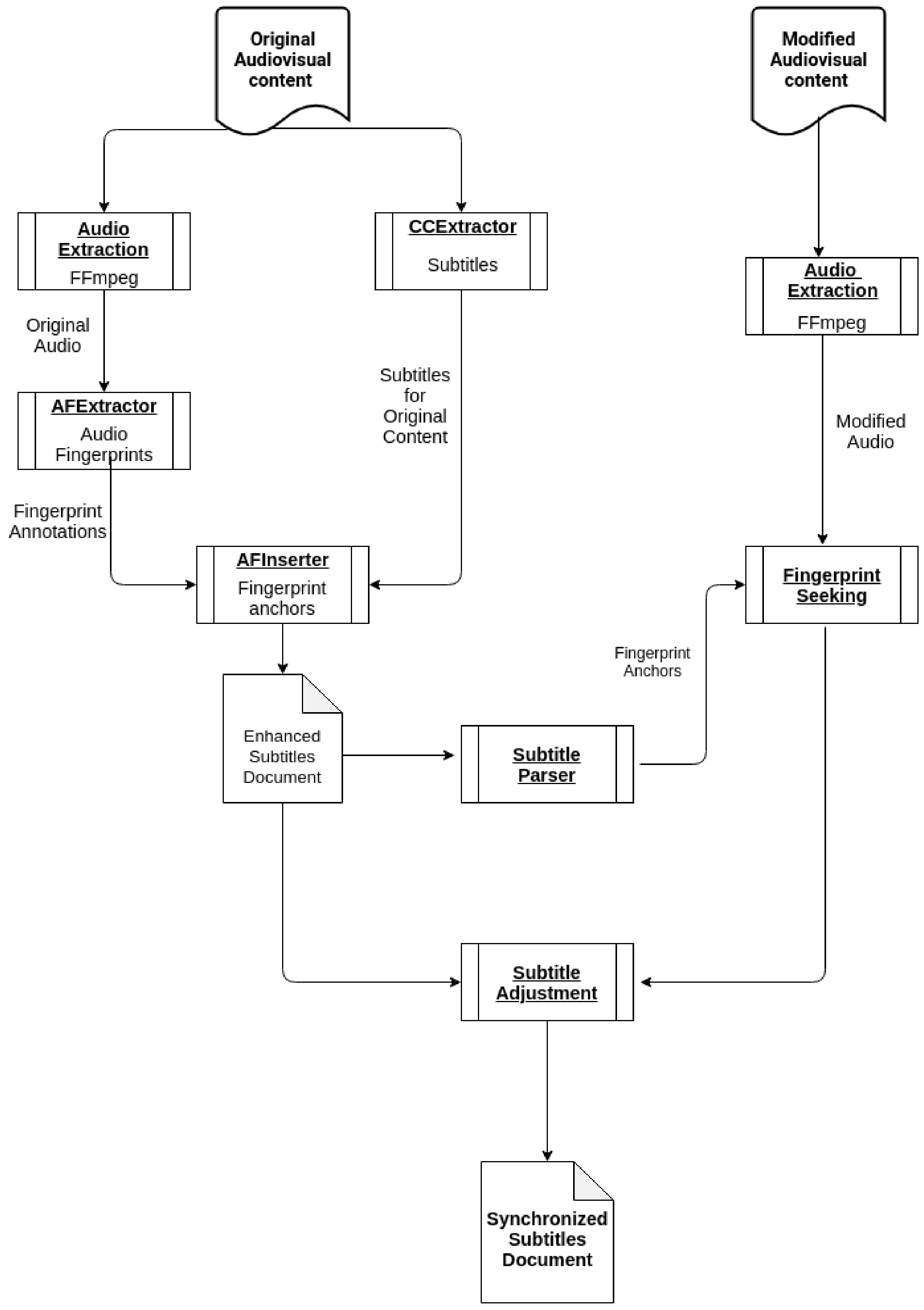
  However, for storing the information on the subtitle document we will use the **base64 encoded string** form.

  The base64 encryption would not be much of an issue, something like this(Note: External project with no personal involvement) can be done.

Outcome: Audio fingerprint anchors

Workflow

Tool A can be clearly understood from the following figure:



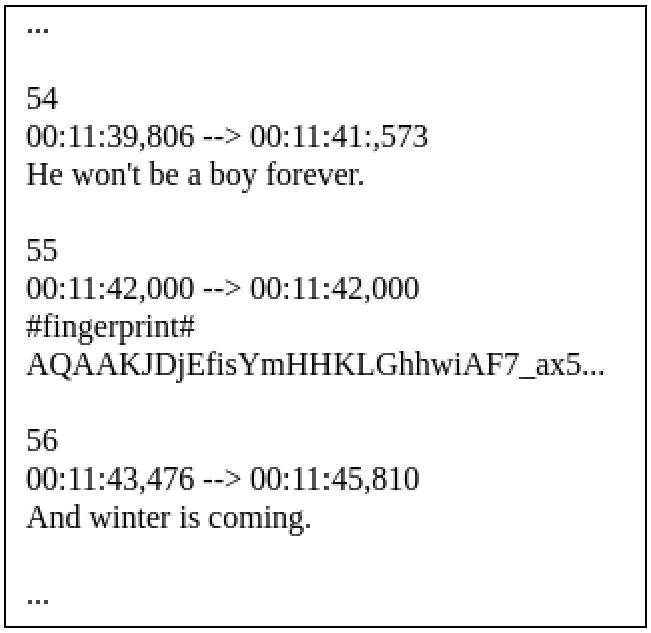
Fingerprint Extractor (AFExtractor in the workflow)

The first step consists of electing subtitle entries as synchronization anchors. To encompass the most possible cases of misalignments, our tool will make use of **3** synchronization anchors.
As a guideline, ToolA will extract 3 audio fingerprints preferably near the beginning, near the middle and close to the end of the audio signal (`#in_audio)` that describe the main audio components of these anchors.

Note: The audio fingerprint annotations do not necessarily need to be associated with an actual speech event.

Fingerprint Insertion (AFInserter in the workflow)

This sub-module will carry out the task of adding the audio fingerprint anchors to the original subtitle file (`#in_CC`) with their corresponding offset time within the audiovisual content. They will have zero duration.



```
...

54
00:11:39,806 --> 00:11:41:,573
He won't be a boy forever.

55
00:11:42,000 --> 00:11:42,000
#fingerprint#
AQAAKJDjEfisYmHHKLGhhwiAF7_ax5...

56
00:11:43,476 --> 00:11:45,810
And winter is coming.

...
```

The keyword ***#fingerprint#*** will indicate that an audio signature is present.

As already mentioned, while storing the fingerprints in the file **base64** string will be used. This includes a quantitative analysis of the size of fingerprints (Not my work). "Average size of one encoded audio fingerprint was 247 ASCII characters. Therefore when we consider all 3 fingerprints captured along with the text that describes their zero-duration timestamps and sequence numbers, the average overhead added is around 890 characters. To put in perspective, in a regular movie with about 1000 caption entries, these annotations would represent just about 1% of the file size."

Outcome: "Enhanced subtitle document" with representative audio fingerprint annotations.

`#in_CC` --> `#rich_CC`

2) Adjustment of the Subtitle entries

The second module in Tool A which will be responsible for the adjustment of the subtitle entries with the corresponding audio and video events in the modified audiovisual content (`#edited_video`).

Subtitle Parser

After the audio has been processed and fingerprints extracted and the enhancement of the subtitle file is also done, the next step is to parse the enhanced subtitle file. (`#rich_CC`)
This sub-module will parse it from where subtitle entries and audio fingerprint annotations with their offset time in the audiovisual content are read.

This is an excellent subtitle parser which was developed by @saurabhshri for his project in GSoC'17 (no personal involvement), which will fit in nicely in our plan of things.

```
std::vector<std::string> payload = sub->getText();
std::vector<std::string> fp = sub->getNonDialogueWords();
```

It can be used to extract and strip texts, speaker names and non-dialogue texts. Our requirement is to fetch ***#fingerprint#*** from the enhanced subtitle file. The offset time and the payload i.e. base64 encoded fingerprint can be easily pulled out.

Outcome: A Fingerprint array, a timestamp array indicating where the fingerprints start.

`#fingerprints[]`

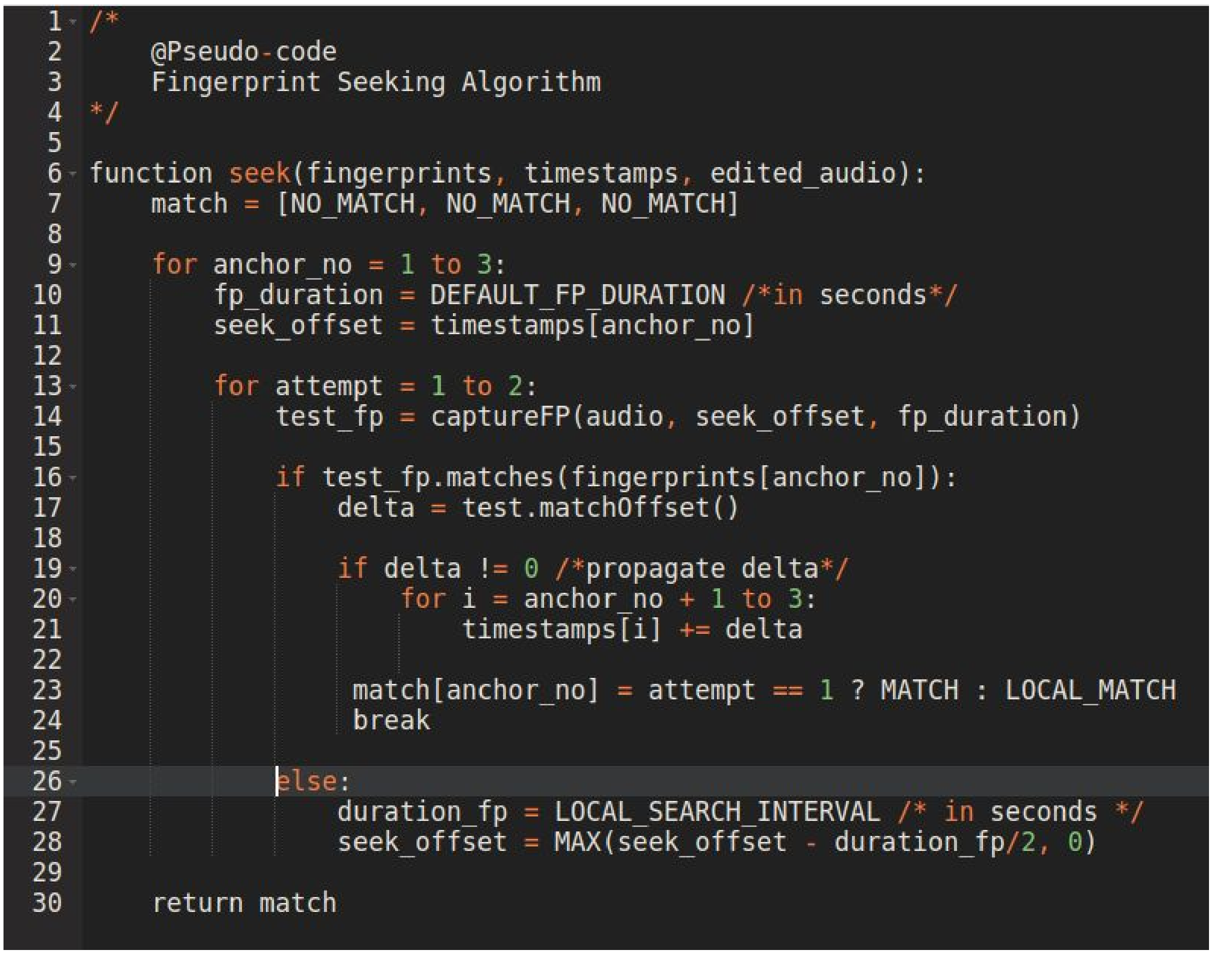
`#timestamps[]`

Seeking & Comparing

In the order of things, next comes the part where we seek the fingerprints in the modified audio and compare them to the fingerprints of the original audio.

After parsing the enhanced subtitle file we get the fingerprint array and their timestamps' array.

The fingerprints of the `#in_audio` will act as anchors, using them we will **SEEK** the fingerprints in the `#edited_audio` at the corresponding offset time from timestamps array and then **COMPARE**.



```
/*
    @Pseudo-code
    Fingerprint Seeking Algorithm
*/

function seek(fingerprints, timestamps, edited_audio):
    match = [NO_MATCH, NO_MATCH, NO_MATCH]

    for anchor_no = 1 to 3:
        fp_duration = DEFAULT_FP_DURATION /*in seconds*/
        seek_offset = timestamps[anchor_no]

        for attempt = 1 to 2:
            test_fp = captureFP(audio, seek_offset, fp_duration)

            if test_fp.matches(fingerprints[anchor_no]):
                delta = test.matchOffset()

                if delta != 0 /*propagate delta*/
                    for i = anchor_no + 1 to 3:
                        timestamps[i] += delta

                 match[anchor_no] = attempt == 1 ? MATCH : LOCAL_MATCH
                 break

            else:
                duration_fp = LOCAL_SEARCH_INTERVAL /* in seconds */
                seek_offset = MAX(seek_offset - duration_fp/2, 0)

    return match
```

Pseudo-code is taken from the paper referenced below (NO personal involvement) with some modifications.

The algorithm seeks the synchronization anchors (denoted as `#fingerprints[]` array of length 3 and it's associated `#timestamps[]`

indicating where the fingerprint start) in the audio extracted from the modified video i.e. `#edited_audio`.

The algorithm will result in an array containing 3 possible values for each fingerprint comparison: NO_MATCH, MATCH or LOCAL_MATCH.

The algorithm starts by seeking to the exact time offset obtained from `#timestamps[]` and generates an audio fingerprint from `#edited_audio` (line 14).
If the two fingerprints, match at the very first attempt (line 16), the subtitle document `#in_CC` is indeed aligned with `#edited_video` which is very rare and only possible if the modifications in the original audiovisual content (`#in_video`) are negligible (delta variable is set to zero in line 17). The algorithm then changes `match[anchor_no]` with MATCH (line 23) and terminates the inner loop proceeding to the next anchor.

When the audio fingerprint annotation does not match the calculated fingerprint at the expected offset, the algorithm increases the size of the search window (lines 27 and 28) and performs a new search (`attempt` variable becomes 2). This new search will be called "*`local search`*", which will consist of generation of all audio fingerprints in a predetermined time frame around the given offset and comparing each one of them with our referred audio fingerprint.

In cases where parts from previous and/or next episodes are removed, we will capture audio fingerprints using a different time length every time we do not find a direct match and thus the search window will be broader. If the algorithm finds a match, `match[anchor_no]` is updated with LOCAL_MATCH (line 23). Otherwise, `match[anchor_no]` keeps the initial value (NO_MATCH). The value of the variable `delta` is then calculated as:

```
delta = offset specified in the synchronization anchor
      - offset where the match is found through local search
```

This delta is taken into account in the verification of the coming anchors (lines 19 to 21). Therefore, even if the first synchronization anchor is found through a local search, the subsequent anchors can still match perfectly.

Outcome: `delta`, the constant temporal offset which is what we wanted.

Subtitle Adjustment

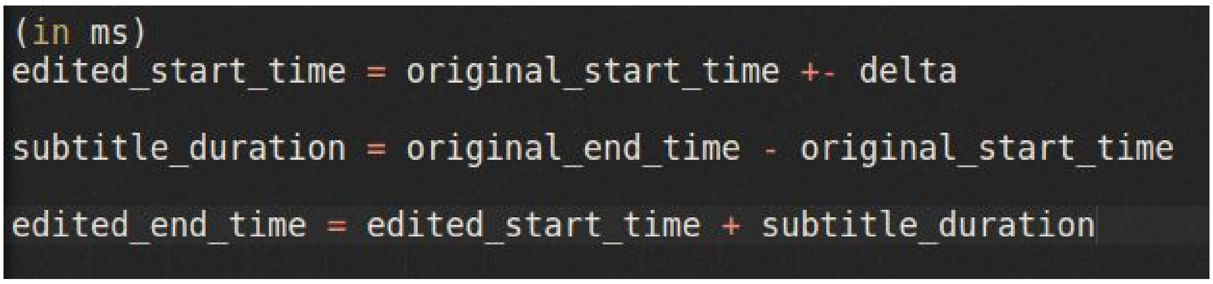
This is the last sub-module.

The adjustment action consists in propagating the constant temporal offset, the value of the variable `delta` obtained from the previous sub-module, to all the subtitle entries in the original subtitle document `#in_CC`.

A simple subtitle editor that will change the timings of each subtitle entry will be needed. Standing on the shoulders of saurabhshri's subtitle parser, a subtitle editor will be developed (Something like this).

When the subtitle file is parsed, starting from the very first subtitle entry, the value of `delta` will have to be added to the starting time of the entries. Based on the time duration of the subtitle entries, the end time of each entry will have to be calculated.

For a subtitle entry,



```
(in ms)
edited_start_time = original_start_time +- delta

subtitle_duration = original_end_time - original_start_time

edited_end_time = edited_start_time + subtitle_duration
```

Repeating this for each of the subtitle entries in the original subtitle file `#in_CC`, the final output of **Tool A** will be an edited subtitle file `#out_CC`.

Outcome: A synchronized subtitle document.

Thus, **Tool A** will create a subtitle document (`#out_CC`) that will **co-oCCur** with the modified audiovisual content (`#edited_video`).

Tool B

The first question that arises is:
Why **Tool B**? What is the need for two different tools? Isn't **Tool A** enough for synchronizing subtitles with a video?
The answer to this lies in the input that is being given to both them and is very simple to understand. **Tool A** uses audio fingerprint annotations from the original audio which are then compared against the modified audio. While **Tool B** is proposed to be used in cases where the original audiovisual content is not present and hence a different approach is required.

Again, this tool will consist of two modules:

1. Construction of audio and subtitle binary strings.
2. Alignment of the binary strings

The input to this tool is:

A. The subtitles for the original content extracted using CCExtractor (`#in_CC`)
B. The modified audiovisual content (`#edited_video`)

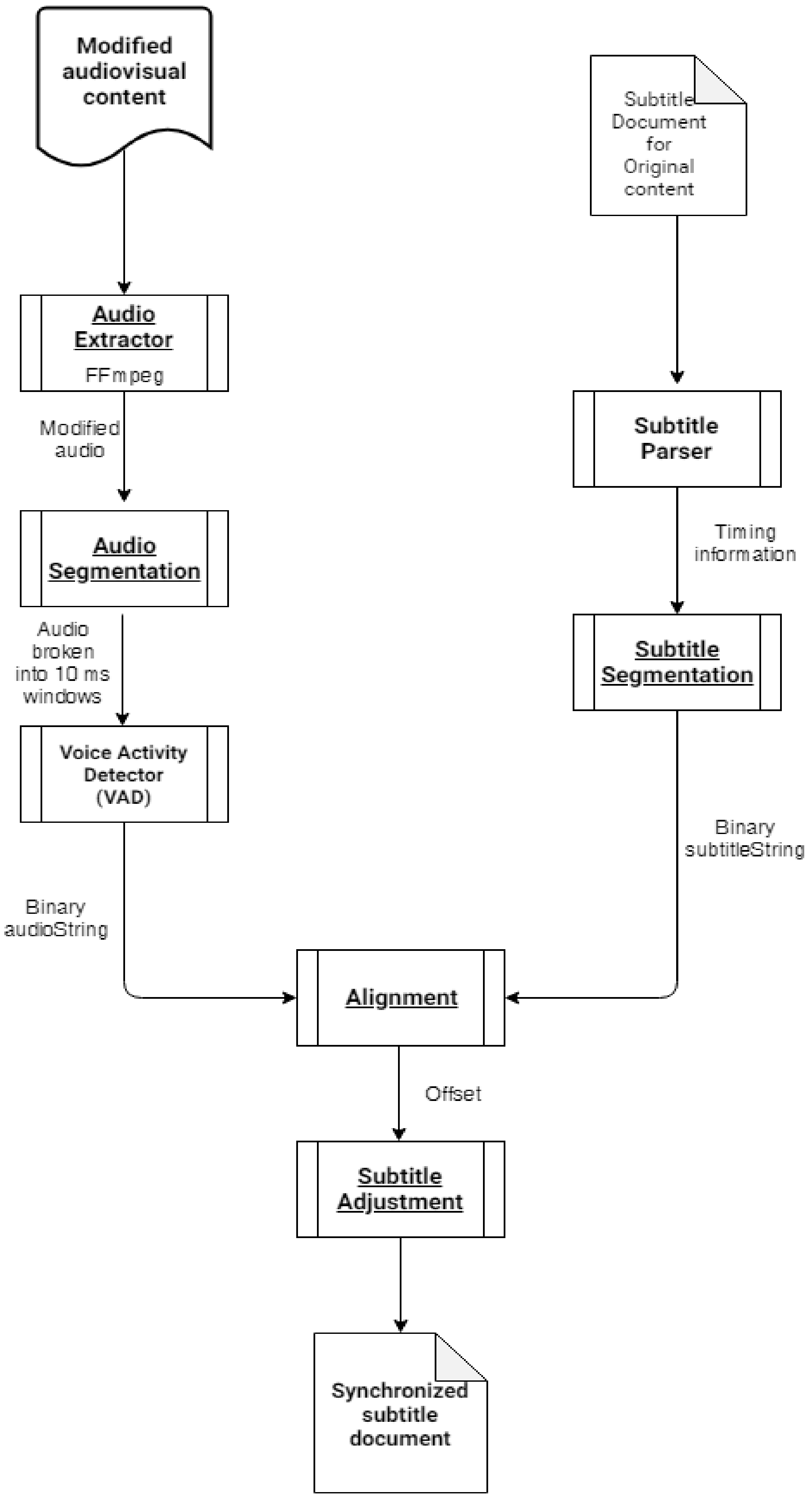
Audio Extraction

Audio will be extracted from the modified audiovisual content in the same way as has been mentioned here in Tool A.

`#edited_video` --> `#edited_audio`

Workflow

The following figure explains the overall workflow of **Tool B**:

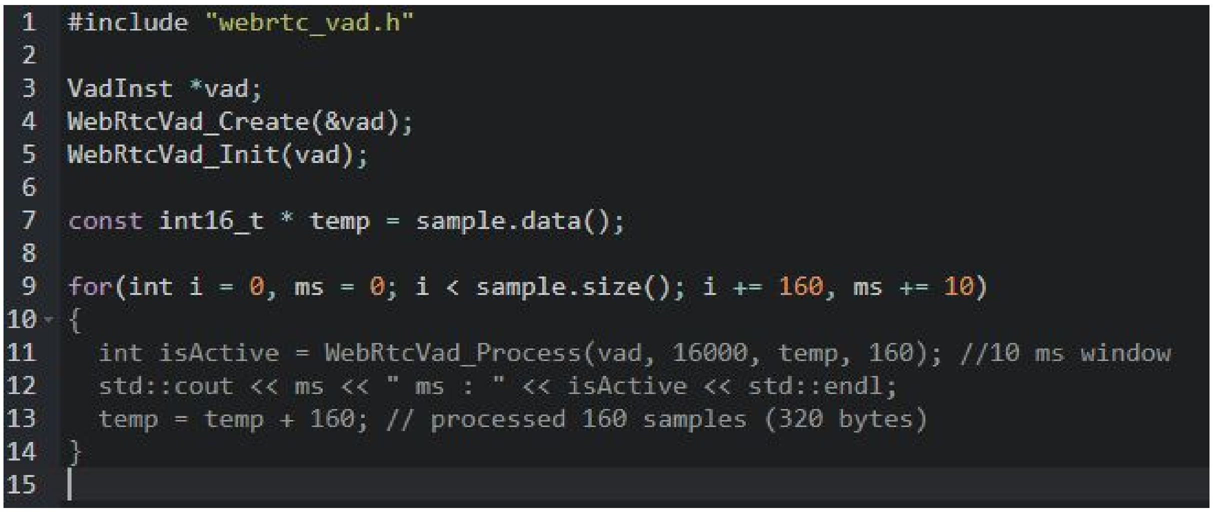


1) Construction of audio and subtitle binary strings

Audio Segmentation & Voice Activity Detection (VAD)

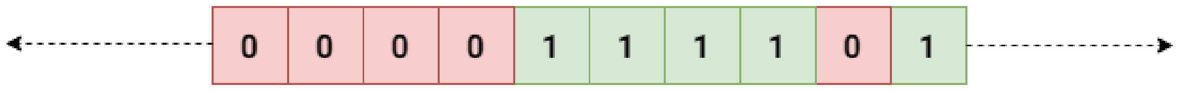
A Voice Activity Detector (VAD) is used to identify speech presence or speech absence in audio. If a frame has some audio activity present in it will result in '1' otherwise '0'.
The extracted audio is discretized by time into small frames (10 ms windows). Using VAD each frame is analyzed in order to check if audio is present in the frame or not.
Combining the results of each frame a binary string is constructed.

The implementation of VAD will be done using WebRTC. This requires the audio to be uncompressed, raw PCM (16-bit signed int). It should be mono (single channel) sampled at 16 KHz. Thus `#edited_audio` will be passed to the VAD.
Using this answer, something like this will be done



```cpp
#include "webrtc_vad.h"

VadInst *vad;
WebRtcVad_Create(&vad);
WebRtcVad_Init(vad);

const int16_t * temp = sample.data();

for(int i = 0, ms = 0; i < sample.size(); i += 160, ms += 10)
{
  int isActive = WebRtcVad_Process(vad, 16000, temp, 160); //10 ms window
  std::cout << ms << " ms : " << isActive << std::endl;
  temp = temp + 160; // processed 160 samples (320 bytes)
}

```

Outcome: A binary string representing the presence of audio `#audioString`



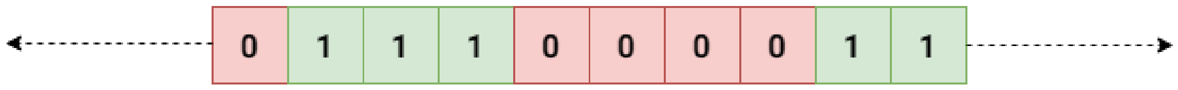
Subtitle Parser

It has been already explained in this section. The original subtitle file `#in_CC` will be parsed. For this tool, we will need timing information of the subtitle file. The subtitle parser will have to be modified to extract the subtitle information in milliseconds.

Subtitle Segmentation

The next task is the creation of a binary string that will hold the information about subtitles, in the same way, the audio has been segmented. In this case, finding the presence or absence of text in the 10ms windows is trivial. The subtitle file will be checked for the timing windows. The binary string will have a '1' in places where textual entries are available and in case there is no subtitle it will hold a '0'.

Outcome: A binary string representing the presence of subtitles `#subtitleString`



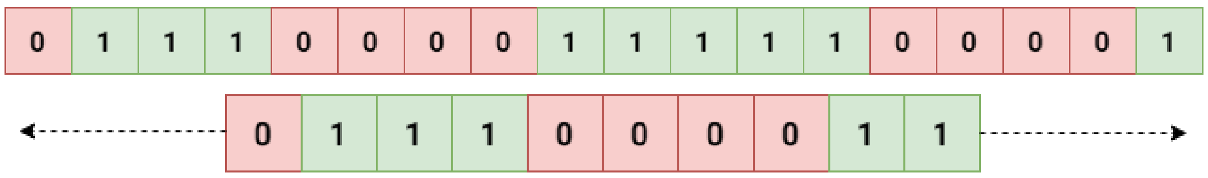
2) Alignment of binary strings

Alignment

This is the sub-module which will actually work on the synchronization problem. After all the previous steps we have with us the `#audioString` and the `#subtitleString` each is a vector indexed by time.

The problem of misalignment can be seen as: we have two binary strings, and we want to offset one of them so that they match up the best.

We can imagine padding the `#subtitleString` out on either side with 0's. The alignment task can be considered as shifting the `#subtitleString` left and right in time until we obtain the "best alignment" between the 1's of `#audioString` and `#subtitleString`.



As the binary strings will be constructed after segmenting audio into 10ms frames, these strings will be immensely long (millions of digits for audio longer than an hour). Our alignment algorithm, in simple terms, for each offset, will take a dot product of one string with the offset version of the other. Computing this naively would result in an **O(n^2)** solution. And "High speed is really a priority" is clearly mentioned in the project's idea page. So, we will use the Fast Fourier Transform (FFT), bringing the complexity down to **O(n log n)**.

We will score the alignment based on the number of matching 1's i.e. the summation at all values of time of the product of both the strings,

$$\#audioString(t) * \#subtitleString(t) = \sum_{t=0}^{\infty} \#audioString(t) \times \#subtitleString(t-\tau)$$

This is **Convolution**. So we can rephrase this problem as, find the index τ which maximizes the value of the convolution of the sequences.
"FFT convolution uses the principle that multiplication in the frequency domain corresponds to convolution in the time domain. The input signal is transformed into the frequency domain using the DFT, multiplied by the frequency response of the filter, and then transformed back into the time domain using the Inverse DFT, where DFT is Discrete Fourier Transform."
By using the FFT algorithm to calculate the DFT, convolution via the frequency domain can be faster than directly convolving the time domain signals.
It's relevant here because it "turns convolution into multiplication":

Fourier{#audioString(τ)} $\times$ Fourier{(#subtitleString)(τ)} =
Fourier{ $\#audioString(t) * \#subtitleString(t)$ } .

So finally to solve the problem,

1. we get the pointwise product sequence
   S = Fourier{#audioString(τ)} $\times$ Fourier{(#subtitleString)(τ)}
2. Do the inverse Fourier transform on it,
   inverseFourier(S)
3. Maximize inverseFourier(S) over τ .

Which will be the `delta` for Tool B.
For all the mathematical operations FFmpeg/avcodec and/or fftw3 libraries will be leveraged.

Subtitle Adjustment

This would be done with the help of the subtitle editor which has been explained in this section of the proposal and would work exactly the same for both the tools.

Outcome: A synchronized subtitle document. Thus, **Tool B** will create a subtitle document (`#out_CC`) that will **co-oCCur** with the modified audiovisual content (`#edited_video`).

TIMELINE, AT A GLANCE

- Till 5th May: Pre - GSoC period
- **Phase 0**: 6 May to 27 May
  Community Bonding Period
- **Phase 1**: 27 May to 24 June
  Start Coding
- **Evaluations**: 24 June to 28 June
  First Phase Evaluations
- **Phase 2**: 28 June to 22 July
  Keep Coding
- **Evaluations**: 22 July to 26 July
  Midterm Evaluations
- **Phase 3**: 26 July to 19 August
  Finish Coding
- **Evaluations**: 19 August to 26 August
  Final Evaluations & Code Submission

TIMELINE, IN-DEPTH

Pre-GSoC Till 5th May
- I will go through all the different concepts that are needed for the project in detail (audio fingerprinting, VAD, concepts of Signal Processing). - Alongside I will work on the core CCExtractor tool and try to solve bugs and also focus on GSoC qualification tasks.

Phase 0: Community Bonding Period 6 May to 27 May
- With the help of mentor(s), I will fine tune the milestones of my deliverables and get a clear idea of implementation of all the modules. - Create a skeleton of the project to keep the development smooth ahead. - Setup up server and development environment.

- Collect samples for testing and create a repository for it.
- **Bond** with the community!

Phase 1: Start Coding
May 28 to June 8

- Start working on the testing environment.
- Create a pseudo-code for “scoring” the alignment in Tool B
- Writing the subtitle parser tool that is required for both Tool A and Tool B. Although both have different requirements, I will extend saurabhshri’s subtitle parser to be usable by both of them.

Phase 1
June 9 to June 17

- Writing the subtitle editor tool that is required for both Tool A and Tool B. Both tools have almost the same requirements. They will take as input the `delta` which is the offset that has to be propagated throughout the subtitle entries.
- The study on all different concepts involved will continue.
- I have kept the workload less up to this point as I might have end semester exams during this time (an issue mentioned here)

Phase 1
June 18 to June 24

- Things will pick up pace from this point.
- Implementation of VAD using WebRTC for the audio segmentation in Tool B.
- Complete pending work, documentation and bug fixing.
- Prepare a report for the first Evaluations.

First Evaluations (Deliverables)
June 24 to June 28

- Tools for Subtitle parsing and Subtitle editing.
- The basic architecture of the project.
- VAD implementation
- A sample test repository

Phase 2: Keep Coding

June 29 to July 5
- Start the work on ***Dactylogram,*** the audio fingerprinting library, using dejavu as a reference. - Focus on the part of *spectrograms and peak detection* - Create the final architecture of Tool A - Work on the command line interface part, make it usable from a script.

Phase 2 July 6 to July 12
- Finish Dactylogram it must be able to generate audio fingerprints for any audio at a given offset - Using the subtitle editor insert the fingerprint anchors into the original subtitle file and enhance it. - Continue working on the command line interface.

Phase 2 July 13 to July 21
- Work on the fingerprint seeking sub-module of Tool A. It receives fingerprints[] and timestamps[] and then does the Seeking & Matching. - Use the subtitle editor and the `delta` obtained from the seeking sub-module adjust the timings of subtitles. - Complete Tool A.

Midterm Evaluations (Deliverables) July 22 to July 26
- Prepare the report for midterm evaluations. - If any documentation is remaining, finish it. Also testing and bug fixing. **- Tool A**

Phase 3: Finish Coding July 27 to August 2
- Using the subtitle parser work on the subtitle segmentation and extract `#subtitleString` - VAD will be used to process audio and extract `#audioString`

Phase 3

August 3 to August 12
- This week will be completely dedicated to the algorithm for "Alignment" of the two binary strings, resulting in the value of `delta.` - Use the subtitle editor and the `delta` obtained above to adjust the timings of subtitles. - Finish Tool B.

Phase 3 August 13 to August 19
- This will be kept as a buffer week in case of any work gets delayed. - Finish any pending work, fully-fledged documentation - Testing the working of the different sub-modules and the Tools.

Final Evaluations (Deliverables) August 19 to August 26
- **co-oCCur: A high-speed synchronization tool** consisting of Tool A and Tool B - **Submit code,** final evaluations of mentor and a detailed report on my work during the summer.

NOTES REGARDING TIMELINE

The above timeline is a tentative timeline and gives a rough idea of my planned project work. At the very least progress during GSoC will be made according to this schedule. A more detailed and final timeline will be developed with my mentor(s) help during the Community Bonding period. No separate week for documentation has been kept as it will be a part of development hence would be done alongside.

If I complete any task before the proposed time, I will proceed to the next task on schedule.

Fortnightly blog updates will keep the community aware of my progress.

I am sure about the timeline that I have set for Tool A but for Tool B I would like to discuss it with my mentor since things might take more time.

REQUIREMENTS

- #### High-Speed Server
  The Project's idea page itself mentions that "..video files are very large. You will need to deal with files that are several gigabytes long". Thus I would require a high-speed server for efficient and faster testing.

- #### A large number of Samples
  I will need a large number of samples containing media files in various container formats for better testing and hence an efficient tool.

Personal Information

Who am I?

I am Suyash Bajpai, a 3rd year undergraduate in the department of Electronics and Telecommunication Engineering at Government Engineering College, Raipur (INDIA).

I stepped into the world of programming in 10th standard, **Java** was the first language I learnt. I scored a perfect 100 in my Computer Applications Board Examinations.
I have had a steep learning curve since then, adding different skills to my knapsack of learning. I have a firm hold in C, C++, Java, Python and a decent familiarity with web-dev technologies viz. HTML, CSS, JavaScript. I try to write clean, documented and properly tested code.

I am comfortable using Linux (Ubuntu) and Windows. I am fairly proficient with version control using Git.
I use PyCharm (for Python), CLion (for C/C++) and Visual Studio Code for all other developments.

I have currently enrolled myself in the specialization "Open Source Development, Linux and Git" by The Linux Foundation on Coursera, to firmly grasp the foundations and learn as much as I can for better and optimized open source development. I intend to complete it before the Community Bonding Period.

Working Schedule, Environment and Absences

I am willing to work on all weekdays (Monday to Friday) sparing Weekends for testing, bug fixing, documentation and blogs. I shall be available to work full time with **CCExtractor** throughout the summer, having no other commitments, thus I intend to work **45+** hours a week.

While working on any project, I like keeping my tasks stacked-up on **Trello**. It enhances the way I manage projects. I will create a Trello Board covering all deliverables, weekly milestones and tasks which will also help my mentor(s) keep track of my updates. I can adjust with any other “way” if my mentor(s) suggests.

My local timezone throughout the summer will be IST (Indian Standard Time) which is UTC +5:30. I am willing to adjust the working hours of the day with what suits my mentor(s) the best.

I will be working from my home where I have a decent internet connection but with a limited monthly allowance. As mentioned in the project’s idea page, I will need to deal with large files hence I would request a high-speed server to be able to carry out efficient and faster testing. (mentioned in requirements)

I will have end semester exams whose official dates are not declared yet by the university because the General Elections of India are scheduled during April-May. According to the tentative calendar, Examinations are scheduled in the second half of May. In any case, I will not be able to contribute actively during the end semester examinations. Nonetheless, I will be actively participating in conversations over slack.
Apart from this, I have **no** planned absences over the summer. In case of an emergency, I will responsibly inform my mentor(s) with enough details and use the buffer week to catch up.

Communication

I believe that behind the success of any project in Google Summer of Code, or any open-source project in general, the communication between the mentor(s) and the student is the key factor. Keeping this in mind, I will be in constant touch with my mentor(s) and keep them informed about my progress and seek their help wherever I get stuck.

I am comfortable with any form of communication that suits my mentor. Below are the various options available:

Email: 7suyashbajpai@gmail.com
Slack: @sypai
Phone (Call, SMS, Whatsapp and Duo): (+91) 7772800787
Alternate Phone (Call and SMS): (+91) 8965833724

The weekly blog updates will be found at **https://sypai.github.io**
The first post will be made regarding selection and will include a basic introduction to the project, at the beginning of the GSoC period (6th May).

Project's Future & Post-GSoC Plans

The GSoC period will be just the beginning, I will keep working and improving co-oCCur. It has such vast possibilities of use cases. As of now, the project will deal with a constant temporal offset but it is sometimes a case that the temporal offset in the subtitles is varying. In this scenario, our approach will fail, tackling it will be a high-priority post-GSoC task. In addition to the work on the improvement and extension of **co-oCCur,** I will love to contribute to other CCExtractor's projects. **Poor Man's Rekognition** is one such project that I find extremely interesting and I'll be looking forward to contributing to it.

References

1. *A Lightweight and efficient mechanism for fixing the synchronization of misaligned Subtitle Documents.*
   Rodrigo Laiola Guimaraes, Priscilla Avegliano, Lucas C. Villa Real

2. Dejavu by willdrevo: Audiofingerprinting's awesome explanation.

3. All the figures and flowcharts are made using www.draw.io

I am not a software developement expert, but I am a learner and I am willing to work hard. I might not know everything but the proposal clearly elucidates the approach to get the project done. I have researched a lot and will keep doing so. I am thrilled about the journey! It's all about learning and growing.