ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

*Website: www.ijircce.com*

**Vol. 5, Issue 1, January 2017**

# Analyzing the Need for an Automated Testing Framework for Characterizing Audio Use Cases

Shreyansh Jain, Varsha Priya JN, Utkarsh Vaidya, Sanjay Singh Chauhan

M. Tech Student, Department of Computer, VJTI, Mumbai, India

Professor, Department of Computer, VJTI, Mumbai, India

Software Engineer, SW-Tegra, NVidia, Pune, India

Manager, SW-Tegra, Nvidia, Pune, India

**ABSTRACT*:*** Music has been an inseparable part of human life since the beginning. In present time, there have been a lot of improvements in the way the audio is stored, processed and transferred. New complex formats have been added, High Resolution Audio, Multi-Channel Audio, Dolby Sound, etc. are a few improvements. This progress is due to result of more processing capabilities added to the CPU. However, with the increase in the computing power, there is improvement in all other software's too include audio processing. Audio is a real-time thread and cannot be halted or delayed. However, in presence of multiple threads and high load on CPU, the audio thread is sometimes delayed and this results in corruption in audio. The result is that the user experience is lost. The paper aims at understanding the complexity of audio pipeline for the chosen Operating System Android and proposing a solution for detecting the use cases which can cause audio corruption and help in reducing the development time for audio related features.

**KEYWORDS*:*** Audio, Playback, Record, Tracer, Android, use-cases, Latency, automation.

## I. INTRODUCTION

Audio may simple refer to music, voice, sound, speech and many more words varying from person to person. Audio may be analog or digital. Digital audio has many advantages and applications over analog audio for compression and processing. In recent years, there has been much advancement in audio. We have many audio file formats, different types of audio input and audio output devices, high sampling rates, multiple channels, etc. to give a realistic audio effect. With time, more demand for clear and realistic audio is arising.

Audio is an integral part of human life. Every action performed in life has some audio associated with it. The blowing winds, flowing water, chirping of birds, etc. please our minds. Similarly, when we listen to good music, it motivates us. Music has touched cultures all over the world since very early times in human history. Listening to music has several health benefits including lowering stress levels, raising states of consciousness, changing moods, accessing different states of mind, relaxes patients before/after surgery by decreasing anxiety, developing the brain and is useful in meditation -which has a ton of health benefits. Several studies have shown that music education at an early age stimulates the child's brain in several ways that helps to improve verbal skills, communication skills and visual skills. Research has shown that having musical training and listening to or playing music in old age can help keep the brain healthy especially as it ages. The fact is that there is no single human culture on earth that is lived without music.

Audio is processed by Operating Systems using threads. Audio is a real-time thread. So, audio thread must be scheduled on the processor at correct interval. The audio data is placed in the system buffers by the threads. If there is delay in the scheduling of the thread, it can result in a buffer underrun. Also, if the thread is scheduled before the data is read from the buffer, it will result in buffer overflow. In both the cases, there will be incorrect audio being played.

The current Operating systems support multithreading and have multiple cores. So, large numbers of processes are executing on the CPU at any given time, and audio process can be one of them. Also, multiple audio can be played at

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

*Website: www.ijircce.com*

**Vol. 5, Issue 1, January 2017**

the same time. The OS must handle these correctly for an undisturbed user experience. If there are pauses in the playback, like silence played for a few milliseconds or there is a skipping of parts of songs, then the user may not like hearing that music and it will result in negative promotion of the software Similarly, if there is some unwanted audio played, it spoils the listener's mood. So, a lot of human effort is needed to make the user's audio experience better.

In the recent times, computing power has become cheap and there has been a boom in the automation industry. So, an experimentation can be done to automate process of identifying issues in the audio threads. With automation, use cases that can an error in the audio threads can be identified with a lesser human effort and with more precision. So, the project aims at identifying different types of audio use cases, automating the identified audio use cases and thus trying to contribute to improve the user experience. Once automated, these use cases can be scheduled to execute at different events, like, change in the audio code stack, or introduction of a new Operating System, etc. The results obtained will provide an overview of which use cases have a greater probability of failing and will give an idea of reason of failure.

Automation helps as there are many audio cases and each of them may follow a different or same path in the audio pipeline and testing each path manually is difficult and time consuming and increases unnecessary human effort. Results obtained from testing gives developers a rough idea of what impact their changes have on the user experience and helps them to make further improvements as needed or rollback the unnecessary changes. It also helps them to keep track of impact per change pushed in the system quickly with reduced efforts allowing them to focus more on development.

Android is a mobile operating system developed by Google. It is based on Linux terminal. Android source code is released under open source license by Google. Most of the devices have the open source code along with other proprietary software. Android's kernel is based on Linux kernel. Android powers hundreds of millions of mobile devices in more than 190 countries around the world. It's the largest installed base of any mobile platform and growing fast—every day another million users power up their Android devices for the first time and start looking for apps, games, and other digital content. But the quality of android audio is not as good as iOS audio. So, there is a lot of scope of improvement for audio related work on Android. So, we are choosing Android as the Operating System for our work. Additionally, [1] explains the superiority of Android Operating System over its competitors like Blackberry, iOS, Symbian and claims that Android will be a leader in Mobile Platform.

## II. ANDROID AUDIO STACK

An audio signal is an electrical voltage. It has frequencies in different ranges. Humans can hear frequencies from 20 Hz to 20 kHz.For usage by computers, storage on digital chips, audio compression, speech recognition, transmission in digital mobile phones, analog signal cannot be used. So, audio must be converted to digital format. The input analog voltage or current is converted to a numerical value proportional to the input current or voltage. The conversion introduces some error as quantization of the signal is done. Also, the conversion is performed periodically, taking samples at regular intervals. The result is a sequence of digital values that have been converted from a continuous-time and continuous-amplitude analog signal to a discrete-time and discrete-amplitude digital signal.

In Android, user applications written in the Java language. These applications can interact with the android hardware using the pipeline. Java APIs are made available to the user, which call the underlying native APIs using the Application Framework. The native libraries then interact with the Linux kernel which includes the hardware drivers. Android is open source and it allows to implement one's own hardware and drivers. The Hardware Abstraction Layer allows provides a standard for allowing communication between the Android stack and developer's hardware. Android's audio Hardware Abstraction Layer is used to connect the higher-level audio specific APIs in android.media to the underlying hardware and audio drivers. In modern Operating Systems like Android, audio is handled as a separate thread on the operating system, along with other existing threads.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

*Website: www.ijircce.com*

**Vol. 5, Issue 1, January 2017**

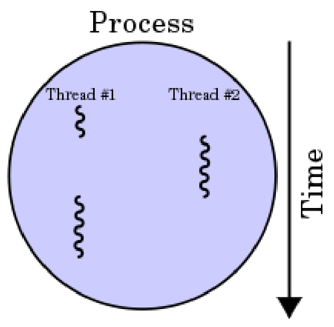

Fig.1. Thread Execution in Multi-Threading Environment

From android point of view, there is an Audio stack which explains how data flow takes place when there can be several other threads or processes running on the system. We will now understand each part in the android audio stack. Below is the Android Audio stack architecture:

- **Application Framework**
  Android comes with a software development kit which is termed commonly as Android SDK. Android SDK provides the API libraries and tools for building and developing new applications on Android operating environment using the java programming language. This procedure of developing the applications on Android platform in java programming language using the tools and API libraries provided by Android SDK is called as Android Application Framework. The application framework incudes the app code which uses android.media APIs for interaction with android audio hardware, which internally makes calls to JNI classes to access the native code for interacting with audio. A user application for playback or recording exists here.

- **JNI**
  JNI stands for Java Native Interface. The JNI code associated with android.media calls lower level native code to access audio hardware. JNI is located at frameworks/base/core/jni/ and frameworks/base/media/jni. The JNI enables the native code to be called from and to call the native applications written in assembly, C and C++ languages. JNI is useful as it helps the programmer to write native methods when then entire application cannot be written as Java. E.g. For I/O Operations, or providing sound capabilities. However, a single error in the JNI can destabilize the entire JVM. Further, for applications requiring changes in JNI, will not work on all platforms without the modification in JNI on all platforms.

- **Native Framework**
  The native framework provides a native equivalent to the android.media package, calling Binder IPC proxies to access the audio-specific services of the media server. Native framework code is located at frameworks/av/media/libmedia. AudioTrack implements the user application's side of the audio output. It runs a thread to periodically send the next audio buffer to Audio Flinger. AudioRecord implements the application side of the audio input. AudioRecord runs a thread to periodically acquire a new buffer from Audio Flinger.

- **Binder**
  Binder IPC proxies facilitate communication over process boundaries. Proxies are located at frameworks/av/media/libmedia and begin with the letter "I". Shared memory in Android's main inter-process communication system is used to transfer the audio buffers between Audio Flinger and the user application. It's the heart of Android, used everywhere internally in Android.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

*Website: www.ijircce.com*

**Vol. 5, Issue 1, January 2017**

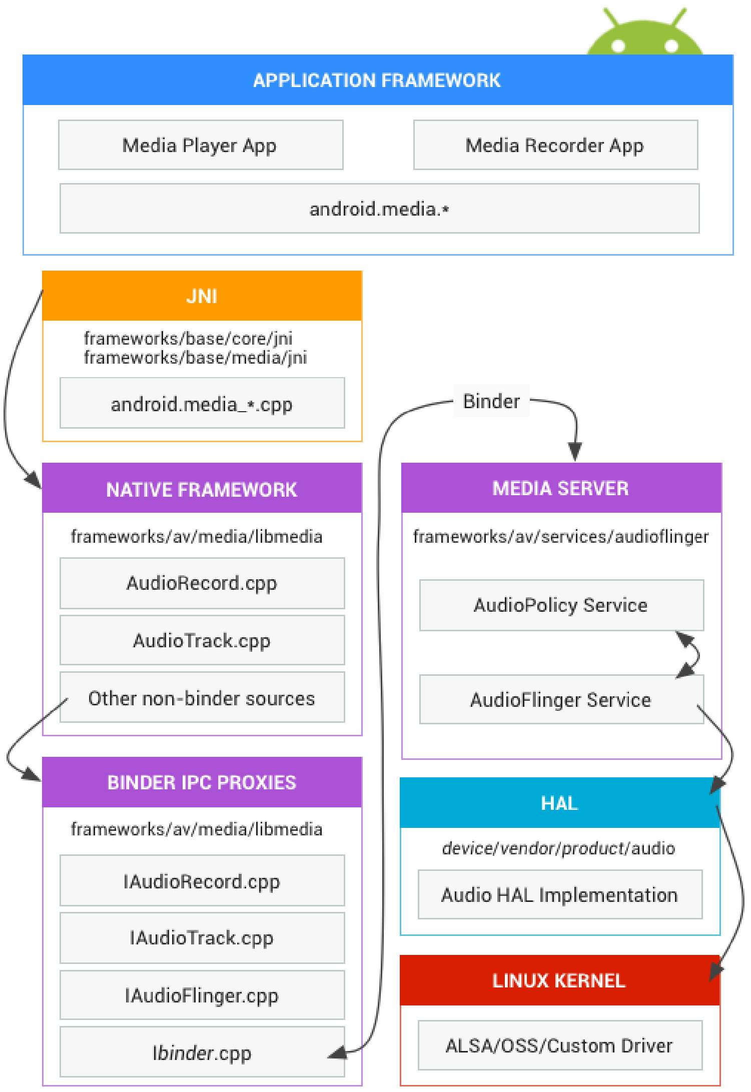

Fig.2. Android Audio Stack

- **Media server**

  The media server contains audio services, which are the actual code that interacts with your HAL implementations. The media server is located at frameworks/av/services/audioflinger. The media server is started at boot time and all its services are also started. Media Server is a separate process from the application. The media server contains codecs, file parsers, etc.

The Android media server consists of two services:

- The AudioPolicy service handles audio session and permission handling, such as enabling access to the microphone or interrupts on calls or access to speakers. It's very similar to iOS' audio session handling.
- The AudioFlinger service handles the digital audio streams. It combines streams from multiple applications into a single stream before sending to the hardware.
- Audio Flinger creates a Record Thread, which acts as a middleman between an application and the audio driver.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

*Website: www.ijircce.com*

**Vol. 5, Issue 1, January 2017**

- Audio Flinger has a "fast mixer" path, if Android is configured that way. If a user application is using native (Android NDK) code and sets up an audio buffer queue with the native hardware sample rate and period size, no resampling, additional buffering or mixing ("Mixer Thread") will happen in this step.
- The Record Thread works with a "push" method, without any strict synchronization to the audio driver. It tries to make an "educated guess" when to wake up and run, but the "push" method is way more sensitive to dropouts.
- For playback, the AudioFlinger creates a Playback Thread, which works in inverse way of the Record thread. It pushes data to the next output audio buffer of the audio driver's ring buffers using the audio HAL.

- **HAL**

HAL is Hardware Abstraction Layer. The HAL defines the standard interface that audio services call into and that you must implement for your audio hardware to function correctly. The audio HAL interfaces are located at hardware/libhardware/include/hardware. The HAL acts as a middleman between the Android media server and the Linux audio driver. HAL implementations are provided by the mobile device's manufacturer upon "porting" Android onto the device. Implementations are open, vendors are free to create any kind of HAL code. Communication with the media server happens using predefined structures. The media server loads the HAL and asks to create input or output streams with optional preferred parameters such as the sample rate, buffer size or audio effects. The typical HAL implementation is tinyALSA, which is used to communicate with the ALSA audio driver. Some vendors put closed source code here to implement audio features they feel important. The HAL may or may not perform according to the parameters and the media server must "adapt" to the HAL.

- **Kernel Driver**

The audio driver interacts with your hardware and HAL implementation. You can use Advanced Linux Sound Architecture (ALSA), Open Sound System (OSS), or a custom driver (HAL is driver-agnostic).

- **Audio Driver**

The audio driver receives the incoming audio into a ring buffer in "bus buffer size" steps using the audio chip's native sample rate, 48000 Hz in most cases. This ring buffer plays an essential part in smoothing bus transfer jitter and connects the bus transfer buffer size to the operating system audio stack's buffer size. Consuming data from the ring buffer happens in the operating system audio stack's buffer size. ALSA handles the ring buffer like this:

- Audio is consumed from the ring buffer in "period size" steps.
- The ring buffer's size is a multiple of the "period size".
- For example:
  - Period size = 480 samples.
  - Period count = 2.
  - The ring buffer's size is 480x2 = 960 samples.
  - Audio input is received into one period (480 samples), while the audio stack reads/processes the other period (480 samples).

Audio output in the audio driver works identically to the audio input and uses a ring buffer too.

- **Ring Buffer**

Ring Buffer is a data structure that uses a single fixed size buffer, which is logically connected end-to-end. It is useful for buffering streams. A circular buffer is useful for FIFO operations as it doesn't need element reshuffling when one element is consumed. Also, these are useful for overwriting data as in case of audio. The old data is overwritten with the new data. These buffers are useful for audio processing and passing audio efficiently. The frame buffer is a delay built into the audio processing loop that allows the CPU enough time to process the audio without errors and reduce the amount of CPU usage necessary to keep up. The smaller the buffer, generally the more buffers you must process ahead of time - and the more responsive/efficient processing or CPU you need - to avoid problems with processing. Buffer size is always greater than one period size. Mostly, it is twice the period size, but can be ten times too. Also, it is possible that buffer size is not a multiple of period size.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

*Website: www.ijircce.com*

**Vol. 5, Issue 1, January 2017**

- **Period Size**

It is the time between two successive interrupts by the system for filling the data in the buffer. It is an asynchronous process and hence is measured in terms of frames. Let us consider the following example:

- Sample Rate: 44100 Hz
- Audio Format: 16 bit, i.e., 2 bytes
- Number of channels: Stereo
- 1 frame = Sample in bytes * No. of channels
- 1 frame is of 4 bytes (2(bytes) x 2(stereo) = 4(frame size))
- So, total number of bytes received per second = 44100 * 4 = 176400 byte/s = 176.4 kB/s.
- If we want the system to interrupt two times in a second, then the Period Size is 176.4/2 = 88.2 kB

The higher the interrupt frequency, higher the CPU load and lower the latency. Period Size can also be seen as the number of frames in between each hardware interrupt. Period Size helps in controlling when an interrupt must be generated.

- **Analog Audio Device**

There may be several different analog components, such as a pre-amplifier for the built-in microphone, headsets, etc.

- **Analog to Digital Convertor (ADC)**

The audio chip measures the incoming audio stream in predefined intervals and converts every measurement to a number. This predefined interval is called the sampling rate, measured in Hz. 48000 Hz is the native sample rate for most audio chips on Android and iOS devices, meaning that the audio stream is sampled 48000 times in every second. Now that the audio stream has been digitized, from this point forward the audio stream is now digital audio. Digital audio almost never travels one-by-one, but rather, in chunks, called "buffers" or "periods".

- **Digital to Analog Convertor (DAC)**

The inverse of ADC, digital audio is "converted" back to analog in this point.

- **Bus Transfer**

The audio chip has several tasks. It handles ADC and DAC, switches between or mixes several inputs and outputs, applies volume, etc. It also "groups" the discrete digital audio samples into buffers and handles the transfer of these buffers to the operating system. The audio chip is connected to the CPU with a bus, such as USB, PCI, Firewire, etc. Every bus has its own transfer latency depending on its internal buffer sizes and buffer counts.

Thus, we now have a good understanding of the Android Audio stack. The entire audio path can be summarized as shown below:

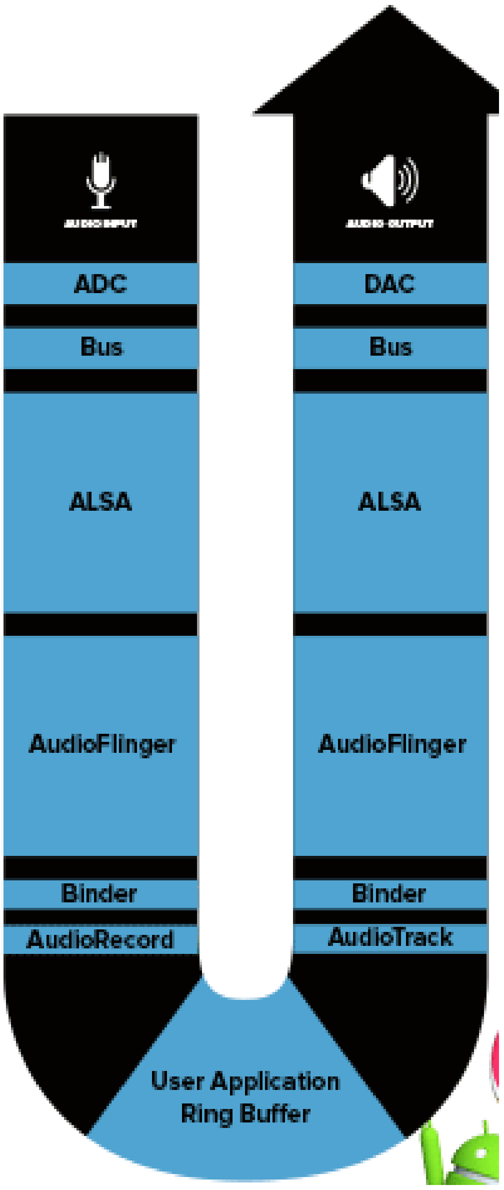

Fig.3. End to End Audio Path

## III. AUDIO USE CASES

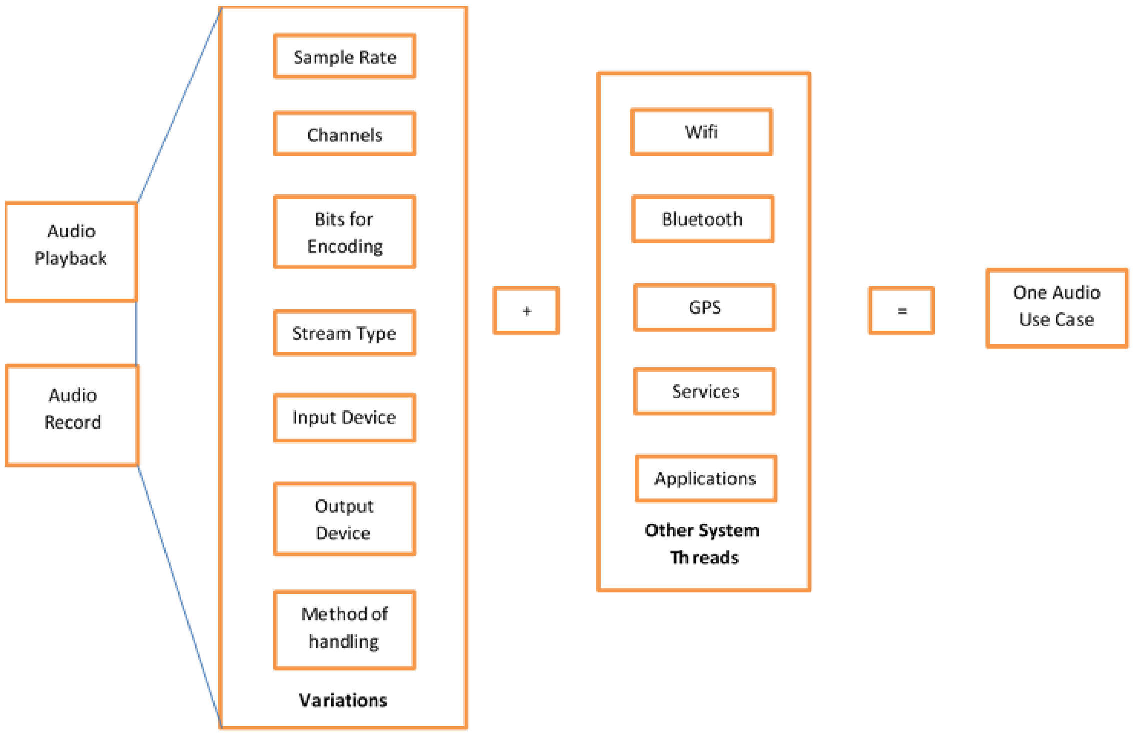

Fig.4. Audio Use Cases

An audio use case is execution of audio specific threads in the multithreaded multicore environment. There can be large number of audio use cases. A simple example is a system notification while hearing to music. A normal playback or recording of audio are also different audio use cases. Also, different applications and processes like GPS, Bluetooth, Wi-Fi, etc. are simultaneously executing on the operating system. Further audio has different properties which can be varied. So, we can have large number of audio use cases.

Digital audio has sampling rate, channels, audio formats (bits needed for encoding), stream types, output and input audio devices, and the path followed for playing the audio on the OS. These all contribute to the audio use cases.

- **Sample Rate**: Sample rate is the number of audio samples taken per second. A sample is a number representing the audio value for a single channel at a point of time. It can have values like 44.1 kHz, 48 kHz, 96 kHz, 192 kHz, etc. High Resolution audio has higher sampling rates. Samples are independent of the number of audio channels. For a one channel signal, one sample means one number. One sample for a two-channel signal means two numbers and so on.
- **Channel**: Single stream of audio information, usually corresponding to one location of recording or playback. Audio can have a single channel (mono), two channels (stereo) or can be a multi-channel audio (6, 8, etc.). Channels help in increasing the ability of a listener to perceive sound position.
- **Audio Format**: It is the number of bits of information per sample. Bit depth is meaningful with reference to a PCM digital signal. Non-PCM formats, such as lossy compression formats, do not have associated bit depths. Different values can be 8, 16, 24 or 32 bits.
- **Stream Type**: Stream type is the way audio is interpreted by the operating system. These can be as a notification, as music, as a system sound, as alarm, etc.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

*(An ISO 3297: 2007 Certified Organization)*

*Website: www.ijircce.com*

**Vol. 5, Issue 1, January 2017**

- **Input Devices**: Audio can be captured using different input devices like a microphone, mic, etc. Audio can also be played using USB cables, HDMI ports, etc.
- **Output Devices**: Audio playback can occur from number of output devices, like speaker, headsets, headphones, Bluetooth, etc.
- **Routing of Audio**: Routing means to change the audio device being used. Audio playback or capture can be routed to a different device in between as the use case is executing.
- **Audio Path**: Based on the requirement, the OS may use different path for audio use cases. It can use Fast Path in which two buffers are used for audio playback and interrupt arrives at a smaller interval. Also, deep path can be used where four buffers are used for audio threads, and interrupt occurs after a greater time interval. Further, audio data may be processed on the CPU or a separate circuit may be available for audio.
- **Audio File Formats**: Based on the file format, processing of audio data varies. MP3 data is processed in a different compared to AC3. PCM data has no header information, so playback information must be provided.

All the above parameters can be varied individually or in groups. Further these use cases can also be executed with different application executing on the OS. We rarely use the system only for playing audio. We are performing some other tasks while the audio plays in background. Also, many applications using audio for playing some sounds, or for giving instructions. Audio is an inseparable part of the applications. Also, speech is going to now become a very important part of using computer systems. This increases the use cases for audio further. In conclusion, the number of audio use cases is almost infinite.

## IV. PROBLEM STATEMENT

Android audio is far from ideal and performance is much lower when compared to other platforms. On the other hand, iOS has all the things needed for music app creation. The success of iOS devices would not have been possible if they hadn't solved the audio latency issue. Further, music apps make up only 3% of all downloads in the iOS App Store but still the Music app category is the 3rd highest revenue generating app category after Games and Social networking. On the other hand, in the Google Play store, the Music category is not even a top five revenue producing app category. The overwhelming majority of Android devices suffer from too high audio latency, preventing developers from building apps that would satisfy consumer demand on Android. As such, Google and Android app developers are leaving billions of dollars on the table for Apple and iOS developers because of the audio latency issue.

We have now understood the need for improving the audio quality on Android OS. Due to lack of proper support for audio, a lot of revenue is not generated for the Android developers. Android is open source and so attracts developers further. Also, compared to iOS, it provides better stability and multi-tasking capabilities. Further, the ease of usage for Android devices is much better compared to iOS. But still due to audio latency issues, Android is not able to generate as much as revenue as iOS. Also, [3] explains the reasons for poor response times for Android. The resource allocation is improper which causes under-utilization of the resources. Poor responsiveness of Android software can be very harmful to user perception and marketplace success.

Also, we have understood the stack followed on audio for implementing audio use cases. It is a complex architecture and has many requirements to be fulfilled. Further, Android is a multiprocessing multithreaded OS. There is no limit by software on the number of tasks running at the same time. There can be downloading in background which involves Wi-Fi or Mobile data usage. Some Bluetooth transfer may be taking place simultaneously. Increasing the complexity, a video is being played online. This involves audio, video, Wi-Fi, etc. threads running simultaneously. There must be synchronization in the audio and video; and downloading of video and playing it must all happen simultaneously. It is possible that after a certain percent of video is played, audio glitches will be heard or the synchronization between audio and video is lost or there is loss of audio. So, an issue can occur at any place in the pipeline.

The developer needs to analyze the entire system state to identify the cause of issue. The developer may need to execute the use case multiple times to reproduce the issue again as it is a real-time issue. It may need matching the configuration of the developer machine and the machine on which bug is reported. The developer then enables logs and tries to identify what is the exact cause of the bug. System traces will be captured to analyze the CPU load and scheduling of threads. Audio being a real-time thread needs the CPU as soon as it is requested. If there is any delay in scheduling, it can cause bugs in audio threads. Thus, we realize that analyzing the audio threads in a multithreading environment is a very difficult task. It involves a lot of human effort which can be up to man weeks to identify reason for a single glitch or the root cause of latency. The glitch and latency can be added at any level in the stack. So, there is a need to find certain way to improve the audio quality and try to limit the human effort needed.

In past few years, computers have become cheap with increase in speeds, storage and processing capabilities. So, we can harness this to reduce the human effort needed. With computers doing the work, it is more accurate and it takes lesser time to complete the task. Further, computer can do the same task with similar efficiency multiple times. So, developing an automated testing framework seems a good idea to analyze such a complex audio processing system.

Instead of taking system traces and analyzing the logs manually, a computer can be programmed to analyze the audio stack calls and identify which audio write call took more time than the threshold. The different audio use cases can be executed and the results can be verified which will help in assuring the quality of audio that the system is playing or recording.

With automation, tests can be scheduled at regular intervals needing reduced or no human intervention at all. The results can then be made available to the user and the user can now quickly verify the results and identify the root cause of problems in audio threads. Different types of use cases can be automatically verified and the user effort is reduced. Also, code changes in the stack can be analyzed using automation testing framework created, which can identify if there is a break in the audio stack and if yes, then what is causing the break.

So, we see that with automation, many problems get solved and it provides accurate results. Thus, helping in improving the audio quality. So, the aim is to automate the process of verifying audio threads execution in the Operating System in the presence of other OS threads by creating an application on the Android OS.

## V. PROPOSED SOLUTION

We will now discuss the solution proposed to the identified problem and how it will be implemented. So, we have discussed the need for automation testing framework for identifying audio thread related issues in the Android Operating System. By automated, we mean that human interference is not needed for executing the tests or analyzing the results. All the tasks will be done by the software and the result will be given to the user along with the analysis performed.

The first step in the solution is to identify all the use cases to be automated. This involves lot of effort as the correct use cases must be identified to avoid wastage of efforts and obtain maximum benefits. The CTS test cases are to be tested for every build created. Apart from it, device specific use cases will be identified and automated. There are different types of devices like Android TV, tablets, mobiles and other systems based on Android OS. So, the use cases must be identified for individual devices.

Once the use cases are identified, an audio application will be created which will call the Audio APIs and execute the use cases. This audio app will be written in C++. A binary of the application will be created and pushed on the Android device. Using native level code, helps in reducing the latency of JNI calls. Data of the use cases will be captured on the Android device while the use case is executed. This data can be recorded audio, traces of audio write calls or logs. This data is then pushed to the Linux machine.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Website: www.ijircce.com

Vol. 5, Issue 1, January 2017

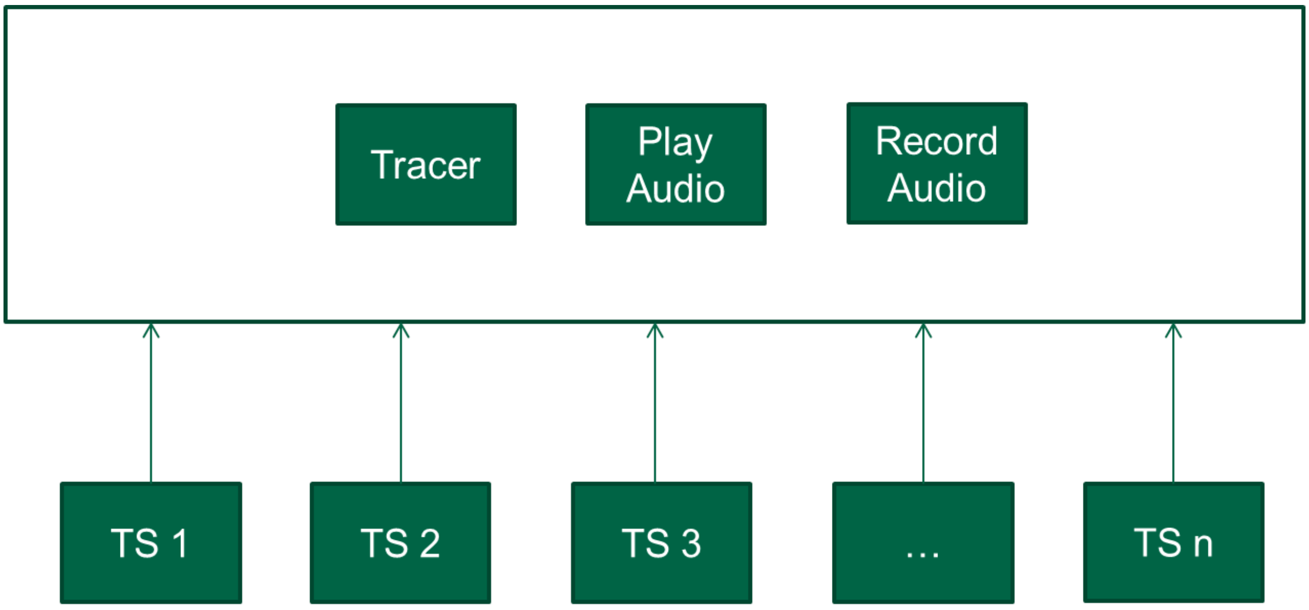

Fig.5. Proposed Automated Testing Framework Architecture

On the Linux machine, python script (marked as TS in above diagram) will be running. This script will initiate the use case execution by executing it on the Android devices, then pulling the captured data and analyzing the data of the use case. After analysis, the result generated will be compared with the standard results expected and decision will be made to pass or fail the use case execution.

## VI. CONCLUSION

We started with understanding the need for audio in a person's life. The benefits obtained from listening to music are tremendous and help in the longer run. Also, we understood how audio is become an integral part in a person's life in today's modern world. Audio is now being considered as a new dimension for interaction with machines. Everything is being transformed to use audio.

Next, we understood the Android Audio Stack and the justified the reason for selecting Android OS for the project like large customer base, open source, easy development, etc. The Audio Stack is a pipeline for processing the audio from microphone (input device) to the recording application and from the playback application to the speaker (output device). A client server model is used where the AudioTrack and AudioRecord APIs act as client and services like AudioFlinger and AudioPolicyManager act as the server. Also, we understood how HAL is to be implemented for different hardware devices and how compatibility is assured by Google among the vendors using Android OS on their devices. On understanding the Audio Stack, we next understood how audio is processed at each level in the Audio Stack. Audio is a managed as a thread in the Android OS. Then we understood the different use cases related to audio and how audio bugs are solved. Then, a solution is proposed using the automated testing framework to reduce the human efforts needed and improve the audio quality of the device.

However, it must be understood that we are creating an automated testing environment. This will only report possible bugs. The need for such a solution is that audio use cases are numerous and testing each change for each use case manually wastes a lot of human effort and precision cannot be guaranteed. So, instead of testing for each use case for every change, only selective use cases which are failing continuously can be debugged manually. The results obtained from the automated tests, also give a fair enough idea of what is causing the failure in the use case. So, analysis becomes further simple.

Thus, with the automated testing framework, we aim to reduce the testing effort and allow the developers to focus majorly on development related tasks instead of wasting unnecessary time on testing. Also, for testing, it allows them to focus only on specific use cases instead of all of them. So, more attention can be given in solving the actual bugs instead of trying to identify what is causing the bug.

Thus, the proposed solution will benefit all the developers saving their time and efforts. Also, it will help the customers as their audio experience on the device will improve. Further, as bugs will be identified before the actual release of the build, there will be a very less chance of those bugs being reported again by the customer. Thus, the company can earn more profits with customer satisfaction.

## VII. ACKNOWLEDGEMENTS

The work is supported by NVidia, Pune center for developing an automated testing framework for the SW-Tegra audio team. The use cases are developed and being tested on the NVidia devices.

## REFERENCES

1. Shyam Bhati, Sandeep Sharma, Karan Singh, “Review on Google Android a Mobile Platform”, IOSR Journal of Computer Engineering (IOSR-JCE) e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 10, Issue 5 (Mar. - Apr. 2013), PP 21-25
2. Rudi Villing, Victor Lazzarini, DawidCzesak, Sean O'Leary Joseph Timoney, “Approaches for Constant Audio Latency on Android”, Proc. of the 18th Int. Conference on Digital Audio Effects (DAFx-15), Trondheim, Norway, Nov 30 – Dec 3, 2015
3. Shengqian Yang,Dacong Yan,AtanasRountev, “Testing for Poor Responsiveness in AndroidApplications”
4. Igor Zinken, “Understanding Android audio towards achieving low latency response”, https://github.com/igorski/MWEngine/wiki/Understanding-Android-audio-towards-achieving-low-latency-response
5. Musical Android, “Understanding Output Latency on Android”, http://www.musicalandroid.com/blog/audio-output-latency-on-android-by-planet-hcom
6. https://source.android.com/devices/audio/index.html
7. Android's 10 ms problem, http://superpowered.com/androidaudiopathlatency
8. Frame Periods, http://www.alsa-project.org/main/index.php/FramesPeriods
9. Michael Tyson, “A simple, fast circular buffer implementation for audio processing”, http://atastypixel.com/blog/a-simple-fast-circular-buffer-implementation-for-audio-processing/
10. MartimLobao, “Android Audio Latency in depth”, http://www.androidpolice.com/2015/11/13/android-audio-latency-in-depth-its-getting-better-especially-with-the-nexus-5x-and-6p/
11. Presentation,JerrinShaji George, “Android Media Framework Overview”, http://www.slideshare.net/jerrinsg/android-media-framework-overview?from_action=save
12. Document, “Android Compatibility Definition Document”, https://static.googleusercontent.com/media/source.android.com/en//compatibility/android-cdd.pdf