A Framework to Measure Reliance of Acoustic Latency on Smartphone Status

Duc V. Le, Jacob Kamminga, Hans Scholten, and Paul J.M. Havinga
Department of Computer Science, University of Twente, Enschede, The Netherlands
Email: {levietduc, j.w.kamminga, hans.scholten, p.j.m.havinga}@utwente.nl

***Abstract*—Audio latency, defined as the time duration when an audio signal travels from the microphone to an app or from an app to the speakers, significantly influences the performance of many mobile sensing applications including acoustic based localization and speech recognition. It is well known within the mobile app development community that audio latencies can be significant (up to hundreds of milliseconds) and vary from smartphone to smartphone and from time to time. Therefore, it is essential to study the causes and effects of the audio latency in smartphones. To the best of our knowledge, there exist mobile apps that can measure audio latency but not the corresponding status of smartphones such as available RAM, CPU loads, battery level, and number of files and folders. In this paper, we are the first to propose a framework that can simultaneously log both the audio latency and the status of smartphones. The proposed framework does not require time synchronization or firmware reprogramming and can run on a standalone device. Since the framework is designed to study the latency causality, the status of smartphones are deliberately and randomly varied as maximum as possible. To evaluate the framework, we present a case study with Android devices. We design and implement a latency app that simultaneously measures the latency and the status of smartphones. The preliminary results show that the latency values have large means ($50 - 150\ ms$) and variances ($4 - 40\ ms$). The effect of latency can be considerably reduced by just simply subtracting the offset. In order to achieve improved latency prediction that can cope with the variances an advanced regression model would be preferred.**

I. Introduction

Smartphones are a promising platform for pervasive computing applications due to their proliferation and their numerous on-board sensors. Smartphones can be considered reliable for every day consumer use. However, in order to turn smartphones into reliable sensing devices that can be used for promising next generation applications, they often need better specifications in terms of software and hardware latency [1]. Some example applications in which smartphones require low latencies are: i. Time synchronization between smartphones, ii. Sound event detection, ii. Localization by sound, iii. Indoor Localization, iv. Reliable sensor readings (guaranteeing sample rates), and v. Safety critical applications (medical, automotive, military). Typically, a transducer requires a small amount of time to convert a signal from one form of energy to another, usually electrical signal. An additional delay is added when the signal is converted from an analog to a digital signal, which can be processed more easily. On top of this, the delay from interrupts and other concurrent tasks in multiprocessing platforms also contribute to the latency of a measurement [2]. The latency can have a significant effect on the performance of time-critical applications such as acoustic-based localization [3] and speed recognition [4].

Fig. 1 illustrates the error in a distance measurement based on acoustic sensors (microphones and speakers). Many mobile developers have measured a significant delay between the moment a sound signal reaches the microphone and the moment the respective digital value is available at the input buffer. Latency of sound signals can vary from dozens of milliseconds (iOS) to hundreds of milliseconds (Android). The speed of sound is 340.29 m/s, thus an error of 10 ms due to input latency results in an error of 3.4 m in distance ranging. In the worst case, the estimated distance from the sound source to the destination can have an error of dozens of meters. Such a large error is too much for a localization algorithm to deal with. Similarly, a large output latency is also unacceptable when emitting sounds from the output buffer to the speaker as illustrated in Fig. 1.

If a mobile operating system allows third parties to tap into the lowest hardware level, the audio latency can be minimized [5]–[11]. However, a mobile operating system often needs to be universal. For this reason, high level API's are used to support hundreds of thousands of different applications, independent of the hardware platform. As a third-party developer, it is not possible to satisfy real-time requirements when implementing arbitrary hardware that is accessed through high level API's.

In this paper, we propose a framework as a useful aid to develop audio-based pervasive systems with high Quality of Service (QoS), such as accurate localization systems. In theory, variation of the latency in a system (jitter) is mainly caused by dynamic parameters of smartphones such as available RAM, CPU loads, battery level, number of files and folders, and the number of threads [12]. If there is a correlation between a phone's dynamic parameters and audio latency it is possible to predict the current latency. Accurate prediction requires a lot of data from a large diversity of devices that can be collected with the framework presented in this paper. In order to find a correlation we design our framework such that it can simultaneously log the audio latency and the dynamic parameters of smartphones. In addition, the status of smartphones are deliberately and randomly varied as much as possible to maximize the effects on latency, which is useful to study the latency causality. With these causal measurements, audio latency can be estimated using mathematical models, ranging

978-1-5386-3227-7/18/$31.00 ©2018 IEEE

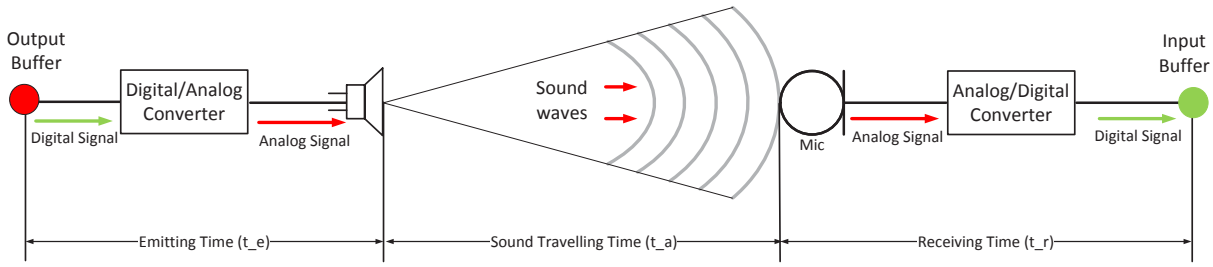

Fig. 1. Acoustic latency when measuring the travel time of sounds. The converting time is usually unknown and unstable in generic sensors devices and results in unacceptable error in the measurements.

from simple distribution estimations to advanced regression models. When the extent of audio latency can be predicted by the actual status of the phone, the effect of latency can be significantly reduced so that of the shelf smartphones can be used for applications with strict latency requirements.

In order to evaluate the framework we present a case study of Android devices. We design and implement an audio app that automatically plays and records special sounds to measure audio latency. The app simultaneously logs the status of smartphones while measuring the audio latency. The preliminary results show that the latency values have large means ($50 - 150$ ms) and variances ($4 - 40$ ms). In other words, a researcher or developer can use this app to easily measure the audio latency offset. The consequence of latency can be considerably reduced by just simply subtracting the offset. In order to achieve improved latency prediction that can cope with the variances an advanced regression model would be preferred.

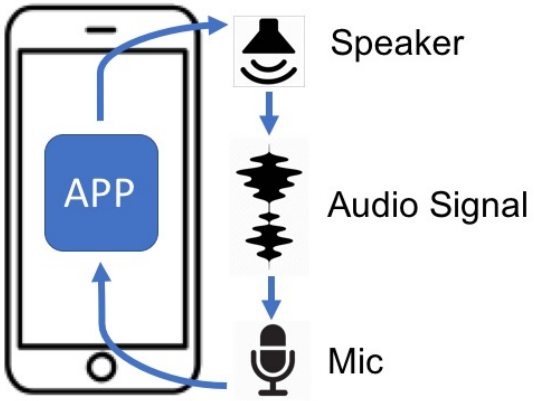

Fig. 2. Measuring the emitting time (t_e) and receiving time (t_r) with round-trip latency (T). Since the distance from speakers to microphones of the same smartphone is small, the travelling time (t_a) can be ignored. Thus $T = t_e + t_a + t_r \approx 2t_e \approx 2t_r$

II. Framework

The output and input latency are principally very similar. Therefore, it is better to measure the audio latency based on the round-trip latency. The round-trip latency is defined as the amount of time it takes from the moment an user touches the start button of the app for audio data to be processed and emitted through the speaker of a mobile device, then enter through the microphone of the same device, and be available at the input buffer to be processed. The entire round trip is illustrated by Fig. 2. By emitting and recording the sound from the same device, the round-trip latency can be consistently measured for each device.

For a clear and consistent presentation, we introduce the framework with regards to Android smartphones. We select the Android platform since Android phones are very popular (85% of market share in the first quarter of 2017 [13]). Note that the considerable latency problem in Android has been existent for years. The latency in Android has recently been reduced but remains significant for many time-critical applications such as localization. Probably latency will be reduced in the future devices; however, the framework we propose in this paper can be used with generic platforms and devices. Therefore, this framework will still be applicable for other pervasive sensing devices even if the latency problem in Android phones is reduced. The framework comprises of three components: time stamping, sampling frequency correction, and latency measurement.

A. Time Stamping

Synchronizing the played soundtrack with the microphone recording is critical. When an output signal is not synchronized with its respective input signal, the latency estimation will be inaccurate. In most arbitrary hardware platforms, e.g. smartphones, it is physically not possible to merge an output channel with an input channel without external equipment. In order to synchronize two signals, time stamps will have to be recorded at the right moment. Inside an operating systems many events can occur between sending the audio data to the output buffer and physically hearing that data through the speaker. Therefore, the actual moments in time of playing and recording the sound need to be recorded. A time stamp from a high resolution counter needs to be recorded at the time the audio track is played, temporally as near as possible to the moment the byte is converted into an analog signal in the Digital to Analog Converter (DAC). Another time stamp from the same counter is required at the moment in time of receiving

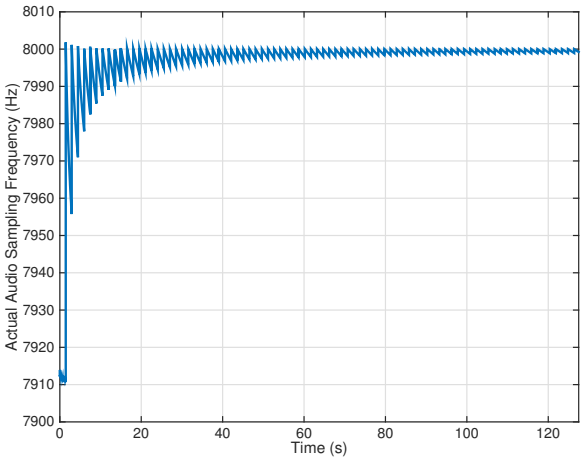

Fig. 3. The actual sampling frequency of audio measured with a Samsung Galaxy Note II smartphone.

the first byte of the recorded track from the Analog to Digital Converter (ADC). With these timestamps the played audio file can be synchronized in time with the actual recorded file.

An additional time stamp is recorded when the recording is terminated, this time stamp is temporally as near to the latest byte received as possible. This time stamp is used to measure the total duration of the recording. This duration is compared with the actual sampling frequency of the recorder. We measured that the actual sampling rate is not always the chosen sampling rate. Section II-B discusses this variation in sampling frequency.

B. Sampling Frequency Correction

Recording accurate time stamps as described in Section II-A allows us to measure the sample rate during the recording of a signal. We observed that the true sampling rate of raw audio input fluctuates around our desired rate of $8000\ Hz$. The true mean sample rate of a recorded signal interval can be obtained by

$$F_s[i,j] = \frac{\delta_{[i,j]}}{d_t}, \tag{1}$$

where $\delta_{[i,j]}$ denotes the total number of samples and d_t the measured duration of the signal in interval $[i,j]$.

The fluctuation of the measured sample rate seems to be largest in the beginning of the recording, see Fig. 3. The fluctuation can be caused by the hardware architecture or OS overhead. The ADC in the platform could be calibrated in this period.

Missing samples alter the temporal location of sound events in the recorded signal. When preforming a cross correlation to detect the sound event, the induced error can not be detected. Therefore an incorrect sampling rate can cause considerable error in the latency measurement. There exists a comprehensive survey of classical signal processing techniques for sampling-rate conversion in [14]. However, those techniques require a digital lowpass filter whose cutoff frequency depends on the sampling-rate conversion factor. This requirement is less convenient when resampling the signal at arbitrary sampling rates. Therefore, we propose to solve this issue by converting the rate based on band-limited interpolation of discrete-time signals. Nyquist-Shannon sampling theorem also tells us that band-limited interpolation can perfectly reconstruct the signal back to a continuous function from its samples. In other words, signal values at arbitrary continuous times can be correctly interpolated from a set of discrete-time samples. Remark that the original signal must be band-limited to half the sampling rate to avoid aliasing distortion.

Let $x(t)$ denote an arbitrary continuous-time signal, of which the continuous Fourier transform is denoted as

$$X(j\omega) \triangleq \int_{-\infty}^{+\infty} x(t)e^{-j\omega t}dt. \tag{2}$$

We assume $x(t)$ is band-limited to $\pm F_s/2$, where $F_s = 1/T_s$ is the sampling rate. Then $x(t)$ can be perfectly interpolated from its samples $x_s(nT_s)$, $n = \ldots, -1, 0, 1, \ldots$, if spectral energy $X(j\omega) = 0\ \forall|\omega| \geq \pi/T_s$ at uniform intervals of T_s seconds.

In fact, band-limited interpolation is the ideal interpolation for digital audio and the ideal band-limited interpolation is sinc interpolation computed through a convolution operation:

$$\widetilde{x}(t) = \sum_{n=-\infty}^{+\infty} x_s(n)h_s(t - nT_s), \tag{3}$$

where

$$h_s(t) \triangleq sinc(t/T) \triangleq \frac{\sin \pi \frac{t}{T_s}}{\pi \frac{t}{T_s}}.$$

Suppose that the sampling rate F_s is not as we desired. To compute the expected sampling rate denoted as F_s', all we need is to use Eq.3 at integer multiples of T_s'

$$\widetilde{x}(kT_s') = \sum_{n=-\infty}^{+\infty} x_s(n)h_s(t - nT_s), k = \ldots, -1, 0, 1, \ldots. \tag{4}$$

C. Latency Measurement

The latency can be determined by measuring the delay between two signals when they are synchronized in time. Fig. 4 depicts both the played and the resulting recorded audio signal. The two signals in Fig. 4 are synchronized in time by means of the timestamps described in section II-A.

The latency between two signals can be determined with either threshold detection or the cross-correlation between each pair of signals at all possible lags. Finding a correlation between two high frequency sound signals is difficult due to the various frequency components in the signal that have been introduced by noise and distortion. The correlation between two signals is significantly higher when the envelopes of the signals are utilized. Therefore, the latency between two

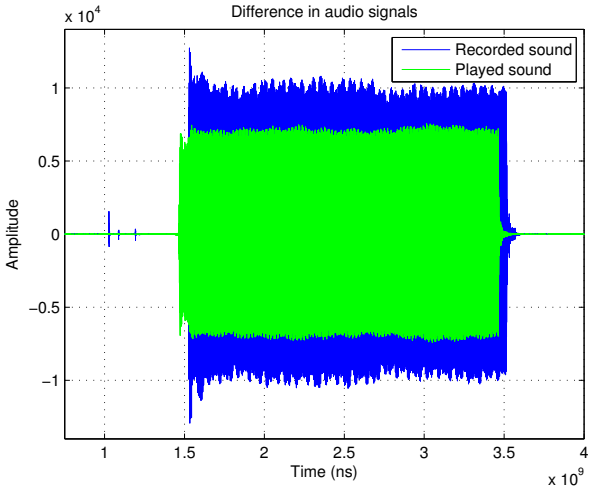

Fig. 4. Played and recorded audio signals, synchronized in time

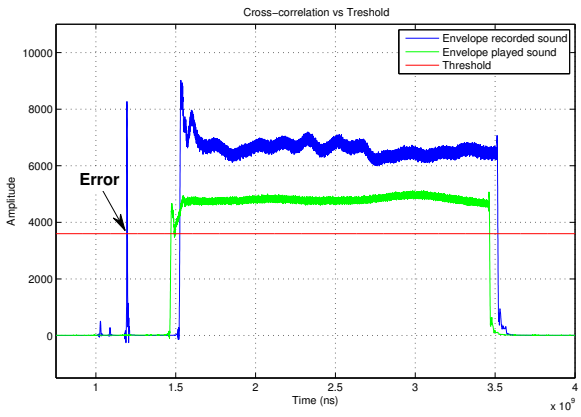

Fig. 5. Threshold versus Correlation

signals $s_1(t)$ and $s_2(t)$ is determined with their respective envelope. The envelope is an analytic signal which contains no negative-frequency components. In continuous time every analytic signal $z(t)$ can be represented as

$$z(t) = \frac{1}{2\pi} \int_0^\infty Z(\omega) e^{j\omega t} d\omega, \tag{5}$$

where $Z(\omega)$ is the complex coefficient (setting the amplitude and phase) of the positive-frequency sinusoid $exp(j\omega t)$ at frequency ω [15]. In order to obtain an analytic signal the Hilbert transform is applied. Let $x(t) = A(t)\cos(\omega t)$, where $A(t)$ is a slowly varying amplitude envelope. The Hilbert transform is very close to $y(t) \approx A(t)\sin(\omega t)$ (if $A(t)$ is constant, this would be exact), and the analytic signal is $z(t) \approx A(t)e^{j\omega t}$. Please note that obtaining the envelope is nothing more than the absolute value. I.e., $A(t) = |z(t)|$ [15]. The Hilbert transform can be obtained by a Fast Fourier Transformation (FFT) of the input signal, zeroing out the negative frequencies and performing an inverse FFT [16]. The real part of the transformed signal is the original signal. The imaginary part is the transformed signal. The absolute value of the real and imaginary part is the envelope. The envelope experiences lag relative to the original signal. This lag is the same for both signals, thus the lag does not influence the latency measurement.

Fig. 5 depicts a situation where there is noise in the recorded signal. A simple search for intersections of the sound signal with a threshold is sensitive to noise that exceeds the threshold level. Therefore, we utilize cross correlation, a more robust method for latency measurement. The normalized cross-correlation between the signals is calculated for each lag of the played audio signal.

III. Framework Implementation: Android Case Study

The software architecture of the framework implementation is shown in Fig. 6. The architecture comprises seven software components: Service Control, Audio Recorder, Audio Player, Timestamp, Dumpsys Execute, Latency Calculator, Storage, and Client User Interface (UI).

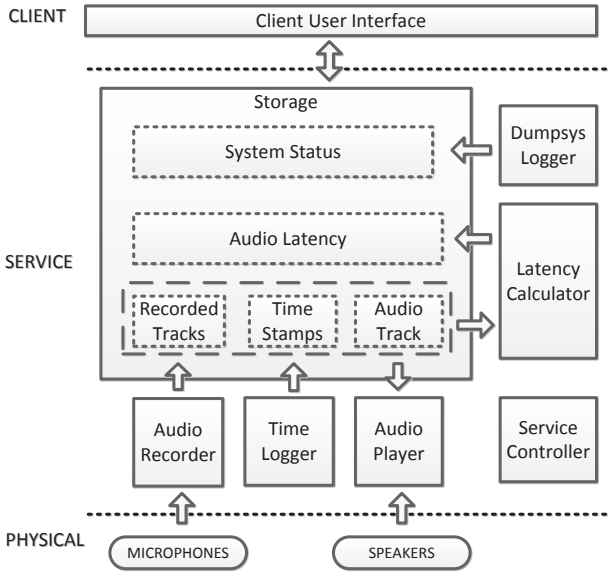

Fig. 6. Architecture of the Audio Latency app.

A. Service Control

This component is the major component of the service, which manages all components in order to obtain a dataset that contains observations of audio latency and corresponding device's status. Moreover, the Service Control connects the service with the Client UI so that users can interact with the service such as starting and stopping services, changing setting parameters, and explore the sensing data.

B. Audio Recorder

The audio signal is converted into a digital format by the Audio Recorder through the onboard microphones. In

particular, this component records audio by calling the AudioRecord and MediaRecorder services, which are available in the Android platform. The audio data is stored in the local storage of the smartphones as raw files in the PCM format. The recording process runs on a separated thread to avoid overflow when reading audio data from the input buffer.

C. Audio Player

This components is responsible for playing a selected audio track repeatedly through the onboard speakers. The audio track is open by users via the Client GUI component. The sounds emitted from the speakers will be recorded simultaneously by the Audio Recorder component via the microphones. To play the audio track smoothly with delay that is as small as possible, we use the low layer API AudioTrack as Google recently recommended. In addition, the playing process also runs on a separated thread.

D. Timestamp

This component is designed to accurately label timestamps at the moment of interests. In particular, we desire to have timestamps as close as possible to the moment when the audio track actually emits from the speakers and the moment when the audio signal actually arrives at the microphones. To do this, we overwrite the onPeriodicNotification and the onMarkerReached functions to acquire the timestamp values when the first sample sent out to the output buffer and when the first sample arrived in the input buffer.

E. Dumpsys Execute

The Dumpsys Execute component logs the status of smartphones at a low level of information. This module comprises of several classes to run the Dumpsys commands in code. We observed that most of the commands take quite a long time to dump the status into the local storage, up to seconds. Because of such considerable delay, we implemented the Dumpsys Execute component using an Async Task, which enables the classes to perform operations in background. We remark that the Dumpsys commands require rooting of the smartphones.

F. Latency Calculator

This component computes the sensing latency given the audio track, recorded tracks and timestamps. It first aligns the audio track and recorded tracks using the corresponding timestamp values. Then an envelop filter is applied on such tracks to filter out noise as well as improve the accuracy of cross-correlation. Finally, the sensing latency is measured by performing cross-correlation between the audio track and recorded tracks. The output is stored locally in the storage.

G. Storage

This component stores the audio track to be played, the recorded tracks, the timestamps, the status of smartphones, and the computed latency. As the latency is computed, the corresponding recorded tracks are deleted. Therefore, no privacy of users is disclosed.

TABLE I
LIST OF SYSTEM PARAMETERS THAT WERE RECORDED DURING THE EXECUTION OF THE AUDIO LATENCY APP

Total RAM	Free RAM	Used RAM
Lost RAM	CPU Load 1	CPU Load 2
CPU Load 3	CPU Time From	CPU Time To
CPU Horizon Length	CPU Total	CPU User
CPU Kernel	Battery Status	Battery Health
Battery Level	Battery Scale	Battery Voltage
Battery Temperature	WiFi Signal Level	Disk Latency
Disk Data Free	Disk Cache Free	Disk System Free
Dropbox Entries	audio Policy Output	

H. Client UI

The Client UI component allows the user to start and stop the sensing latency measurement. The user also can change parameters of other components, such as, audio frequency, replay times, and enabling Dumpsys. Through this component, the user also can select an audio track, record audio, as well as push the dataset to our server. In fact, users are free to determine what part of their collected data will be shared.

IV. PERFORMANCE EVALUATION

In this section we describe our test-bed setup and evaluate the results. During the experiments the Audio Latency app is deployed on four different brands of phones: Samsung, Motorola, LG, and Asus.

A. Experiment Setup

The audio track that was played and recorded in order to measure the round-trip latency was a 3-second Dual-Tone Multi-Frequency signal (DTMF tone). We played the audio track hundreds of times with our Audio Latency app at multiple days to assure a variety of the smartphone's status. In addition, we used the UI/Application Exerciser Monkey [17] to randomly interact with a set of apps such as Google Earth, Google Maps, Chrome, Polaris Office and Real Calculator. The Monkey generated pseudo-random streams of user events such as clicks, touches, or gestures on predefined apps. The purpose of using the Monkey is to stress-change the status of the smartphones randomly during the measurements of latency values. We selected a list of system parameters that are likely to be closely related to the audio sensing latency; these parameters are summarized in Table I. Based on the aforementioned setup we conducted the audio measurements with a set of six different smartphones. Although these smartphones are quite old-fashioned they still serve the purpose. Measurements with more recent smartphones can be done by the research community with our app, which will be openly published.

B. Experiment Results

The results of our experiment are presented in Fig. 7. Looking at the boxplots of measured audio latency, graphically describing the statistical population without making any assumption of the underlying statistical distribution, we find that the latency characteristics are similar for the same kinds

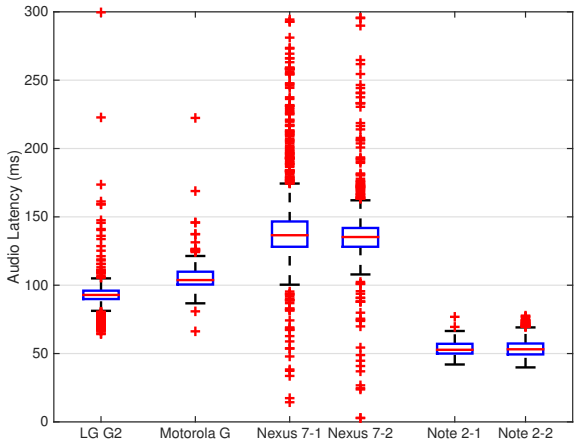

Fig. 7. The boxplot of measured audio latency based on various smartphones.

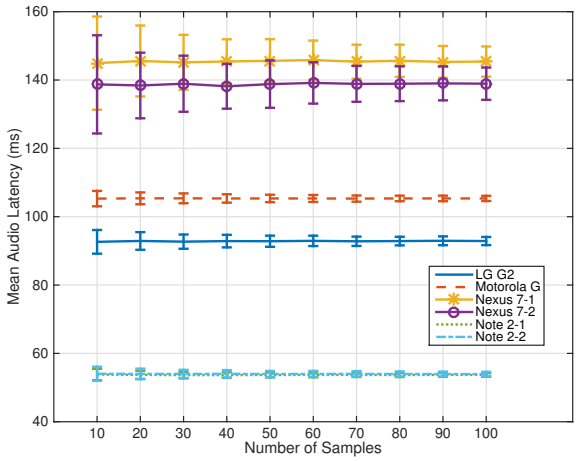

Fig. 8. The means of the sampling distribution of the audio latency means when varying the number of samples.

of smartphones. For example, the latency means of Note 2-1 and Note 2-2 are 53.69 ms and 53.97 ms respectively. In addition, the standard deviations are also similar for the same kind of smartphones, such as 5.23 ms and 6.73 ms for Note 2-1 and Note 2-2 respectively. Moreover, we observed that more expensive devices have a smaller audio latency. Note 2 smartphones have a latency of around 54 ms while Nexus 7 smartphones have latency values around 140 ms. Furthermore, there are much more outliers (the "+" markers) generated by low-cost smartphones. The cheaper smartphones also have larger standard deviation or Interquartile range.

We also investigated the possibility of finding the true mean of latency with limited numbers of measurements. In order to do that we randomly sampled latency values from our dataset. The number of samples varies from 10 to 100. For each sample size, we repeatedly drew samples with replacement 1000 times. We used sampling with replacement since latency values are independent from each other. By doing that for all devices, we obtained the mean of the sampling distribution of the mean as shown in Fig. 8. Remark that, if the population of audio latency has a mean μ, then the mean of the sampling distribution of the mean is also μ. The results plotted in Fig. 8 show that there is a potential to estimate the mean and standard deviation of a smartphone by performing our latency measurements for a quite short times. Measurements acquired with higher quality smartphones, like the Note 2, can give good estimated values from just 30 measurements, the mean and standard deviation are approximately 54 ms and 1 ms. Even in the worst case with a lower quality Nexus 7, it requires only 60 measurements to achieve a good estimate of the latency. Even though, the required time to execute 100 measurements is approximately 20 minutes. That is not long since we only need to do the calibration occasionally, at the first run and when the device updates firmware.

V. Related Work

Many have investigated audio latency in operating systems. In early work Dannenberg et al. measured latency in various operating systems and found significant amount of events that were delayed by tens of milliseconds and an increased latency for higher CPU loads [12].

A popular method to measure audio latency uses a stereo audio recorder to simultaneously record the stimulus input and response output of a device under test. Freed et al. devised a hardware oriented measuring system which can synchronously record or analyze two inputs using standard multichannel audio recording tools [18]. The output is a multichannel audio stream which preserves the temporal relationship between the two input signals. Wright et al. [19] introduce the term "The Stereo-Digital-Recorder Paradighm". The authors mention the following advantages to this method: i. The device is tested under normal conditions since there is no additional profiling software. ii. Both stimulus and response signals are mixed on the same stereo channel. All experienced latencies introduced by the measurement system will thus not cause any difference between the two signals. iii. The latency between stimulus and response is not based on any software's system clock.

Wright et al. [19] have measured the round-trip (input + output) system latencies of MacOS, Linux and windows XP using only external microphones so that the profiling software would not interfere with the system under test. They recorded both audio latencies and gesture-to-audio latencies. Gesture latencies were measured by recording the sound of a key press with a microphone, and mix this in with the computer generated response signal. They did not account for the amount of time it takes to actually press the key and how the sound of hitting actually correlates to the key press in the keyboard.

Mauerer et al. [5] described an early overview on how to adapt android 3.1 to a more real-time environment. In their demonstration Mauerer et al. preformed a latency measurement on a Motorla Xoom tablet by triggering a GPIO pin of the tablet's HDMI port with a signal generator. The system was notified by an interrupt and after a timer duration another

GPIO pin on the HDMI port is set. Both the pins were connected to an oscilloscope. The latency of the system can be determined by looking at the stimulus frequency of the signal generator. Both the works presented by Wright and Maerer et al. will yield an accurate latency measurement. However, they both require external hardware to preform the measurement and a significant effort to obtain the results. Our framework uses an accurate latency measurement that is automatized, requires no external equipment and very little effort by the user. Moreover, our framework simultaneously records the status of the smartphone.

There has been recent interest in exploring the addition of real-time (low latency) features to Android [5]–[10]. In [20] the authors present RTDroid, a variant of Android that provides predictability to Android applications. They replace the standard Dalvik Virtual Machine with a Real Time Virtual Java Machine and, for hard real-time behaviour, replace the Linux kernel with a certified Real Time Operating System (RTOS) such as RTEMS [11]. They show that just replacing these elements is insufficient to run an Android application with real-time guarantees. After redesigning Android's core constructs and system services, they were able to provide tight latency bounds to real-time applications. In [1] the same authors extend on their previous work and measure it against JPapaBench, a real-time Java benchmark. In this work they examine the Android's sensor architecture in detail and show why it is not suitable for use in a real-time context. They then introduce a re-design of the sensor architecture and show that the re-designed sensing architecture can provide predictable performance. Their work also shows the amount of effort that is needed to transform Android into a reliable sensing platform.

The work on the Android structure shows that it is possible to design a more robust open source OS. It is up to OS developers to improve on the OS and implement changes that can lead to a more stable platform with real time guarantees. Even when improving the hardware latencies and OS structure, there will always be latency in embedded systems.

VI. Conclusion

In this paper we have presented a framework, implementation, and practical experiments that measured audio latency in smartphones under stressed-status conditions. Our work is a useful aid to investigate key parameters that affect audio latency in smartphones. By doing so, the latency can be corrected by simply subtracting the offset for each smartphone. More advanced techniques such as a regression model can be used to learn and predict the dynamic latency based on the status of smartphones. While this framework is presented with regards to Android OS and smartphones, it is straightforward to generalize it for other mobile operating systems such as iOS and other devices. Since our app and its source codes will be openly published to collect more status-latency reliance measurements from social smartphones, the community will have sufficient data to investigate the latency dependency with more advanced prediction techniques, such as deep learning.

Acknowledgment

This research is partly supported by the COPAS project (Grant No. 629.002.203) in the NWO Indo Dutch Joint Programme for ICT.

References

[1] Yin Yand, Shaun Cosgroved, Ethan Blanton, Steven Y Kod, and Lukasz Ziarekd. Real-time sensing on android. 2014.
[2] Steven Rostedt. Finding origins of latencies using ftrace.
[3] Duc V Le, Jacob W Kamminga, Hans Scholten, and Paul JM Havinga. Nondeterministic sound source localization with smartphones in crowd-sensing. In *Pervasive Computing and Communication Workshops (PerCom Workshops), 2016 IEEE International Conference on*, pages 1–7. IEEE, 2016.
[4] Yashesh Gaur, Walter S Lasecki, Florian Metze, and Jeffrey P Bigham. The effects of automatic speech recognition quality on human transcription latency. In *Proceedings of the 13th Web for All Conference*, page 23. ACM, 2016.
[5] Wolfgang Mauerer, Gernot Hillier, Jan Sawallisch, Stefan Hönick, and Simon Oberthür. Real-time android: Deterministic ease of use.
[6] Igor Kalkov, Dominik Franke, John F Schommer, and Stefan Kowalewski. A real-time extension to the android platform. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 105–114. ACM, 2012.
[7] Igor Kalkov, Alexandru Gurghian, and Stefan Kowalewski. Predictable broadcasting of parallel intents in real-time android. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, page 57. ACM, 2014.
[8] Mathias Obster, Igor Kalkov, and Stefan Kowalewski. Development and execution of plc programs on real-time capable mobile devices.
[9] Ashraf Armoush, Dominik Franke, Igor Kalkov, and Stefan Kowalewski. An approach for using mobile devices in industrial safety-critical embedded systems. *Mobile Computing, Applications, and Services*, pages 294–297, 2014.
[10] Yuan Cangzhou, Gao Chen, Dong Jibing, and Sun Wei. An optimizing scheme for wireless video transmission on android platform. In *Transportation, Mechanical, and Electrical Engineering (TMEE), 2011 International Conference on*, pages 970–973, Dec 2011.
[11] OAR Corporation. Rtems real time operating system (rtos), October 2014.
[12] Eli Brandt and Roger B Dannenberg. Low-latency music software using off-the-shelf operating systems. 1998.
[13] IDC: Smartphone OS Market Share, 2015.
[14] Norbert J Fliege. *Multirate digital signal processing*, volume 994. John Wiley New York, 1994.
[15] Julius O. Smith. *Mathematics of the Discrete Fourier Transform (DFT)*. W3K Publishing, 2007.
[16] Mathias Johansson. The hilbert transform. *Mathematics Masters Thesis. Växjö University, Suecia. Disponible en internet: http://w3. msi. vxu. se/exarb/mj_ex. pdf, consultado el*, 19, 1999.
[17] UI/Application Exerciser Monkey — Android Studio.
[18] Adrian Freed, Amar Chaudhary, and Brian Davila. Operating systems latency measurement and analysis for sound synthesis and processing applications. In *Proceedings of the 1997 International Computer Music Conference*, pages 479–81, 1997.
[19] Matthew Wright, Ryan J Cassidy, and Michael F Zbyszynski. Audio and gesture latency measurements on linux and osx. In *Proceedings of the ICMC*, pages 423–429, 2004.
[20] Yin Yan, Shaun Cosgrove, Varun Anand, Amit Kulkarni, Sree Harsha Konduri, Steven Y Ko, and Lukasz Ziarek. Real-time android with rtdroid. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 273–286. ACM, 2014.