LSync: A Universal Event-synchronizing Solution for Live Streaming

Yifan Xu, Fan Dang*, Rongwu Xu, Xinlei Chen, Yunhao Liu
Tsinghua University
{xuyifan20, xrw18}@mails.tsinghua.edu.cn, {dangfan, yunhao}@tsinghua.edu.cn, chen.xinlei@sz.tsinghua.edu.cn

***Abstract*—The widespread of smart devices and the development of mobile networks brings the growing popularity of live streaming services worldwide. In addition to the video and audio transmission, a lot more media content is sent to the audiences as well, including player statistics for a sports stream, subtitles for living news, *etc.* However, due to the diverse transmission process between live streams and other media content, the synchronization of them has grown to be a great challenge. Unfortunately, the existing commercial solutions are not universal, which require specific server cloud services or CDN and limit the users' free choices of web infrastructures. To address the issue, we propose a lightweight universal event-synchronizing solution for live streaming, called LSync, which inserts a series of audio signals containing metadata into the original audio stream. It brings no modification to the original live broadcast process and thus fits prevalent live broadcast infrastructure. Evaluations on real system show that the proposed solution reduces the signal processing delay by at most 5.62% of an audio buffer length in mobile phones and ensures real-time signal processing. It also achieves a data rate of 156.25 bps in a specific configuration and greatly outperforms recent works.**

***Index Terms*—live streaming, synchronization, chirp signal**

I. Introduction

With the proliferation of mobile networks [1], [2], there has been a recent surge in popularity for live streaming. Streamers share the live events by recording them with a camera and a microphone, uploading them to the streaming server, and publishing them to websites for people to watch. As long as the Internet is accessible, viewers are able to enjoy the live streams using computers and mobile devices at any time and anywhere. In 2020 and 2021, a large number of people were kept at home due to the COVID-19, which contributes to the growing popularity of live streaming. Live commerce helps sustain a great many business companies, allowing them to interact with audiences and sell products online. Live education also provides a new teaching platform outside the classroom for teachers and students to communicate.

With the new applications based on live streaming, much more media content, including slides, quizzes, and subtitles, is transmitted to audiences in live streaming services. Typical examples include updating player statistics for a sports stream, displaying product details for a live shopping stream, sharing slides with students for a live education stream, and sending questions for a live quiz stream. The synchronization among the video stream and all other content is of significant value. Take Rain Classroom [3], a widely used online education platform, as an example, a teacher can send slides as well as quizzes to students while hosting the live streaming. If the page-turning of slides is inconsistent with the video, students may fail to follow the lesson. While in HQ Trivia [4], a live trivia video game, the out-of-sync time between the live and the quiz would significantly impact the user experience.

Unfortunately, unsynchronization between live streams and other media content is very likely to occur. This is because live stream and other content are transmitted through different protocols and network channels (*i.e.,* the streaming channel and the information channel) with different transmission delays. Moreover, the encoding, transcoding, and distributing process also introduces extra delays [5].

There are various solutions to solve the synchronizing issue. The simplest solution is to add a fixed time delay to the information channel. However, the delay is hard to estimate, and the network fluctuation would weaken its effectiveness. A solution introduced by Alibaba Cloud is to insert several SEI frames, which contain various types of data during the encoding process, somehow representing time-related information when performs H.264 encoding for video stream [6] and the information is recovered for synchronization as soon as the video gets decoded. Amazon IVS allows users to use ID3 tags [7] to embed metadata [8]. However, this technique adds metadata when encoding, thus requires additional procedures running on the server-side. It also requires the proprietary Amazon IVS player to extract the embedded information. These commercial solutions suffer **a significant limitation**: they rely on their specific infrastructures and proprietary software. Users (*e.g.,* start-ups) cannot choose the cloud service providers freely. They may have to pay extra to use a specific commercial solution instead of an open-source or a standard public-cloud solution.

This paper aims to develop a universal method, named LSync, to achieve event-synchronizing for live streaming. LSync is designed to be compatible with general streaming software and infrastructure, including various CDN platforms and mainstream streaming techniques. The **key idea** of LSync is to insert modulated audio signals with metadata into the original audio stream, which is demodulated on the audience to help synchronization. To have a good synchronization performance while keeping the quality of user experience, the proposed synchronization scheme should meet the following requirements: 1) The inserted modulated audio signals do not interweave with the original audio; 2) The inserted modulated

* Corresponding Author

978-1-6654-5822-1/22/$31.00 ©2022 IEEE

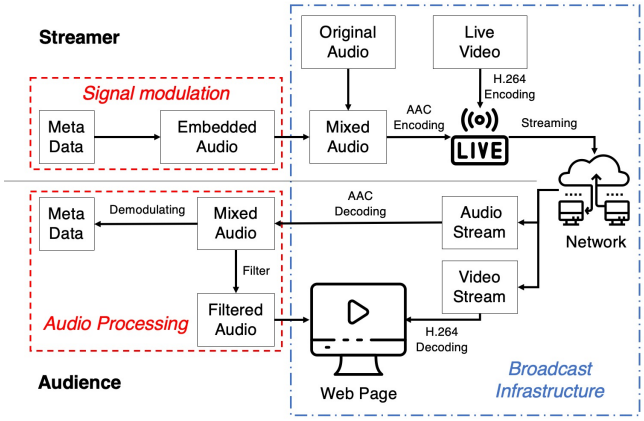


Fig. 1: System architecture

audio signals are not cut off by Advanced Audio Coding (AAC) encoder; 3) The inserted modulated audio signals are able to resist the interference, *i.e.,* be demodulated under a low signal-to-noise ratio (SNR) on the audience side; 4) The demodulation and synchronization process on the audience side should be real-time.

To better understand the concept of LSync, we provide a brief overview of how it works at a high level in Fig. 1. LSync periodically inserts modulated audio signals containing metadata, which helps the synchronization, into the original audio stream at the streamer side. After the modulation, the stream goes through encoding, transcoding, network transmitting, and finally decoding at the client-side, *i.e.,* the web browser. Before the stream is played, the mixed audio signal is identified, and then the embedded metadata is extracted. Finally, the inserted signal is filtered out from the audio stream to ensure that no disturbance is made to the audience.

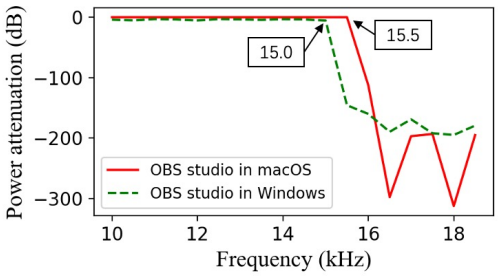
To meet the requirements mentioned above, we design our synchronization scheme as follows. To avoid the inserted modulated data being cut off by AAC encoder and prevent it from interweaving with the original audio, carrier frequencies between $14\,\mathrm{kHz}$ and $15\,\mathrm{kHz}$ is selected. To resist the interference for easy demodulation under a low SNR, we design a chirp spread spectrum (CSS) based method for modulation. Using CSS [9], we can keep the power of embedded signals at a low level so that not only can the data be successfully recovered during signal analysis, but the disturbing signal can be easily suppressed as well. To realize real-time processing on the audience side, we design an efficient algorithm, which performs the fast Fourier transform (FFT) just for once with a little more calculation in each chirp-length time window to locate and synchronize data packets and decode data, ensuring real-time signal processing.

The contributions of this paper are three folds.

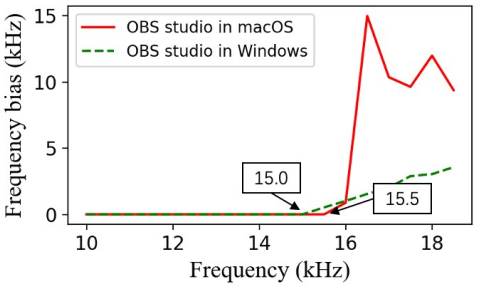
- We design a lightweight and universal event-synchronizing framework. This framework requires no modification to traditional live streaming infrastructure. It also applies to multiple streaming protocols on both desktop and mobile devices.
- We propose to embed metadata into acoustic CSS signals, which not only brings no interference to audiences but is also easy to detect and demodulate for synchronization.
- We implement our system, LSync, on various kinds of devices. The extensive experiments show that the audio signal process is fast and real-time, which takes up less than 6% in average of the audio buffer length among all tested devices and brings at most $96\,\mathrm{ms}$ time delay. In addition, LSync provides a data rate of $156.25\,\mathrm{bps}$ at best, which outperforms recent works.

The rest of the paper is organized as follows. Section II analyzes several factors that impact our choice of acoustic channels and signal modulation. Section III presents the CSS technique and how we leverage it to design inserted packets. We introduce an algorithm for packet synchronization and demodulation in Section IV Through extensive experimental results, we evaluate the performance of the proposed system in Section V. In Section VI, we review the previous work and briefly evaluate the performance. Finally, we conclude the paper in Section VII.

II. Channel and Modulation Selection



(a) Audio power attenuation



(b) Audio frequency bias

Fig. 2: The impact of OBS Studio's AAC encoder on macOS and Windows on audio power attenuation and frequency bias with different frequencies.

The design of hidden acoustic signals should be inaudible and easy to be demodulated, which is highly dependent on the selection of the acoustic carrier frequencies and the modulation method.

A. Carrier Frequencies Selection

As we discussed in the introduction, we should select a proper audio band for embedding the information. The selected band should neither be filtered out by an AAC encoder with a commonly used bit rate nor interfere with content audio.

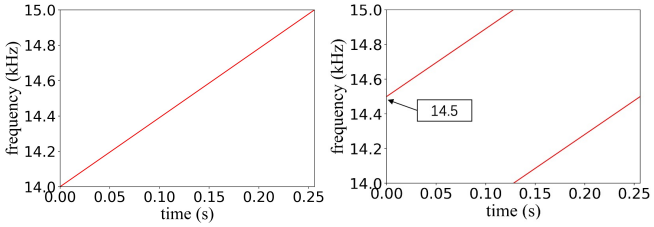
AAC encoder. AAC is the *de facto* audio codec of live streaming [10]. The first thing we need to figure out is how the AAC encoder influences our carrier channel selection. In our experiment, we choose a widely used live-streaming broadcast tool called OBS Studio [11]. The audio should pass its AAC encoder before it is transmitted through the network. The bit rate of the AAC encoder is the most critical parameter, which determines the audio quality. The higher the bit rate is, the better the audio quality will be. However, the more network resources the audio will cost. What is more, the bit rate also determines the bandwidth, *i.e.,* the low-pass filter cutoff, of the encoded audio. This is because the codec reduces the audio bandwidth and modifies the stereo image to keep the most audible frequencies [12]. To study how an AAC encoder affects the audio bandwidth, we utilize OBS Studio in both macOS and Windows with a bit rate of 96k, a commonly used one for live streaming, to broadcast several single tone audio clips with different frequencies ranging from $10\,\mathrm{kHz}$ to $18.5\,\mathrm{kHz}$. Next, we collect the received audio clips and analyze their power decline. Fig. 2a shows that the remained bandwidths are below $15.5\,\mathrm{kHz}$ and $15\,\mathrm{kHz}$ on macOS and Windows, respectively. To further reveal the AAC encoder's influence on audio with different frequencies, we analyze received audio clips with the FFT, find the frequency with maximized power, and calculate the frequency bias towards its original single tone frequency. As shown in Fig. 2b, the audio clips with a frequency higher than $15.5\,\mathrm{kHz}$ on macOS and $15\,\mathrm{kHz}$ on Windows suffer the frequency bias after AAC encoding. It concludes that only the channels with frequency lower than $15\,\mathrm{kHz}$ are available for us to embed data since audio with higher frequency is likely to be filtered out by the AAC encoder, and the embedded data in a higher frequency channel would be lost.

The content audio. Another critical requirement of the embedded acoustic signal is that it should not interweave with the original audio. This prevents the content from being jammed by the embedded signals and also makes the demodulation easier. Previous study [13] shows that the frequencies of the ambient sound in daily life usually lie below $14\,\mathrm{kHz}$.

Based on the above observations, the best choice is the band between $14\,\mathrm{kHz}$ and $15\,\mathrm{kHz}$. However, this is an audible band, making the modulation method selection quite challenging. The selection of the modulation method is discussed in the following section.

B. Modulation Method Selection

Since the embedded signals are audible, the selected modulation method should ensure that these signals are easy to cancel. Besides, even though the ambient sound hardly lies in the band between $14\,\mathrm{kHz}$ and $15\,\mathrm{kHz}$, the interference is still possible. Therefore, the modulation method should also resist the potential interference [14].



(a) Baseline chirp symbol (b) Shifted chirp symbol

Fig. 3: CSS symbol spectrogram

To cancel the embedded signals, we leverage the Web Audio API [15], a high-level Web API for processing and synthesizing audio in web applications, to filter the audio before playing. Specifically, Web Audio API provides the `BiquadFilterNode` [16], a processor implementing very common low-order filters. Although we can use the `BiquadFilterNode` to add a lowpass filter to attenuate the frequencies above the cutoff, *i.e.,* $14\,\mathrm{kHz}$, it is infeasible to wipe out the embedded signals between $14\,\mathrm{kHz}$ and $15\,\mathrm{kHz}$. This is because the filter is a standard second-order resonant lowpass filter with 12dB/octave roll-off instead of an ideal lowpass filter. For instance, supposing that the cutoff frequency is set to $7\,\mathrm{kHz}$, an audio clip with a frequency of $14\,\mathrm{kHz}$ would have $-\log_2\left(\frac{14\mathrm{kHz}}{7\mathrm{kHz}}\right) \times 12\mathrm{dB} = -12\mathrm{dB}$ attenuation instead of being entirely eliminated. Thus, even with the help of the `BiquadFilterNode`, we still have to limit the power of embedded signals to quite a low level so that it could be inaudible after it gets attenuated by the filter.

Based on the above discussion, we need a modulation method that ensures that the modulated signals are able to be demodulated at a low power level and resilient to interference. In LSync, we select the chirp spread spectrum (CSS) technique to modulate data. CSS uses wideband linear frequency modulated chirp pulses to encode information, making it robust to channel noise [17] and easy to be demodulated even if its power level stays low [18]–[20].

III. CSS Symbol and Frame Design

Based on the CSS technique, we propose to encode metadata into chirps during the signal modulation process and form a complete packet with other significant frame components. The encoding and frame designs are elaborated on below.

A. CSS Symbol Design

The CSS technique is ideal for applications that require low power usage and need relatively low data rates in digital communications. Unlike previous works [13], [21], [22], which use chirp-BOK or QOK to encode data, we leverage an approach, which manipulates the starting frequency offset of a baseline up-chirp to form various shaped chirps and represent different

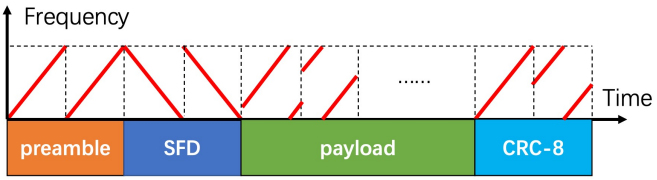


Fig. 4: The structure of the embedded packet

numbers to further improve the data rate and speed up the demodulation process.

As shown in Fig. 3a, the frequency of a baseline up-chirp increases linearly from $fc = 14\,\text{kHz}$, the lower bound of the band we select, to the upper bound $fc + BW$, where BW represents the bandwidth. Suppose that the duration of a chirp is T (0.256 s as the example in Fig. 3). Then the time-domain function for baseline up-chirp can be expressed as

$$C(t) = \sin\left(2\pi(fc + \frac{BW}{2T}t) \cdot t\right).$$

In order to make the alignment of a packet with the time window more precisely during the demodulation process, which would be elaborated in Section IV, we set T to be a power of 2 ms including 32 ms, 64 ms, 128 ms, and 256 ms.

Given the frequency shift f of a baseline up-chirp, the time-domain function for the resulted symbol is $\sin\left(2\pi(fc + f + \frac{BW}{2T}t) \cdot t\right)$. Then all the frequencies higher than $fc + BW$ will be folded back to fc as shown in Fig. 3b. We introduce a parameter BN, which is a positive integer and represents how many bits a chirp is able to encode. In our design, there are 2^{BN} different equally-distributed shifted starting frequencies, which results in 2^{BN} uniformly shaped up-chirps, and each one represents a unique number so that one specific up-chirp represents an BN bits number. In particular, the **baseline up-chirp** shown as Fig. 3a represents 0. And a **shifted chirp** whose starting frequency is $fc + f$ represents the number n, where

$$n \times \frac{BW}{2^{BN}} = f. \tag{1}$$

Thus for the chirp shown in Fig. 3b where its frequency shift is half of the bandwidth, it represents 2^{BN-1}.

With T and BN, we can calculate the bit rate of chirps as

$$R_b = \frac{BN}{T} \text{ bps}.$$

Therefore, the smaller T is, and the greater BN is, the better the data rate will be. However, the decrease of T and increase of BN results in lower reception sensitivity, which means that at the same reception power level, the chirp with a lower T and a larger BN may be unable to be demodulated while the chirp with a larger T and a lower BN can. The details are explained in Section V. Hence, we have to select a proper T and BN setting to meet the data rate requirements and the need for demodulation under low embedded signal power.

To demodulate, we leverage a baseline down-chirp, where the frequency sweeps decreasingly and its time-domain function is $C^*(t) = \sin\left(2\pi(BW - \frac{BW}{2T}t) \cdot t\right)$. By multiplying a baseline down-chirp, each shifted up-chirp is concentrated on a single frequency, and the result can be calculated as

$$\sin\left(2\pi(fc + f + \frac{BW}{2T}t) \cdot t\right) \sin\left(2\pi(BW - \frac{BW}{2T}t) \cdot t\right)$$
$$= \frac{1}{2}\Big[\cos\left(2\pi(fc + f + BW) \cdot t\right)$$
$$- \cos\left(2\pi(fc + f - BW + \frac{BW}{T}t) \cdot t\right)\Big],$$

where the first part of the result is centralized on a specific frequency $fc+f+BW$ while the second part spread in a wide frequency band. With the help of the FFT, we can find a peak in the spectrum, analyze the frequency shift f, and decode the data that the shifted up-chirp indicates.

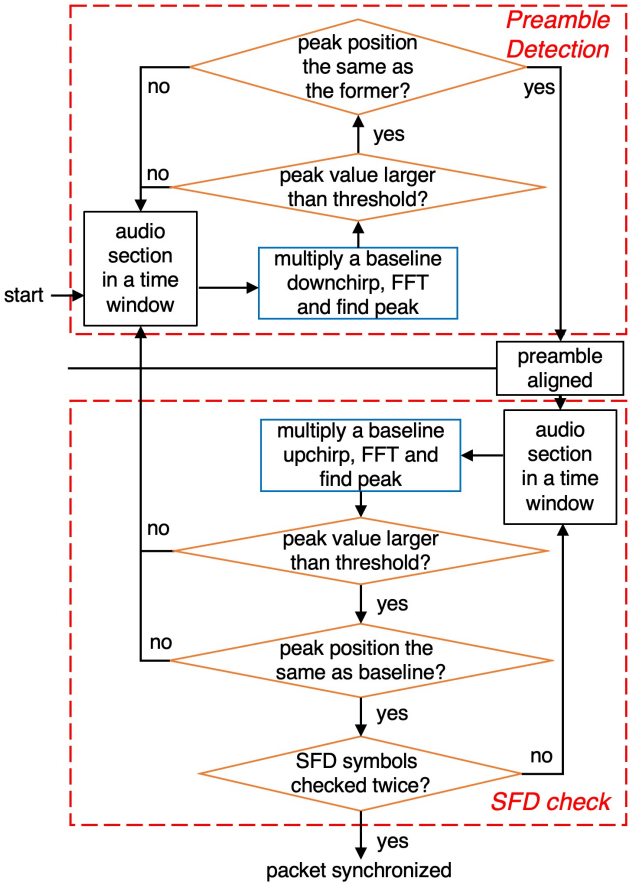


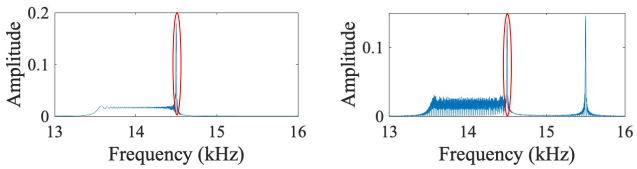
Fig. 5: Packet synchronization process

B. Frame Design

As shown in Fig. 4, a whole packet frame contains two baseline up-chirps as the preamble, two baseline down-chirps as the start of frame delimiter (SFD), several shifted up-chirps as payload, and two shifted up-chirps for CRC-8 symbols.

Preamble and SFD. At the beginning of a packet, the preamble will be used for packet detection. We use two baseline up-chirps for the preamble due to the trade-off between the detection reliability and the efficiency. Using more than two

Fig. 6: Align chirps of the preamble with time window



(a) FFT result of the first window (b) FFT result of the second window

Fig. 7: FFT result of two consecutive time windows for the preamble, where two peaks share the same position

baseline up-chirps would make the detection more reliable but bring more redundancy to a packet. In comparison, if we only set one baseline up-chirp as the preamble, false-positive detection is much more likely to occur since a single baseline up-chirp may appear in the payload or the CRC part other than preamble. According to our experiments, two adjacent baseline up-chirps are much less common.

To further confirm that a new packet is found as well as to separate the preamble and payload, we add the start of frame delimiter (SFD), two continuous baseline down-chirps after the preamble because the down-chirp does not exist in any other part of a packet. Using two instead of one single down-chirp is to increase the robustness.

Payload. The payload consists of a few shifted up-chirps, depending on how much message the packet aims to deliver and the BN setting. According to our experiment in Section V, BN should be between 4 and 8 to guarantee an acceptable data rate between 31.25 bps to 156.25 bps and the possibility to demodulate the embedded signal. In LSync, we encode timestamp into the payload, and the number of chirps to encode hour, minute, second, and millisecond respectively should depend on the BN setting. For instance, if the BN is no less than 6, two chirps should be used to represent milliseconds with one single chirp for hour, minute, and second respectively.

CRC. To ensure the integrity of the payload, we use CRC-8 for bit error detection [23]. Since we set BN to be lower than 8 and greater than 4, we need to leverage two chirps to encode a CRC-8 symbol.

IV. Packet Synchronization and Decoding

Before the audio stream is played, the embedded data must be decoded on the audience side, which is the web browser in LSync. The key point is how we can process the signal in real-time with limited web browser resources. It would be time-consuming and resource-consuming to leverage similar approaches in recent work [13], [21], [22], since they require storing a relatively long audio section that contains at least one whole packet. We intend to process the audio stream with a sliding time window whose length is the same as a chirp's to achieve real-time analysis and save storage resources. The demodulation process consists of two main parts: packet synchronization and decoding.

A. Packet Detection and Synchronization

The first aspect is to locate the exact start point of an embedded packet in the audio stream. The process comprises three steps: preamble detection, packet aligning, and SFD check. Fig. 5 describes the detailed packet synchronization process.

Preamble detection. The preamble, as the start section of a packet, consists of two continuous baseline up-chirps. Thus, in each of the two successive time windows, we multiply a baseline down-chirp with the audio section, perform FFT and find the peak to detect preamble. Even if the preamble is misaligned with the time windows like the left part of Fig. 6, the FFT results have the same peak position as shown in Fig. 7(We transform the original FFT result to the frequency domain) because half of the shifted chirp in the second window is the same as the chirp in the first window in terms of the frequency domain. If the two peak values exceed a threshold, which means that chirps do exist and share the same peak position after being multiplied with a down-chirp and FFT operation in consecutive time windows, we consider it as a part of a preamble, and then we use the peak position of the second window to align the packet with time window.

Packet alignment. To align the packet, we need to calculate how many sample points of the signal should be moved forward so that each chirp of the signal could be aligned with the time window. In the left part of Fig. 6, which shows the misalignment, we express the starting frequency of the shifted chirp in the second window as f', the starting frequency of a baseline up-chirp as f_0 while the total number of a baseline up-chirp's sample points as $chirp\_n$. $chirp\_n$ equals $T \times fs$, where fs denotes the sample rate of the audio and T denotes the duration of a chirp. The number of sample points to be moved can be calculated as

$$n = \frac{(f' - f_0) \times chirp\_n}{BW}.$$

We multiply a baseline down-chirp with it and perform FFT for the audio signal in the second time window. There are two peaks as in Fig. 7b , and we select the greater peak position of the original FFT result as id', which representing the left part of that shifted chirp and indicates its starting frequency. Also, we make the same operation for baseline up-chirp and there is only one peak in its FFT result. Let id_0 be the single peak position and fft_n be the sample number of FFT. Note that

$$(f' - f_0) = \frac{(id' - id_0) \times fs}{fft\_n}.$$

Thus, n can be computed as

$$n = \frac{(id' - id_0) \times fs \times fs \times T}{BW \times \mathit{fft\_n}}.$$

To ensure the precision of alignment, the computation result should be a non-negative integer. Consider that the configuration we used in our experiment is $fs = 48\,\mathrm{kHz}$, $BW = 1\,\mathrm{kHz}$ and $\mathit{fft\_n}$ should be an integer that is a power of 2 and no less than $\mathit{chirp\_n}$. Supposing that T is a power of $2\,\mathrm{ms}$ and denoted as $2^l \times 10^{-3}s$, $\mathit{chirp\_n}$ equals $2^{l+4} \times 3$. Thus we set $\mathit{fft\_n}$ to be 2^{l+6} and n can be recomputed as: $n = (id' - id_0) \times 36$, which is definitely an non-negative integer. Otherwise, if T is not a power of $2\,\mathrm{ms}$ and is like $100\,\mathrm{ms}$ and $\mathit{chirp\_n}$ equal 4800, $\mathit{fft\_n}$ should be at least $8192 = 2^{13}$. Then n equals $(id' - id_0) \times 225/8$, which is possibly not an integer if $(id' - id_0)$ is not a multiple of 8.

By moving n sample points of the signal forward, we are able to align the signal with the time window chirp by chirp.

SFD check To make sure that the consecutive two baseline up-chirps we have found are exactly the beginning of a packet, we should then examine whether the SFD follows them. Since the packet has been aligned, similarly, we multiply a baseline up-chirp with the audio signal of each time window, perform the FFT, and find the peak. If the result is accurately the same as a baseline down-chirp would produce in both windows, we confirm that the SFD is found after the preamble and a new packet is successfully discovered and aligned.

B. Packet Decoding

Once completing the SFD check, it is quite easy to decode data buried in the packet. In each time window, we multiply a baseline down-chirp with the shifted chirp, perform the FFT, and find the peak. Let f be the starting frequency of the shifted chirp and f_0 be the starting frequency of the baseline up-chirp. According to Eq. 1, we have

$$(f' - f_0) = n \times \frac{BW}{2^{BN}},$$

where n is the encoded data. Let id be the greater peak position in the FFT result of the specific shifted chirp and id_0 be the FFT result of a baseline up-chirp. Then n can be decoded as

$$n = \frac{(id - id_0) \times fs \times 2^{BN}}{\mathit{fft\_n} \times BW}.$$

When we finish decoding the whole packet, the CRC code appended to the payload should be used to validate whether the payload is correctly received and decoded.

V. Implementation and Evaluation

As shown in Fig. 1, LSync consists of three main components, the signal modulator at the streamer side, the live streaming server for receiving and distributing the stream, and the audio processing and demodulator at the audience side. To evaluate the performance of the proposed method, these components are implemented as follows.

Streamer side. On the streamer side, we leverage OBS Studio to record live audio/video and push them to the live streaming server through Real-Time Messaging Protocol (RTMP) [24]. OBS Studio allows multiple audio inputs. Therefore, we introduce a virtual audio cable with VB-Cable [25] and generate the modulated signals to OBS Studio via this cable. The modulated signals are generated following the frame design, with the current timestamp as the payload. The signals are generated every $5\,\mathrm{s}$ and then mixed with the live audio and transmitted to the live streaming server together. We set the sample rate to $48\,\mathrm{kHz}$ and the bit rate of audio to $96\,\mathrm{kHz}$, which are general settings for a daily live broadcast. We also turn down the power of the virtual cable to $-62.5\,\mathrm{dB}$, which is nearly the extreme limit for embedded signal demodulation reliability through practice. Our system does not hamper a normal live broadcast process, and it works well in both macOS and Windows. A more user-friendly software plug-in can further replace the virtual audio cable for OBS Studio in the future.

Audience side. We develop a web application written in JavaScript to process audio in real-time before being played on the audience side. The audio is analyzed with Web Audio API. We use `MediaElementAudioSourceNode` interface [26] to access the audio from an HTML `<video>` element for later processing. Then we leverage `ScriptProcessorNode` interface [27] to handle the audio buffer. When the buffer gets full, a callback function is invoked, where we demodulate the signals as our design in Section IV. We use `fft.js`, an implementation of Radix-4 FFT, to perform FFT operations in signal processing. Lastly, we use the `BiquadFilterNode` to filter the audio, and then it gets played. Since most modern browsers support Web Audio, the application is available for Google Chrome, Microsoft Edge, and Mozilla Firefox on Windows and macOS. We also perform evaluations on Android using Google Chrome and Mozilla Firefox. Unfortunately, due to the bug of Webkit [28], Safari on both macOS and iOS is not tested yet, but once the bug is fixed, this method should work as expected.

Live streaming server. The live streaming server is implemented using Node.js. With Node.js package `node-media-server` [29], we launch an RTMP server [24] to receive the live stream pushed by the streamer. In addition, the server also outputs the stream with various formats and protocols, including HTTP-FLV [30], HLS [31], and DASH [32].

The rest of this section presents the experiment results. The devices that we employ include a PC with an Intel Core i7-10710U CPU running Windows 10, a MacBook with an Intel Core i7-9750H CPU running macOS 11.4, and a Redmi Note 8 Pro mobile phone with a Helio G90T CPU running Android 10. Most experiments are conducted in a static indoor office environment. The power of the virtual audio cable is set to be $-62.5\,\mathrm{dB}$ so that the embedded signal after processed can be totally unaware of by audiences even in such a silent environment.

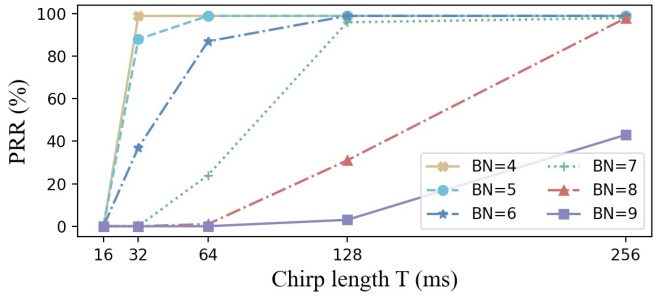


Fig. 8: Packet reception ratio under different BN and T settings

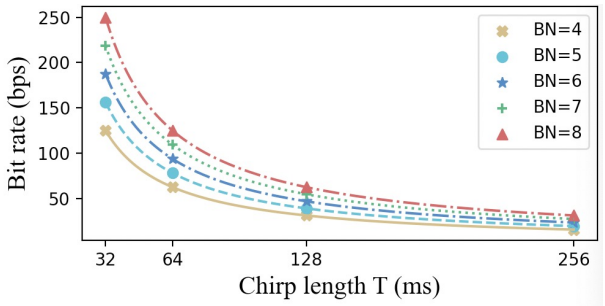


Fig. 9: Data rate under different BN and T settings

A. Reception Accuracy

We first present the reception accuracy on the audience side under different BN and T settings. We generate the modulated packet every 5 s. Therefore, the longest T among the experiments is set to 256 ms. Supposing that T is greater than 256 ms, *e.g.,* 512 ms, the length of a total packet will be over 5.63 s, longer than the sending interval. Meanwhile, to simplify the encoding process, the lowest BN is set to 4 so that two chirps are enough to encode CRC-8 for all cases.

Fig. 8 shows the relationship among packet reception ratio (PRR), BN, and the chirp length. A too short chirp length T, *e.g.,* 16 ms, makes it impossible to decode. Instead, a longer chirp length T results in a higher PRR. If a chirp with a longer duration in time domain multiplies with a down-chirp, more energy would be centralized on a shifted frequency after the FFT, making it easier to be correctly demodulated under the same low audio power level. Meanwhile, a large BN, *e.g.,* 9, is also not usable. A greater BN hampers the PRR because it implies a finer division of the starting frequency for shifted chirps, making it harder to distinguish the different shifted frequencies during the demodulation process.

As a result, to ensure the reception reliability, we should set T to be no less than 32 ms and BN lower than 9.

B. Data Rate

In section III, we have shown that the data rate is proportionate to BN and inversely proportional to T, and we depict the relationship in Fig 9. Based on Fig 8 and Fig 9, we summarize the greatest data rate that our system can provide when PRR is greater than 95% in Table I. We list the result

TABLE I: Data rate under different BN and T settings with a PRR over 95%

T (ms)	32	64	128	256
BN	4	5	7	8
Data rate (bps)	125.00	78.125	54.69	31.25
PRR	99%	99%	96%	98%

TABLE II: Data rate under different BN and T settings with a PRR over 85%

T (ms)	32	64	128	256
BN	5	6	7	8
Data rate (bps)	156.25	93.75	54.69	31.25
PRR	88%	87%	96%	98%

when T ranges from 32 ms to 256 ms with the setting of BN to achieve the data rate. To compare with our system, the previous work in Tagscreen [13] reaches a data rate of 50 bps with 2 kHz bandwidth, which is the most efficient one among recent works. While LSync can provide a 125 bps data rate that outperforms their performance when the chirp length is 32 ms, and BN is 4. Besides, the bandwidth we could use is 1 kHz, only half of that in their implementation. What is more, with a little sacrifice on reception reliability, we can boost the data rate to 156.25 bps when the chirp length is 32 ms, and BN is 5 as shown in Table II.

It is worth noting that the selection of the parameters BN and T should take the data we are to encode into account. For instance, in LSync, it is optional to use $T = 64ms$, $BN = 6$ rather than the configuration with the greatest data rate where the setting is $T = 32ms$, $BN = 5$. With the former configuration, we can encode minute and second into one chirp, respectively. While for the latter, two chirps are needed respectively, which decreases the virtual data rate instead.

C. Versatility

Then, we discuss the versatility of our system. On the streamer side, we evaluate the performance of LSync on different operating systems with the configuration shown in Table II. In these tests, we run the decoding algorithm on Google Chrome. The results (shown in Fig. 10) indicate that the streamer side implemented on macOS provides a slightly better PRR performance compared to Windows. The better performance results from the feature of the low bit rate AAC encoder inside OBS Studio according to Section III, where the cut-off frequency of the encoder on macOS is a bit higher than that on Windows. Therefore the embedded signal is better preserved. Even so, the streamer side on Windows can provide an acceptable PRR in most cases with the last three settings in Table II.

While on the audience side, we would like to know if LSync works well with various protocols and playing terminals. HTTP-FLV, DASH, and HLS are the three most widely used live streaming protocols that transmit media content over HTTP. To evaluate the influence that different streaming protocols and browsers have, we measure the PRR with

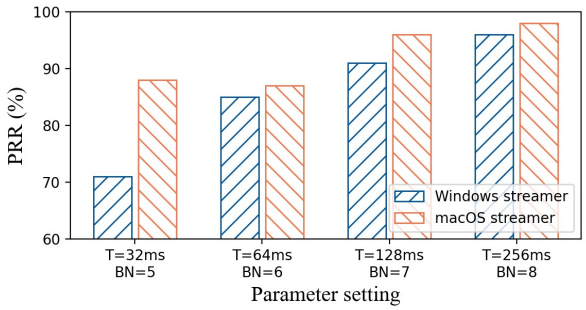


Fig. 10: Packet reception ratio with different streamer environments

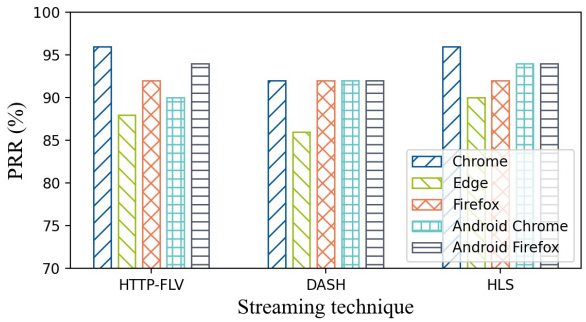
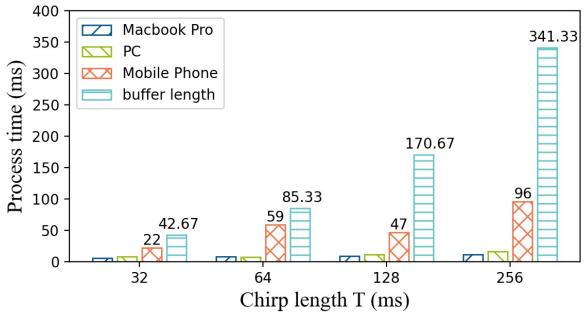
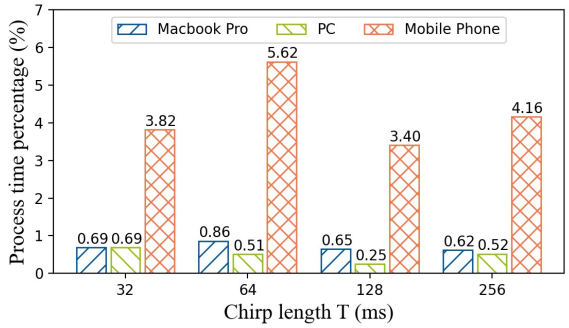


Fig. 11: Packet reception ratio with different streaming protocols and browsers on the audience side



(a) Longest processing time for each buffer compared with buffer length



(b) Percentage between mean processing time for each buffer and the buffer length

Fig. 12: Processing time for each audio buffer in diverse devices under different settings

different mainstream browsers and the above three protocols. Fig. 11 demonstrates the result with the configuration of $T = 128ms,\ BN = 7$.

In general, the PRR is 92% on average among all settings. On desktop devices, the performances are quite similar in the aspect of streaming protocols, except that the PRR using DASH is slightly lower than the other two protocols. This may be due to the fragment lengths of different protocols are different.

LSync works best on Google Chrome, where the PRRs are 96%, 92%, and 96% when tested with HTTP-FLV, DASH, and HLS, respectively. Nevertheless, the PRRs slightly drop to 88%, 86%, 90% when the audience service runs on Microsoft Edge.

On mobile phones, LSync also works well with the three streaming protocols, *i.e.,* HTTP-FLV, DASH, and HLS. The PRRs on Android Chrome are 90%, 92%, and 94%, respectively, and on Android Firefox are 94%, 90%, and 94%, respectively. Although the tests failed to run on iPhone, the results show that this method does work on ordinary mobile devices, and it should work on iPhone in the future.

D. Process Delay

According to ITU-R BT.1359 [33], the lip-sync error can be accepted if the delay is less than 185 ms. Referring to this standard, we aim to build an event-synchronizing system with a delay of less than 185 ms. Previous work in [13], [21], [22], [34] requires the reception of the whole packet before processing, which makes the delay from 600 ms to 1600 ms. In LSync, we process the signal chirp by chirp to eliminate the delay. With the Web Audio API, we use a *double buffer design* to load audio signals while processing them: during loading the next section of audio signals, the current section is copied to another buffer and processed. In order to test how long the audio analysis process takes for each buffer in various types of devices, we carry out the experiments under the configuration shown in Table II. Fig. 12a exhibits the results of the longest processing duration for each audio buffer compared with the buffer length. The greater T is, the longer the chirp length will be, and thus the buffer length increases. For all devices, the processing time for a single buffer is no longer than 22 ms, 59 ms, 47 ms and 96 ms, respectively with different T, which is less than each single buffer. Therefore, during a period when the buffer is receiving the next audio signal section, the last received section can be entirely processed. The delay is also less than 185 ms, satisfying our requirement. To further illustrate our processing efficiency, we calculate the mean processing time for different settings. Fig. 12b presents the result as a percentage between the processing time and the buffer length. The average processing time for each buffer on desktop devices is less than 1% of the buffer length. While on mobile phones, the processing takes up a bit more time but less than 6% on average of each buffer. According to our

experimental results, not only real-time signal processing is achieved, but also it is highly efficient.

VI. Related Work

Many researchers have well studied hidden acoustic or visual channel communications to provide helpful side information to audiences [35], [36]. In addition, real-time communication protocols for the web like WebRTC emerge as well which enable nearly real-time data transmission for users. We discuss them from the following three aspects.

A. Hidden Visual Channel Communication

Inframe++ [37] leverages the spatial-temporal flicker-fusion property of the human visual system and the fast frame rate of the modern display to embed data onto video content through complementary frame composition. It achieves a $150\,\mathrm{kbps}$ to $240\,\mathrm{kbps}$ data rate at 120 fps over a 24' LCD monitor with one data frame per 12 display frames, but noticeable flicker remains. Hilight [38] encodes bits into the pixel translucency change, which supports a low bit rate of $1.1\,\mathrm{kbps}$ but reduces flicker to unnoticeable levels. ImplicitCode [39] combines both techniques to simultaneously achieve invisibility and a high capacity, which is $12\times$ that of HiLight. In TextureCode [40], they utilize spatial content-adaptive encoding techniques to achieve both a high goodput of $22\,\mathrm{kbps}$ and minimal flicker. Uber-in-light [41] encodes the data as complementary intensity changes over different color channels for any screen content and significantly improved transmission accuracy compared to Hilight.

Although the above approaches provide relatively high throughput, they suffer from a few disadvantages in live streaming. These approaches become less reliable due to possible frame loss in the live streaming process. Besides, they require a high frame rate, generally over 120 fps, which is not available in standard live broadcast tools and typical play terminals. For instance, the OBS Studio supports 60 fps at most.

B. Hidden Acoustic Channel Communication

Hidden acoustic communication has also been explored for years [13], [21], [22], [34], [42]. PhoneEar [42] uses frequency-shift keying (FSK) modulation to encode information in frequencies from $17\,\mathrm{kHz}$ to $20\,\mathrm{kHz}$, which transmit data at the speed of $8\,\mathrm{bps}$. Hyewon Lee *et al.* adopt chirp binary orthogonal keying (BOK) to encode data in [21]. They choose a $19.5\,\mathrm{kHz}$ to $22\,\mathrm{kHz}$ band for inaudible acoustic communication and achieve a data rate of $16\,\mathrm{bps}$. Soonwon Ka *et al.* leverage chirp quaternary orthogonal keying (QOK) for modulation in an $18.5\,\mathrm{kHz}$ to $19.5\,\mathrm{kHz}$ band to deliver information at $15\,\mathrm{bps}$ in [22]. By leveraging masking effects of the human auditory system, Dolphin [34] adopts OFDM for modulation in frequencies of $8\,\mathrm{kHz}$ to $20\,\mathrm{kHz}$ and achieves a high data rate of $500\,\mathrm{bps}$. Tagscreen [13] inserts hidden sound markers (*i.e.,* binary orthogonal chirps at $18\,\mathrm{kHz}$ to $20\,\mathrm{kHz}$) into audio for data communication and design an efficient decoding algorithm, which reduces computations.

The aforementioned work has a few limitations. The greatest one is that all of them leverage near-ultrasound bands with frequencies higher than $17\,\mathrm{kHz}$ to embed data. However, based on our study in Section II, such a high-frequency acoustic band would be cut off by an AAC encoder of standard live broadcast software with a generally used bit rate of 96k. In addition, few of them mentioned their signal demodulation delay while the best one [34] claims the delay of $600\,\mathrm{ms}$, which is not good enough for real-time signal processing.

C. WebRTC

WebRTC is an open-source project released by Google in 2011. By establishing a peer-to-peer connection, both clients can send video, voice, and generic data to each other [43]. It supports sub–500 milliseconds of real-time latency, which makes it the fastest protocol on the market. Such a low latency reduces the burden of synchronizing video stream and other media content and makes it a fundamental transmission protocol for lots of video meeting and chat applications including Google Meet [44] and Facebook Messenger [45].

Nevertheless, there are several disadvantages of WebRTC that render it not suitable for our target applications. Basically, WebRTC is a P2P protocol. The limited bandwidth resource restricts the number of the audience side that the streamer side would like to directly communicate with, without sacrificing the video quality. What is worse, the streamer side should serve as a CDN itself to deliver streams to the audiences which requires heavy resource occupation of the computer. However, in the scenario of live streaming, there might be over a hundred thousand audiences watching the live at the same time and thus make it impossible to establish direct communication between the streamer and audiences. Nonetheless, adding a CDN in between instead sacrifices the low latency feature of WebRTC. Besides, there is a serious security concern in browsers that support WebRTC, which is exposing the user's internal IP address to the web. This problem still surfaces on Mozilla Firefox [46].

VII. Conclusion

In this work, we present LSync, for the synchronization of traditional live streams and other media content. A key innovation is to insert hidden signal in an audible band and eventually recover the embedded data and makes no disturbance to the audience as well through elaborate signal modulation and processing. We leverage the CSS technique to modulate signals, which makes sure that the signal can be demodulated at a very low power level. We achieve completely real-time signal processing and improve the data rate to $156.25\,\mathrm{bps}$ with a bandwidth of only $1\,\mathrm{kHz}$. We implement both the streamer side and the audience side with various streaming protocols like HTTP-FLV, DASH, and HLS at several mainstream browsers on desktop and mobile devices, which validate the versatility of LSync.

Acknowledgments

This work is supported in part by the National Key R&D Program of China under grant 2021YFB2900100.

REFERENCES

[1] Z. Yin, C. Wu, Z. Yang, and Y. Liu, "Peer-to-Peer Indoor Navigation Using Smartphones," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 5, pp. 1141–1153, 2017.

[2] W. Gu, Z. Yang, L. Shangguan, W. Sun, K. Jin, and Y. Liu, "Intelligent Sleep Stage Mining Service with Smartphones," in *Proceedings of the 2014 ACM international Joint Conference on pervasive and ubiquitous Computing*, 2014, pp. 649–660.

[3] "Rain Classroom," https://www.yuketang.cn/, (Accessed on April 21, 2021).

[4] "HQ Trivia," https://hqtrivia.com/legal/faq.html, (Accessed on June 20, 2021).

[5] "Live Streaming Process," https://www.dacast.com/blog/what-is-live-streaming/, (Accessed on June 24, 2021).

[6] "Solution of Alibaba Cloud for Live Quiz," https://help.aliyun.com/document_detail/67427.html, (Accessed on April 21, 2021).

[7] "ID3," https://id3.org/, (Accessed on June 20, 2021).

[8] "Amazon IVS User Guide," https://docs.aws.amazon.com/ivs/latest/userguide/ivs-ug.pdf, (Accessed on June 20, 2021).

[9] B. Reynders and S. Pollin, "Chirp Spread Spectrum as a Modulation Technique for Long Range Communication," in *Proceedings of the 2016 Symposium on Communications and Vehicular Technologies*. IEEE, 2016, pp. 1–5.

[10] M. Bosi, K. Brandenburg, S. Quackenbush, L. Fielder, K. Akagiri, H. Fuchs, and M. Dietz, "ISO/IEC MPEG-2 Advanced Audio Coding," *Journal of the Audio engineering society*, vol. 45, no. 10, pp. 789–814, 1997.

[11] "OBS Studio," https://obsproject.com/, (Accessed on June 20, 2021).

[12] "SBR White Paper," http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/drm/SBR_White_Paper_v1.pdf, (Accessed on July 24, 2021).

[13] Q. Lin, L. Yang, and Y. Liu, "Tagscreen: Synchronizing Social Televisions through Hidden Sound Markers," in *Proceedings of the 36th IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[14] F. Dang, P. Zhou, Z. Li, and Y. Liu, "NFC-enabled Attack on Cyber Physical Systems: A Practical Case Study," in *Proceedings of the 36th IEEE Conference on Computer Communications Workshops*, 2017, pp. 289–294.

[15] "Web Audio API," https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API, (Accessed on June 24, 2021).

[16] "Web Audio BiquadFilterNode," https://developer.mozilla.org/zh-CN/docs/Web/API/BiquadFilterNode, (Accessed on June 20, 2021).

[17] A. Berni and W. Gregg, "On the Utility of Chirp Modulation for Digital Signaling," *IEEE Transactions on Communications*, vol. 21, no. 6, pp. 748–751, 1973.

[18] Y. Lin, W. Dong, Y. Gao, and T. Gu, "SateLoc: A Virtual Fingerprinting Approach to Outdoor LoRa Localization Using Satellite Images," *ACM Transactions on Sensor Networks*, vol. 17, no. 4, pp. 1–28, 2021.

[19] H. Jiang, J. Zhang, X. Guo, and Y. He, "Sense Me on the Ride: Accurate Mobile Sensing over a LoRa Backscatter Channel," in *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2021, pp. 125–137.

[20] D. Lin, Q. Wang, W. Min, J. Xu, and Z. Zhang, "A Survey on Energy-efficient Strategies in Static Wireless Sensor Networks," *ACM Transactions on Sensor Networks*, vol. 17, no. 1, pp. 1–48, 2020.

[21] H. Lee, T. H. Kim, J. W. Choi, and S. Choi, "Chirp Signal-based Aerial Acoustic Communication for Smart Devices," in *Proceedings of the 34th IEEE Conference on Computer Communications*. IEEE, 2015, pp. 2407–2415.

[22] S. Ka, T. H. Kim, J. Y. Ha, S. H. Lim, S. C. Shin, J. W. Choi, C. Kwak, and S. Choi, "Near-ultrasound Communication for Tv's 2nd Screen Services," in *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, 2016, pp. 42–54.

[23] "Cyclic Redundancy Check," https://en.wikipedia.org/wiki/Cyclic_redundancy_check, (Accessed on June 24, 2021).

[24] "Real-time Messaging Protocol (RTMP) Specification," https://www.adobe.com/devnet/rtmp.html, (Accessed on July 27, 2021).

[25] "VB-Cable Virtual Audio Device," https://vb-audio.com/Cable/, (Accessed on June 20, 2021).

[26] "MediaElementAudioSourceNode," https://developer.mozilla.org/en-US/docs/Web/API/MediaElementAudioSourceNode, (Accessed on July 27, 2021).

[27] "ScriptProcessorNode," https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode, (Accessed on July 27, 2021).

[28] "Audio Passed through WebAudio is Delayed and Glitchy on Safari," https://bugs.webkit.org/show_bug.cgi?id=221334, (Accessed on July 27, 2021).

[29] "Node Media Server," https://github.com/illuspas/Node-Media-Server, (Accessed on June 24, 2021).

[30] "Flash Video," https://en.wikipedia.org/wiki/Flash_Video, (Accessed on June 24, 2021).

[31] "HTTP Live Streaming," https://www.encoding.com/http-live-streaming-hls/, (Accessed on June 20, 2021).

[32] "Dynamic Adaptive Streaming over HTTP," https://en.wikipedia.org/wiki/Dynamic_Adaptive_Streaming_over_HTTP, (Accessed on June 24, 2021).

[33] "BT.1395," https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.1359-1-199811-I!!PDF-E.pdf, (Accessed on June 20, 2021).

[34] Q. Wang, K. Ren, M. Zhou, T. Lei, D. Koutsonikolas, and L. Su, "Messages Behind the Sound: Real-time Hidden Acoustic Signal Capture with Smartphones," in *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, 2016, pp. 29–41.

[35] X. Chen, X. Wu, X.-Y. Li, X. Ji, Y. He, and Y. Liu, "Privacy-Aware High-Quality Map Generation with Participatory Sensing," *IEEE Transactions on Mobile Computing*, vol. 15, no. 3, pp. 719–732, 2016.

[36] Z. Yang, L. Jian, C. Wu, and Y. Liu, "Beyond Triangle Inequality: Sifting Noisy and Outlier Distance Measurements for Localization," *ACM Transactions on Sensor Networks*, vol. 9, no. 2, pp. 1–20, 2013.

[37] A. Wang, Z. Li, C. Peng, G. Shen, G. Fang, and B. Zeng, "Inframe++ Achieve Simultaneous Screen-human Viewing and Hidden Screen-camera Communication," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, 2015, pp. 181–195.

[38] T. Li, C. An, X. Xiao, A. T. Campbell, and X. Zhou, "Real-time Screen-camera Communication behind Any Scene," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, 2015, pp. 197–211.

[39] S. Shi, L. Chen, W. Hu, and M. Gruteser, "Reading between Lines: High-rate, Non-intrusive Visual Codes within Regular Videos via Implicitcode," in *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing*, 2015, pp. 157–168.

[40] V. Nguyen, Y. Tang, A. Ashok, M. Gruteser, K. Dana, W. Hu, E. Wengrowski, and N. Mandayam, "High-rate Flicker-free Screen-camera Communication with Spatially Adaptive Embedding," in *Proceedings of the 35th IEEE Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[41] M. Izz, Z. Li, H. Liu, Y. Chen, and F. Li, "Uber-in-light: Unobtrusive Visible Light Communication Leveraging Complementary Color Channel," in *Proceedings of the 35th IEEE Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[42] A. S. Nittala, X.-D. Yang, S. Bateman, E. Sharlin, and S. Greenberg, "Phoneear: Interactions for Mobile Devices that Hear High-frequency Sound-encoded Data," in *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 2015, pp. 174–179.

[43] "WebRTC," https://webrtc.org/, (Accessed on July 27, 2021).

[44] "Google Meet," https://meet.google.com/, (Accessed on July 27, 2021).

[45] "Facebook Messenger," https://www.messenger.com/, (Accessed on July 27, 2021).

[46] "Prevent WebRTC from Leaking Local IP Address," https://github.com/gorhill/uBlock/wiki/Prevent-WebRTC-from-leaking-local-IP-address, (Accessed on July 27, 2021).