Towards Data Transmission Through Inaudible Sound in Ginga-NCL

João Victor G. de S. Nunes, Álan L. V. Guedes, Guilherme F. Lima, Sérgio Colcher
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Rio de Janeiro
jvgirard@telemidia.puc-rio.br

ABSTRACT

In this paper, we report our efforts to add support for data transmission through inaudible sound to the Ginga-NCL Digital TV middleware. We present an algorithm for encoding a bitstream in an inaudible audio signal, and to do so reliably on consumer-grade hardware. We also discuss two attempts to implement this algorithm in NCL, the language in which Ginga-NCL applications are written. The first attempt was to transmit prerecorded inaudible audio signals in a Ginga-NCL-compatible set-top-box. And the second attempt was to use NCLua to generate at runtime the inaudible audio signal. For the second attempt we extended NCL with a novel media object type, called *SigGen*, which can be used to generate arbitrary audio signals. In the paper, we describe in detail the implementation of SigGen and the result of these attempts.

KEYWORDS

Inaudible sound; Ginga-NCL; Nested Context Language; NCL

1 INTRODUCTION

Current consumer devices, such as TVs, smartphones, smartwatches, etc., often come with built-in support for radio-frequency (RF) wireless communication. This support requires RF hardware and depends on technologies such as Bluetooth, IEEE 802 WiFi, or 4G mobile networks. There are situations, however, where the RF data network is not available, or where it is more advantageous to use some other form of wireless communication. One such alternative is communication through inaudible sound.

The same consumer devices that come with RF support usually come equipped with speakers and microphones which enables them to produce and detect inaudible sound. By inaudible sound we mean sound waves in frequencies that cannot be heard by humans—usually those above 19–22 kHz. There are many works in the literature that show that communication through inaudible sound is feasible on consumer-grade PCs, laptops, and smartphones [3, 4, 6]. Here we are mainly concerned with the applicability of this technique to digital TV scenarios, in particular, to those scenarios where the TV broadcasts data to nearby devices through inaudible sound.

In this paper, we report our ongoing efforts to enable inaudible sound applications in the Ginga-NCL Digital TV middleware [1]. These efforts, so far, consisted of two steps. First, we came up with a method for encoding a bitstream in an inaudible audio signal, and to do so in a way that the resulting signal could be generated and decoded reliably on consumer-grade hardware.

After that, in a second step, we tried to implement this encoding algorithm in NCL, which is the language in which Ginga-NCL applications are written. NCL is a declarative language with no support for audio synthesis. So, our first attempt was to record the signals generated by our algorithm into raw audio files; then we combined these files in an NCL document and tried to play the document in a Ginga-NCL-compliant set-top box. Unfortunately, due to hardware limitations, that didn't work.

In face of this, we decided to extend the NCL language with a new type of media object, called SigGen, which can generate arbitrary audio signals. We implemented the SigGen object and its underlying player in PUC-Rio's NCL player, using the GStreamer multimedia framework. Then we implemented our encoding algorithm in an NCLua script (the scripting language of NCL) which was used to drive SigGen objects. Unfortunately, that also didn't work. (In PUC-Rio's implementation, the rate at which changes occur within an NCL document is limited by the display framerate—about 60Hz—which is too low for the correct operation of our encoding algorithm.)

The rest of the paper is organized as follows. In Sections 2 and 4, we describe our main contributions: the encoding algorithm and the SigGen media object. In Sections 3 and 4, we give a detailed account of our attempts to implement the proposed algorithm in NCL and of the lessons we learned in the process. In Section 5, we compare our encoding algorithm with similar algorithms we found in the literature. Finally, in Section 6, we present our conclusions and future work.

2 THE ENCODING ALGORITHM

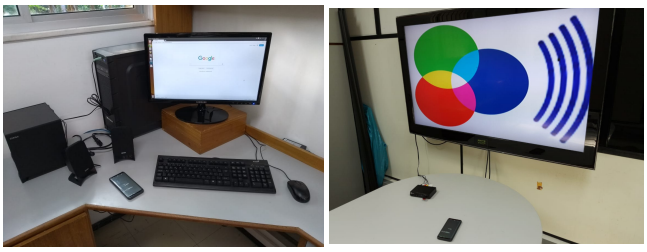
We will describe our encoding algorithm in two parts. In the first part, Section 2.1, we discuss how we selected inaudible frequencies that combined minimize the audible background noise. In the second part, Section 2.2, we discuss the encoding algorithm together with its transmission and reception parts.

The hardware we used in the development of this algorithm is illustrated in Figure 1a. It consisted of a PC with speakers and a smartphone. The PC had an Intel Core i5 (6th gen.) processor, 16 GB of RAM memory, Edifier XM2PF speakers, and run Ubuntu 16.04 LTS. The smartphone was a Samsung Galaxy S8 running Android 8.0 Oreo.

We implemented the transmission part of the algorithm in the GStreamer multimedia framework. The GStreamer pipeline we used for the tests consisted of three elements connected in series: *AudioTestSrc* to generate frequencies; *AudioConvert* to convert raw audio buffers to samples; and *AutoAudioSink* to send the raw samples

In: XV Workshop de Trabalhos de Iniciação Científica (WTIC 2018), Salvador, Brasil. Anais do XXIV Simpósio Brasileiro de Sistemas Multimídia e Web: Workshops e Pôsteres. Porto Alegre: Sociedade Brasileira de Computação, 2018.

ISBN 978-85-7669-435-9.



(a) (b)

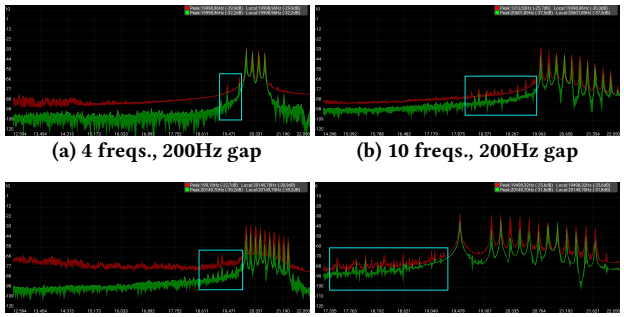
Figure 1: (a) PC environment and (b) TV environment (set-top box); both broadcasting to an Android smartphone.

to the audio driver. For reception, we used the Android AudioRecord API[1] to analyze the audio spectrum by taking samples.

2.1 Frequency Selection

To explain our selection process, we first need to introduce the concepts of *guard band* and *beat frequencies*. The guard band is the gap in the audio spectrum between two data-carrying frequencies. This gap is introduced to ensure the unambiguous reception of the data carried by contiguous frequencies. Sometimes, however, the combination of data-carrying frequencies and guard band may generate beat frequencies. These beats happen, for instance, when two frequencies interfere with each other. Such interference can generate a new frequency with a periodic variation of volume and a rate that is equal to the difference of the two initial frequencies. Beat frequencies are undesirable because they can generate noise in frequencies which can be heard by humans.

We chose frequencies based on experiments we did in a quiet room. The goal was to select the maximum number of frequencies which generated fewer *beats* and had a good *guard band*. Figure 2 depicts the spectrum visualization of four experiments using the Spectrum Analyzer[2]. In the figure, the cyan rectangle highlights beats.



(a) 4 freqs., 200Hz gap (b) 10 freqs., 200Hz gap

(c) 10 freqs., 150Hz gap (d) 12 freqs., 150 Hz gap+ref.freq

Figure 2: Spectrum analysis of four experiments with frequencies ranging from 17,335Hz to 22,050Hz.

In our definition of "inaudible", we chose the starting point of frequencies to be at least above 19000Hz, since some people can hear sounds in frequencies below this value. Fewer can hear above it, so it is viable to work in this area of the spectrum. The reception hardware defines our maximum frequency to be around 22000Hz, which seems to be the operational limit for consumer-grade hardware.

In the first experiment (Figure 2a), we used 4 frequencies with a 200Hz guard band. These four frequencies generated fewer beats but also carried fewer data (assuming one bit per frequency). In the second experiment, we used 10 frequencies (Figure 2b) with the same 200Hz guard. Although more data was transmitted, these 10 frequencies increased the width and amplitude of the beats. In the third experiment we changed the guard band to 150Hz (Figure 2c) and kept the same 10 frequencies. This retained a good reception and reduced both problems caused by the added frequencies with a larger guard band.

We found our best condition using 13 frequencies (Figure 2d). In this experiment, the first frequency starts at 19500Hz and is separated from the next frequency by a gap of 500Hz. This first frequency is intended to be easily captured as a reference frequency. The greater initial gap between the reference frequency and the others reduces the probability of interference between them. The remaining frequencies are located from 20000Hz to 21650Hz and are separated by a 150Hz guard band. The inclusion of the reference frequency added some beats with very low amplitudes in the high-frequency area, making it possible but hard to be heard by users, but it improved reception without affecting the bit-rate.

2.2 The Algorithm

In our algorithm, the reference frequency is at 19500Hz; this frequency signals the presence of the transmission but carries no data itself. The remaining frequencies carry one bit each, which are to be interpreted in the little-endian format. A message is transmitted by varying the amplitude of these data-carrying frequencies over time, i.e., by toggling them up and down. If the frequency being analyzed is below -60dB the message as a whole is considered invalid, provided that a single misread bit could interfere on the message as a whole.

Thus, above -60dB we set a new amplitude threshold. An amplitude above this new threshold corresponds to a bit 1 and, otherwise, to a bit 0. This threshold is determined dynamically accordingly to the current amplitude of the reference frequency. This is done because the distance between the speaker and the smartphone affects the amplitude of the frequencies when is received by the microphone. The fixed threshold for invalid bits (-60 dB) should not be lower, due to other ambient sounds possibly be mistakenly considered valid or higher so that the dynamic threshold has a wide enough area for the bit 0 amplitude reception.

The reception is done in three steps: sample capture, conversion and analysis (we have not implemented a synchronization method). First we capture 44100 samples per second, as it is recommended to function in all Android devices in the API's documentation and it suits our needs. By the Nyquist Theorem, this provides us, at most, coverage for 22050Hz frequency. Each sample takes around 0.5ms to be captured. Then we use a FFT (Fast Fourier Transform) algorithm with a buffer size of 512 to convert the raw sound sample to values in dB. Finally, we analyze the amplitude of each frequency comparing

[1] developer.android.com/reference/android/media/AudioRecord
[2] play.google.com/store/apps/details?id=com.raspw.SpectrumAnalyze

itself to the reference frequency amplitude. One frequency is a bit 1 if it has the same or greater amplitude than the reference frequency.

After all the samples of the bits of the message are collected during its 100 ms period, the previous amount of 1's and 0's of that particular message are analyzed. It is determined which one has at least 2/3 of the total count of samples. In the end, it is set if that bit is a 1 or 0 and then compose the message's eventual outcome.

3 THE SET-TOP BOX EXPERIMENT

Before evaluating our approach in the TV environment (Figure 1b), we experimented on the PC environment (Figure 1a). In particular, we checked if the same signal which was generated in real-time by our GStreamer pipeline could be transmitted by pre-rendered audio samples—and the result was positive. The pre-rendered message was received just as the real-time one. On average, in the PC environment we achieved the same bit-rate using pre-rendered samples. We did this experiment because although TV environment has no support for audio synthesis it can play audio files. So, one approach to inject an inaudible audio signal in the TV environment is through an application which uses such pre-rendered samples.

The Brazilian Digital TV system uses the Ginga-NCL middleware for application development. Those applications are written in the NCL language [1]. Figure 3 depicts an initial version of this application. Each frequency uses a `<media>` element for each pre-rendered sample. The volume `property` defines the bit value.

```
<media id="freq_ref" src="19500.mp3">
  <property name="volume" value="100%">
<media>
<media id="freq_1" src="20000.mp3">
  <property name="volume" value="30%">
<media>
```

Figure 3: Part of the NCL code using pre-rendered samples.

The TV environment we used (depicted on Figure 1b) consisted of a TV plus a TS 2017 set-top box. This set-top box was distributed by the Brazilian government during the switch-off of the analog TV signal. The deployment of our application in this environment lead us to discover a *decoding bottleneck*. Although the TS 2017 set-top box is compliant with the SBTVD [1] requirements, it has only one decoder for video and one decoder for audio, which means that it cannot play more than one audio file at a time. This limitation prevented us from using the pre-rendered audio samples.

One possible workaround to this bottleneck is to encode the signal into the audio transmitted by the broadcaster. The audio signal can then be mixed with the video being broadcast and received in all television sets, including the sets of those users not interested on it. But regarded that only high frequency sounds are being generated to transmit the message, those users do not need to worry about it.

4 THE SIGGEN MEDIA OBJECT

With the failure of the set-top box experiment, our next attempt was to extend Ginga-NCL with support for audio synthesis. We did this by integrating our GStreamer pipeline into PUC-Rio's NCL player [7], as a new media player, called SigGen. More precisely, we extended NCL with new type of `<media>` element, called `x-ginga-siggen`. Figure 4 presents a NCL application that creates such `<media>` elements. The `freq` and `volume` properties of these elements are used to synthesize a pure frequency.

```
<media id="freq_ref" type="application/x-ginga-siggen">
  <property name="freq" value="19500"/>
  <property name="volume" value="1"/>
</media>
<media id="freq_1" type="application/x-ginga-siggen">
  <property name="freq" value="20000"/>
  <property name="volume" value="0.3"/>
</media>
```

Figure 4: SigGen media object declarations in NCL.

We extended our initial pipeline to also present a visual representation of each frequency. This way a visual representation of the frequencies can be shown on screen. The complete GStreamer pipeline we used is depicted in Figure 5. Note that the SigGen player does not require an audio decoder: it generates raw audio data which can be passed directly to the audio driver.

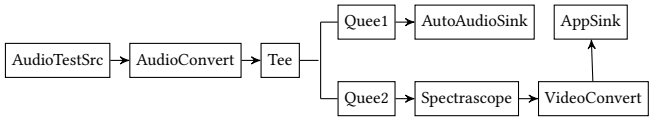


Figure 5: Player SigGen pipeline.

With the SigGen player working, we moved on to implement our encoding algorithm using an NCLua script. The idea was to use such script to change the properties of SigGen media elements (one per frequency) dynamically via the NCLua `event.timer` API. However, we run into another problem. In PUC-Rio's NCL player, the speed at which the event-loop runs is limited by the monitor frame-rate (usually, 60Hz). This upper limit is too low for the correct implementation of our encoding algorithm, which requires a precision of the order of microseconds.

One way to work around this problem is what is called, the *use case Timeline*. This use case is not intended to have a high bit-rate requirement to transmit messages. Instead it focus on transmitting time-stamps with precision greater then Ginga's event loop, so the bottleneck is surpassed by making the algorithm slower. To send time-stamps during a TV show, for instance, we may use the first four frequencies to determine the show and the remaining 8 frequencies to carry the time-stamps since the show began. For example, a pre-downloaded image or video propaganda can be presented at some specific moment of the TV show on the smartphone. Two videos were made to present a simulation of this propaganda scenario. On video[3] the smartphone screen is shown and on video[4] the transmission as a whole.

5 RELATED WORK

Other works also share our motivation in terms of data transmission using inaudible sound. Jang et al. [5] and Bang et al. [2] focus on sending only one message during all transmission, whereas

[3] imgur.com/a/FUov3mu
[4] imgur.com/a/MirE3sy

Table 1: Comparison summary.

Feature	Jang [5]	Bang [2]	Lakhwani [8]	Tsugawa [9]	This work
Number of data/total frequencies	32/40	32/40	2/2	8/20	12/13
Sampling rate (Hz)	48000	44100	44100	44100	44100
FFT sample size	32768	8192	1024	512	512
Frequencies used (kHz)	18 to 24	18 to 22	20.7 and 21	17.959 to 20.973	19.5 and 20 to 21.6
Dynamic bit threshold	yes	no	yes	yes	yes
Error handling method	CRC 8 bits	CRC 8 bits	no	Hamming Code (7,4)	Invalid bit
Send time (ms)	92	500	105	150	100+300 (data+silence)
Bit-Rate (bps)	347 (same msg)	64 (same msg)	19	53	30
Sucess rate	99.8%	97.5%	76%	93%	60%
Synchronization method	no	no	handshake	preamble detection	no
Noise when redone with GStreamer	yes	yes	no	yes	no

Lakhwani et al. [8] and Tsugawa et al. [9] focus on sending a sequence of messages. We discuss each of these works next. Table 1 presents a summary of their main features.

Jang et at. [5] use 40 evenly divided frequencies in a flexible range between 18000Hz and 24000Hz. They use 8 bits for CRC (cyclic redundancy correction) and transmit each piece of data for 92ms, achieving a bit-rate of 347bps. Their main limitation is the fact that they use 48kHz as sampling rate. This sampling rate increases the number of frequencies that can be used, but is not recommended for reception on Android smartphones.

Bang et al. [2], similarly to Jang et al., use 40 frequencies between 18050Hz and 21950Hz with a guard band of 100Hz. They also use 8 bits for CRC and transmit each piece of data for 500ms, achieving bit-rate of 64bps. Their main limitation is that they do not have a dynamic bit threshold to handle distance variation between the transmitter and the receptor.

Lakhwani et al. [8] use only two frequencies, 21000Hz and 20700Hz, the former representing a bit 1 and the latter representing a bit 0. They transmit each piece of data for 105ms achieving a bit-rate of 19bps. To handle the synchronization of messages, their transmitter also have a microphone and performs a handshake with the receiver. Lakhwani et al. are not concerned with improving the bit-rate or correcting errors.

Tsugawa et al. [9] use 20 frequencies between 17959Hz and 20973Hz. Six of these are used for Hamming Code error correction, two for dynamic bit threshold, and two for synchronization. The synchronization bits are used before the data bit analysis happens at the start of each transmission. Their algorithm waits for two successful transmissions of the synchronization bits in order to guarantee the correct start of the data bits analysis. Tsugawa et al. transmit each piece of data for 150ms achieving a bit-rate of 53bps. However, they work only with pre-rendered audio samples generated on MATLAB.

Our transmission algorithm achieves a bit-rate of 30bps. On the PC environment described earlier, it has achieved an average rate of successful reception for the whole message of 60% and for the bits individually of 76%. The algorithm with the highest transmission rate in the works we are considering is the one from Tsugawa et al. This algorithm also has a good success rate. We used our GStreamer pipeline to test the frequency selections of all algorithms discussed in this section. Except for Lakwhwani et al., which only uses two frequencies, all of the selections generated frequency beats and noise.

Comparing the number of frequencies we used (13) to the number used in other works, e.g., 20 in Tsugawa et al., our work indeed uses less frequencies, but retains no audible noise reproduction coming from undesired beat frequencies while using the GStreamer pipeline with our frequencies placement . An increase in the number of frequencies to 20 could be attempted in two ways. One way is to widen the area of the spectrum used by the reallocation of the reference frequency, but this could increase the amount of noise generated. Another way is to decrease the gap between the frequencies and increase the FFT sample size. This, however, would slow down the capture speed of the Android devices which would reduce the bit-rate.

6 FINAL REMARKS

This paper presented lessons learned from our experiments to transmit data through inaudible sound in the PC and TV environments. Although related work achieves higher bit-rates, our contribution is in the discussion of the problems and workarounds for making the technique feasible on the digital TV environment.

As future work, we intent to improve the use of our data-carrying frequencies in order to increase the bit-rate achieved by our algorithm. We also intend to extend the algorithm to handle errors and to improve the transmission-reception synchronization.

REFERENCES

[1] ABNT 15606-2. 2007. Digital Terrestrial TV — Data Coding and Transmission Specification for Digital Broadcasting — Part 2: Ginga-NCL for Fixed and Mobile Receivers: XML Application Language for Application Coding. (2007).
[2] Green Bang, Myoungbeom Chung, and Ilju Ko. 2016. Data communication method based on inaudible sound at near field. (2016), 4.
[3] Luke Deshotels. 2014. Inaudible Sound As a Covert Channel in Mobile Devices. In *Proc. 8th USENIX Conf. Offensive Technologies*.
[4] Michael Hanspach and Micahel Goetz. 2013. On Covert Acoustical Mesh Networks in Air. *J. Communications* 8, 11 (2013). DOI:http://dx.doi.org/10.12720/jcm.8.11.758-767
[5] Insu Jang, Myoungbeom Chung, and Hyunseung Choo. N/A. Reliable Short-Distance Data-Transmission Mechanism Using Inaudible High-Frequency Sound. (N/A), 10. https://insujang.github.io/assets/pdf/research_paper_data_communication.pdf
[6] Soonwon Ka, Tae Hyun Kim, Jae Yeol Ha, Sun Hong Lim, Su Cheol Shin, Jun Won Choi, Chulyoung Kwak, and Sunghyun Choi. 2016. Near-ultrasound Communication for TV's 2nd Screen Services. In *Proc. ACM 22nd Ann. Int. Conf. on Mobile Computing and Networking*.
[7] TeleMidia Lab. 2018. (2018). https://github.com/TeleMidia/ginga
[8] Sahil Lakhwani, Nishant Pardamwar, and Nikhil Khewalkar. 2015. High Frequency Sound Based Device Communication. *IJARCCE* 4, 3 (March 2015).
[9] Hiroaki Tsugawa and Masakatsu Ogawa. 2017. Proposal of Ultrasonic Communication Method and Its Application to Position Estimation System. *J. Signal Processing* 21, 4 (2017).