LSync: A Universal Timeline-Synchronizing Solution for Live Streaming

Fan Dang, *Member, IEEE, ACM*, Yifan Xu, *Student Member, IEEE*, Rongwu Xu, Xinlei Chen, *Member, IEEE*, and Yunhao Liu, *Fellow, IEEE, ACM*

***Abstract*— The widespread use of intelligent devices and the development of mobile networks have led to the increasing popularity of live-streaming services worldwide. In addition to video and audio transmissions, a wide range of media content is also sent to audiences, such as player statistics for sports streams and subtitles for live news. However, due to the diverse transmission process between live streams and other media content, synchronizing them has become a significant challenge. Unfortunately, existing commercial solutions are not universal, requiring specific server cloud services or CDNs and limiting users' free choices of web infrastructures. To address this issue, we propose a lightweight and universal solution called LSync, which inserts a series of audio signals containing metadata into the original audio stream. Based on the embedded metadata, a well-designed timeline-synchronizing solution helps to synchronize the information stream to the live stream. It brings no modifications to the original live broadcast process and thus fits prevalent live broadcast infrastructures. Evaluations show that the proposed solution reduces the signal processing delay to around 5% of an audio buffer length in mobile phones and ensures real-time signal processing. It achieves a channel utilization of more than 150 bps/kHz in a specific configuration, greatly outperforming recent works. Furthermore, the proposed synchronization mechanism reaches a precision of 24.84 ms on average, which matches people's viewing habits.**

***Index Terms*— Live streaming, synchronization, chirp signal.**

Manuscript received 24 March 2023; revised 23 November 2023 and 13 May 2024; accepted 24 May 2024; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor L. Cai. Date of publication 11 June 2024; date of current version 17 October 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2021YFB2900100, in part by the Natural Science Foundation of China under Grant 62302259, in part by Guangdong Innovative and Entrepreneurial Research Team Program under Grant 2021ZT09L197, and in part by Meituan. An earlier version of this paper was presented in part at the IEEE International Conference on Computer Communications (IEEE INFOCOM 2022) [DOI: 10.1109/INFOCOM48880.2022.9796933]. *(Fan Dang and Yifan Xu are co-first authors.) (Corresponding author: Yunhao Liu.)*

Fan Dang and Yunhao Liu are with the Global Innovation Exchange, Tsinghua University, Beijing 100084, China (e-mail: dangfan@tsinghua.edu.cn; yunhaoliu@gmail.com).

Yifan Xu is with the School of Software, Tsinghua University, Beijing 100084, China (e-mail: xuyifan20@mails.tsinghua.edu.cn).

Rongwu Xu is with the Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing 100084, China (e-mail: xrw22@mails.tsinghua.edu.cn).

Xinlei Chen is with the Peng Cheng Laboratory and the RISC-V International Open Source Laboratory, Shenzhen International Graduate School, Shenzhen, Guangdong 518000, China (e-mail: chen.xinlei@sz.tsinghua.edu.cn).

Digital Object Identifier 10.1109/TNET.2024.3408147

I. INTRODUCTION

THE proliferation of mobile networks [1], [2] has led to a surge in the popularity of live streaming. Streamers use cameras and microphones to record live events, which are then uploaded to streaming servers and made available on websites for viewers to watch on their computers or mobile devices anytime, anywhere. The COVID-19 pandemic has kept many people at home in recent years, contributing to the growing popularity of live streaming. Live commerce has emerged as a way for businesses to interact with audiences and sell products online, while live education provides teachers and students with a new platform for communication outside the classroom.

With the increasing use of live streaming for various applications, more media content, such as slides, quizzes, and subtitles, is being transmitted to audiences through 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. Synchronization between the video stream and all other content is crucial. Take Rain Classroom [3], a widely used online education platform, for instance. Teachers can send slides and quizzes to students while hosting the live stream. If the page turn of the slides is inconsistent with the video, students may not follow the lesson. Similarly, in HQ Trivia [4], a live trivia video game, an out-of-sync time between the live stream and the quiz would significantly impact the user experience.

Unfortunately, unsynchronization between live streams and other media content is a common problem, caused by the use of different protocols and network channels (*i.e.,* the streaming channel and the information channel) with varying transmission delays. Additionally, the encoding, transcoding, and distribution processes introduce additional delays [5], [6].

Several solutions have been proposed to address the synchronization issue. The simplest solution is to add a fixed time delay to the information channel. However, this approach is difficult to estimate and network fluctuations weaken its effectiveness. Amazon IVS makes use of ID3 [7] timed metadata to enable users to transmit custom data within a live stream at specific time intervals [8]. However, based on our experiments, we observed a synchronization error that exceeded

400 ms, which is noticeable to the audience. Our goal is to achieve a much higher level of synchronization. Alibaba Cloud offers a solution that involves the insertion of Supplemental Enhancement Information (SEI) frames containing information during the H.264 encoding process for video streams [9]. As a result, the information is inherently synchronized with the live stream, leading to minimal synchronization errors on the viewer's end during stream decoding. However, it is worth noting that the SEI interpolation approach becomes less reliable when video frame losses occur. The primary drawback of these commercial solutions is their lack of versatility. For instance, Amazon IVS only supports HLS stream playback for the most widely used live-streaming protocols. On the other hand, Alibaba Cloud lacks support for DASH. For streamers, Alibaba Cloud requires the use of a specific, modified Open Broadcaster Software (OBS) Studio for SEI frame embedding or the development of a custom streamer application based on their provided SDK. Additionally, users do not have the freedom to choose their cloud service providers and may incur additional costs when using these commercial solutions instead of open-source or standard public cloud solutions.

This paper presents LSync, a universal method to achieve timeline-synchronization for live streaming that is compatible with 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 that can be demodulated by the audience to aid synchronization. Periodically embedded timestamps serve as a critical tool for evaluating the latency of live streams. By attaching timestamps to events within the information stream before transmission and once again upon receipt, we can easily measure the delay these events experience. The difference between the latency of the live stream and that of the events within the information stream provides us with a time offset. This offset dictates the waiting period before the events are displayed on the viewer's end, ensuring synchronization with the live stream. To manage the impact of network fluctuations on live stream latency as well as the possible loss of embedded metadata, we incorporate historical latency records. By calculating the exponential moving average of these records, we can effectively handle latency variations, ensuring a more stable and synchronized streaming experience. In contrast to the SEI insertion method, which attaches information directly to the live stream, our approach involves periodically embedding timestamps into the audio stream. This ensures that the information transmission remains on its designated channel, and its transmission reliability is upheld through its specific protocols. To ensure good synchronization performance and user experience quality, the proposed synchronization scheme must meet the following requirements: 1) the inserted modulated audio signals must not interfere with the original audio, 2) the inserted modulated audio signals must not be cut off by the Advanced Audio Coding (AAC) encoder, 3) the inserted modulated audio signals must resist interference and be demodulated under a low signal-to-noise ratio (SNR) on the audience side, and 4) the demodulation and synchronization process on the audience side should be real-time.

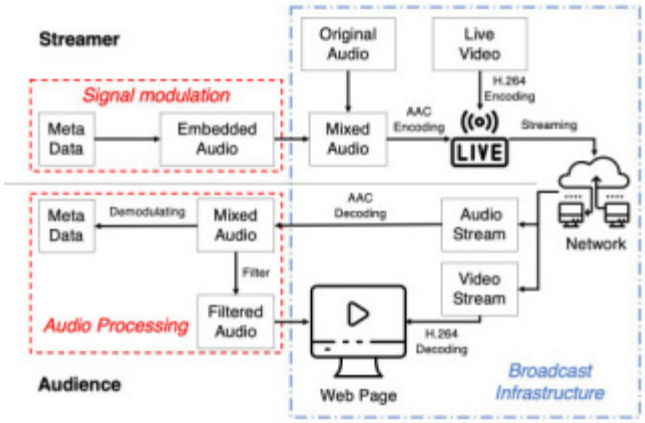


Fig. 1. System architecture overview. On the streamer side, LSync periodically adds modulated audio signals with metadata into the original audio stream. The stream is then encoded, transcoded, and transmitted over the network before being decoded by the web browser on the client side. Once a mixed audio signal is identified, the metadata is extracted from it and the audio stream passes through a filter before playing.

To understand how LSync works, we provide a brief overview of the process at a high level (see Fig. 1). LSync periodically inserts modulated audio signals containing metadata to aid synchronization into the original audio stream on the streamer side. After modulation, the stream is encoded, transcoded, transmitted over the network, and finally decoded by the web browser on the audience side. Before the stream is played, the mixed audio signal is identified, and the embedded metadata is extracted for stream delay estimation and timeline synchronization. Finally, the inserted signal is filtered out from the audio stream to ensure not disturbing audiences.

To meet the abovementioned requirements, we design our synchronization scheme as follows. Firstly, to avoid the inserted modulated data from being cut off by the AAC encoder and prevent it from interweaving with the original audio, we select carrier frequencies between 14 kHz and 15 kHz. This approach marks a substantial departure from previous research methods of hidden acoustic communication, which relied on near-ultrasound frequencies exceeding 17 kHz for data embedding [10], [11], [12], [13], [14]. Our experiments indicate that such high-frequency acoustic signals would be filtered out by a standard AAC encoder commonly used in live broadcast software. Secondly, to resist interference and enable easy demodulation under a low SNR, we use a chirp spread spectrum (CSS)-based method for modulation. CSS allows us to keep the power of embedded signals low, ensuring successful data recovery during signal analysis and easy suppression of disturbing signals. Lastly, to achieve real-time processing on the audience side, we design an efficient algorithm that performs the fast Fourier transform (FFT) just 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 threefold.

- We introduce a lightweight and universal synchronizing framework that requires no modification to traditional

live-streaming infrastructure and can be applied to multiple streaming protocols on both desktop and mobile devices.

- We propose embedding metadata into acoustic CSS signals, which not only bring no interference to audiences but are also easy to detect and demodulate for synchronization.
- We implement our system, LSync, on various kinds of devices and conduct extensive experiments to evaluate its performance. Our results show that the audio signal process is fast and real-time, taking up less than 6% on average of the audio buffer length among all tested devices and the longest processing delay is less than a single buffer. In addition, LSync provides a data rate of $156.25\,\mathrm{bps}$ at best, outperforming recent works. Furthermore, the proposed synchronization mechanism reaches a precision of $24.83\,\mathrm{ms}$ on average, that fits people's viewing habits. In the presence of substantial network congestion, with more than 83% of the available bandwidth allocated to other applications, or in cases of severe signal interference, where the power of the embedded signal falls below -65 dB, LSync may encounter a performance deterioration. Nonetheless, it continues to function effectively unless the bandwidth is entirely saturated or the power of the embedded signal decreases to -80 dB.

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

II. Related Work

The mechanism we propose falls within the realm of hidden channel communication. Many researchers have well studied hidden acoustic or visual channel communications to provide helpful side information to audiences [15], [16]. As for streaming synchronization, several commercial solutions have been proposed as well. In addition, real-time communication protocols for the web like WebRTC emerge, enabling nearly real-time data transmission for users. We discuss them from the following four aspects.

A. Hidden Visual Channel Communication

Inframe++ [17] 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 [18] 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 [19] combines both techniques to simultaneously achieve invisibility and a high capacity, which is $12\times$ that of HiLight. In TextureCode [20], they utilize spatial content-adaptive encoding techniques to achieve both a high goodput of $22\,\mathrm{kbps}$ and minimal flicker. Uber-in-light [21] encodes the data as complementary intensity changes over different color channels for any screen content and significantly improves transmission accuracy.

Although the above approaches provide relatively high throughput, they suffer from a few disadvantages in live streaming. They are 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 [10], [11], [12], [13], [14]. PhoneEar [10] 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 $8\,\mathrm{bps}$. Lee et al. [11] adopt chirp binary orthogonal keying (BOK) to encode data. They choose a $19.5\,\mathrm{kHz}$ to $22\,\mathrm{kHz}$ band for inaudible acoustic communication and achieve a data rate of $16\,\mathrm{bps}$. Ka et al. [12] leverage chirp quaternary orthogonal keying (QOK) for modulation in an $18.5\,\mathrm{kHz}$ to $19.5\,\mathrm{kHz}$ band to deliver information at $15\,\mathrm{bps}$. By leveraging masking effects of the human auditory system, Dolphin [13] 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 [14] inserts hidden sound markers (*i.e.,* binary orthogonal chirps at $18\,\mathrm{kHz}$ to $20\,\mathrm{kHz}$) into audio for data communication and designs 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 III, 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 [13] claims the delay of $600\,\mathrm{ms}$, which is not good enough for real-time signal processing.

C. Commercial Streaming Synchronization Solutions

Several commercial solutions have been put forward to tackle the synchronization challenge between live streams and information streams. Amazon IVS, for instance, leverages ID3 timed metadata [7] to allow users to convey custom data within a live stream at specified time intervals [8]. However, our experiments revealed a synchronization error exceeding $400\,\mathrm{ms}$, a discrepancy noticeable to viewers. On the other hand, Alibaba Cloud presents a solution that incorporates the insertion of SEI frames loaded with information during the H.264 encoding process for video streams [9]. Consequently, the information is intrinsically synchronized with the live stream, resulting in minimal synchronization errors on the viewer's end during stream decoding. Yet, it's important to

highlight that the reliability of the SEI interpolation approach diminishes when video frame losses occur. A significant limitation of these commercial solutions lies in their lack of adaptability. For instance, Amazon IVS solely supports HLS stream playback, while Alibaba Cloud does not support DASH. Furthermore, these solutions restrict users' freedom to select their cloud service providers and may lead to additional costs compared to using open-source or standard public cloud solutions.

Modern television videos and films typically operate at a frame rate between 24 and 30 frames per second (fps) [22], [23]. The human eye can generally tolerate a delay of less than $\frac{1}{24}$s (41.67 ms). As such, an ideal synchronization accuracy between the live stream and the information stream should fall below 41.67 ms. We consider this as one of our primary objectives. Moreover, we aim to devise a method that is more versatile than the previously mentioned commercial solutions. Our goal is to achieve synchronization that is compatible with a wide range of CDN platforms and mainstream streaming technologies.

D. 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 [24]. 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 streams and other media content and makes it a fundamental transmission protocol for lots of video meeting and chat applications including Google Meet [25] and Facebook Messenger [26].

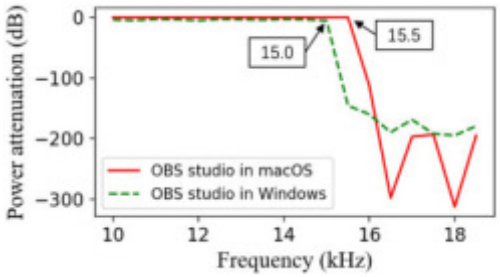
Nevertheless, there are several disadvantages of WebRTC that render it not suitable for our target applications. Basically, WebRTC is a P2P protocol. The limited bandwidth resource restricts the number of the audience side that the streamer side would like to directly communicate with, without sacrificing 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 exposes the user's internal IP address to the web. This problem still surfaces on Mozilla Firefox [27].

III. Channel and Modulation Selection

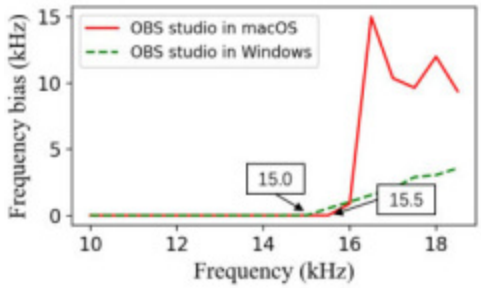
The design of hidden acoustic signals should be inaudible and easy to be demodulated, which is highly dependent on the selection of the acoustic carrier frequencies and the modulation method.

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.



(a) Audio power attenuation



(b) Audio frequency bias

Fig. 2. The effects of OBS Studio's AAC encoder on macOS and Windows in terms of audio power attenuation and frequency bias across various frequencies.

AAC encoder. AAC is the *de facto* audio codec of live streaming [28]. AAC encoder is the most widely used acoustic encoder for live streaming on the internet nowadays due to the incredibly small files it produces. It is also the audio codec for OBS studio, the most widely used open-source live streaming software [29]. The first thing we need to figure out is how the AAC encoder influences our carrier channel selection. In our experiment, we choose the OBS studio as the live-streaming broadcast tool. 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 [30]. 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 kHz to 18.5 kHz. Next, we collect the received audio clips and analyze their power decline. Fig. 2a shows that the remained bandwidths are below 15.5 kHz and 15 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 kHz on macOS

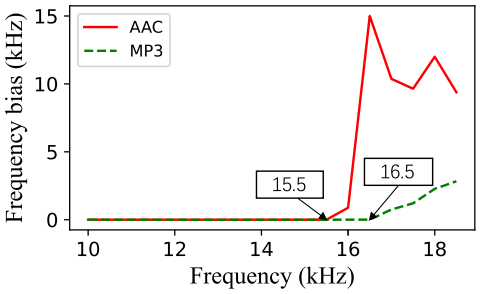
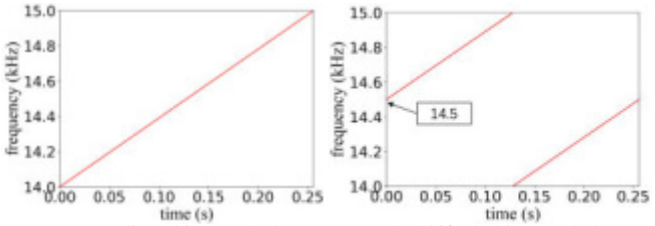


Fig. 3. The audio frequency cutoff phenomenon when using AAC and mp3.



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

Fig. 4. CSS symbol spectrogram.

and 15 kHz on Windows suffer the frequency bias after AAC encoding. It concludes that only the channels with frequency lower than 15 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.

Other codec. The phenomenon of audio frequency cutoff is not exclusive to the AAC encoder. We have conducted similar experiments using the MP3 encoder, another widely utilized lossy audio codec on macOS. As illustrated in Fig. 3, audio clips with frequencies exceeding 15.5 kHz and 16.5 kHz respectively experience frequency bias after AAC and MP3 encoding. The band below 15 kHz selected for AAC is equally applicable for the MP3 encoder. FLAC, a prevalent codec for high-quality audio storage, employs lossless compression, exhibiting no frequency cutoff in our tests [31]. However, its low compression ratio, necessitating increased bandwidth and processing power, along with potential device compatibility and network stability issues, render it unsuitable for live streaming. Taking into account compatibility, we continue using AAC as the audio codec for live streaming in the subsequent sections.

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 [14] shows that the frequencies of the ambient sound in daily life usually lie below 14 kHz.

Based on the above observations, the best choice is the band between 14 kHz and 15 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 kHz and 15 kHz, interference is still possible. Therefore, the modulation method should also resist the potential interference [32], [33].

To cancel the embedded signals, we leverage the Web Audio API [34], 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` [35], 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 kHz, it is infeasible to wipe out the embedded signals between 14 kHz and 15 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 kHz, an audio clip with a frequency of 14 kHz would have $-\log_2\left(\frac{14\text{kHz}}{7\text{kHz}}\right) \times 12\text{dB} = -12\text{dB}$ 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 and easy to be demodulated even if its power level stays low [36], [37].

IV. 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 [11], [12], [14], 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 numbers to further improve the data rate and speed up the demodulation process.

As shown in Fig. 4a, 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. 4). 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 V, 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. 4b. 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. 4a 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. 4b 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 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 VII. 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

$$\begin{aligned}&\sin\left(2\pi(fc+f+\frac{BW}{2T}t)\cdot t\right)\sin\left(2\pi(BW-\frac{BW}{2T}t)\cdot t\right)\\&=\frac{1}{2}\Bigg[\cos\left(2\pi(fc+f+BW)\cdot t\right)\\&\quad-\cos\left(2\pi(fc+f-BW+\frac{BW}{T}t)\cdot t\right)\Bigg],\end{aligned}$$

where the first part of the result is centralized on a specific frequency $fc+f+BW$ while the second part is 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.

B. Frame Design

As shown in Fig. 5, 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.

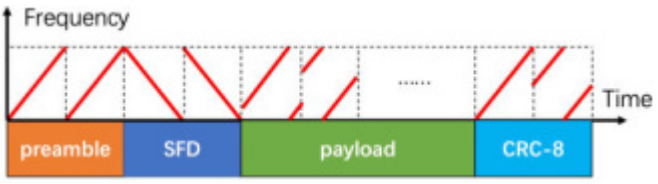


Fig. 5. The structure of the embedded packet. It includes a preamble and SFD for packet detection, a payload that contains metadata, and a CRC for integrity checking.

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 detection reliability and efficiency. Using more than two 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 the 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 VII, 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 [38]. 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.

V. 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 [11], [12], [14] 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.

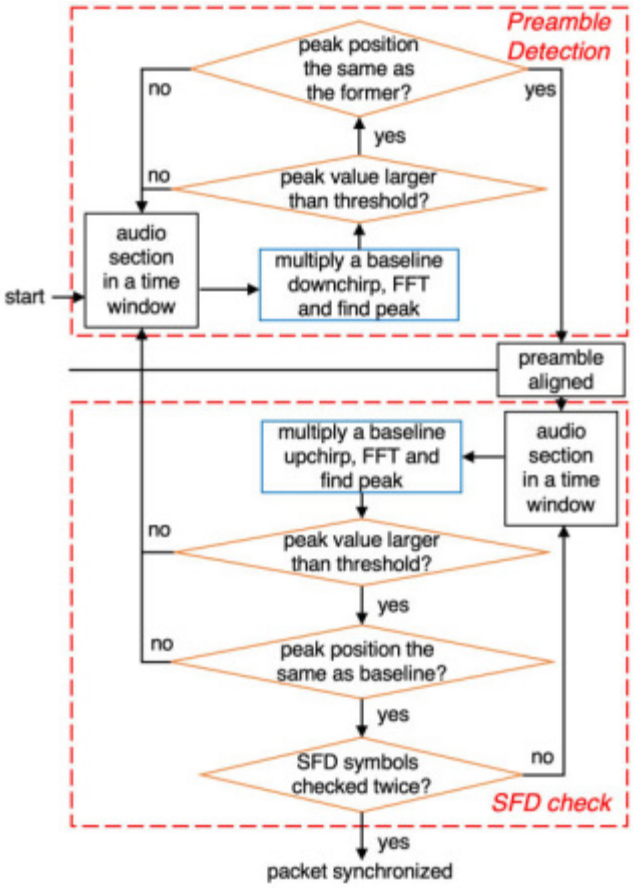


Fig. 6. Packet synchronization process.

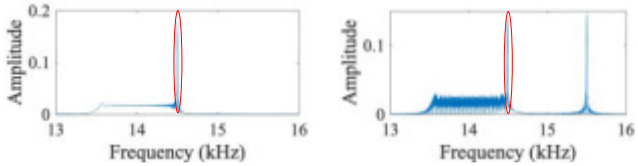
A. Packet Detection and Synchronization

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. 6 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 the preamble. Even if the preamble is misaligned with the time windows like the left part of Fig. 7, the FFT results have the same peak position as shown in Fig. 8(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

Fig. 7. Align chirps of the preamble with time window.



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

Fig. 8. FFT result of two consecutive time windows for the preamble, where two peaks share the same position.

aligned with the time window. In the left part of Fig. 7, 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

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

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. 8b, and we select the greater peak position of the original FFT result as id', which represents 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, m can be computed as

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

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 $fs = 48\,\text{kHz}$, $BW = 1\,\text{kHz}$ and fft_n should be an integer that is a power of 2 and no less than $chirp\_n$. Supposing that T is a power of $2\,\text{ms}$ and denoted as $2^l \times 10^{-3}s$, $chirp\_n$ equals $2^{l+4} \times 3$. Thus we set fft_n to be 2^{l+6} and m can be recomputed as: $m = (id' - id_0) \times 36$, which is definitely a non-negative integer. Otherwise, if T is not a power of $2\,\text{ms}$ and is like $100\,\text{ms}$ and $chirp\_n$ equal 4800, fft_n should be at least $8192 = 2^{13}$. Then m 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 m 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

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

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.

VI. Timeline-Synchronization Mechanism

In this section, we will elaborate on the detailed synchronization mechanism, which solves the asynchrony problem between the streamer and the audience side, as well as copes with network fluctuations.

A. Synchronization With Little Network Fluctuation

The information stream delay is typically much shorter than that of the live stream. To ensure synchronization, the transmitted message in the information channel has to wait for a short interval $interval_{i\&l}$ before it appears on the audience side, which matches the live stream. We use $latency_i$ and $latency_l$ to represent the latencies of the information stream and the corresponding live stream, respectively. The $interval_{i\&l}$ is expressed as:

$$interval_{i\&l} = latency_l - latency_i,$$

The latency of a live stream is a multifaceted issue, encompassing aspects such as stream encoding, transcoding, transmission, and decoding. The encoding, transcoding, and decoding processes are handled locally on PCs, and the delay they introduce is largely determined by the PCs' capabilities, which tend not to vary significantly. In contrast, the transmission delay can experience significant variations due to network fluctuations. If we assume minimal network fluctuations, we can simplify the problem by treating the latency of the live stream as a nearly constant value. On the streamer side, the local timestamp t_l^s is recorded and encoded into the embedded packet, which is transmitted within the audio stream through the network and demodulated on the audience side. During the demodulation process, the packet's start point is timestamped as t_l^a with the audience's local clock. Then, the live stream transmission latency can be computed as:

$$latency_l = t_l^a - t_l^s - \theta_1,$$

where θ_1 represents the time drift between the streamer's and the audience's clocks at the point of t_l^a. As for the information stream, an emergency event message can be timestamped as t_i^s on the streamer side, and both the message and its timestamp are sent to the audience side. Once received, the message is timestamped as t_i^a on the audience side, and its delay can be expressed as

$$latency_i = t_i^a - t_i^s - \theta_2,$$

where θ_2 represents the time drift between the streamer's and audience's clock at the point of t_i^a.

Considering the time drift between the streamer's and the audience's clocks remains stable, that is $\theta_1 \approx \theta_2$, we have:

$$interval_{i\&l} = t_l^a - t_l^s - t_i^a + t_i^s.$$

The asynchrony problem is mitigated through the calculation.

B. Synchronization With Network Fluctuation

However, in reality, the latency of the live stream can change over time due to network fluctuations. To address this issue, we program the streamer to periodically generate embedded audio packets containing timestamps, such as every 5 s. Upon receiving a new packet, the audience side updates the estimated latency using the exponential moving average method. Given that the last estimated value is $latency_{l_{j-1}}$, we express the updated latency as:

$$interval_{l_j} = \alpha(t_{l_j}^a - t_{l_j}^s) + (1 - \alpha) latency_{l_{j-1}}, \quad (2)$$

where $0 < \alpha \leq 1$ and $j \geq 2$.

When an emergency message with latency $t_i^a - t_i^s$ is received, the audience side schedules it to show up after $interval_{i\&l}$, which is computed as:

$$interval_{i\&l} = latency_{l_j} - t_i^a + t_i^s,$$

where $latency_{l_j}$ is the most recently updated stream latency.

The embedded packet generation interval and the parameter α in Equation 2 can be adjusted according to the application requirements and network situation.

VII. 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) [39]. OBS Studio allows multiple audio inputs. Therefore, we introduce a virtual audio cable with VB-Cable [40] 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. Generally, the signals are generated every 5 s and then mixed with the live audio and transmitted to the live streaming server together. We set the sample rate to 48 kHz and the bit rate of audio to 96 kHz, which are general settings for a daily live broadcast. We also turn down the power of the virtual cable to -62.5 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 to access the audio from an HTML `<video>` element for later processing. Then we leverage `ScriptProcessorNode` interface 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 V. 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 [41], 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` [42], we launch an RTMP server 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 [43], HLS [44], and DASH [45].

The rest of this section presents the experiment results. The performance metrics we prioritize include the packet reception ratio (PRR), the time offset between the information and the live stream, and the data rate. The successful reception of embedded metadata packets is crucial for subsequent synchronization processes, and as such, we consider it the primary criterion. Typically, modern television videos and films operate at frame rates between 24 and 30 fps, ensuring that the videos appear realistic and comfortable to the viewer, as referenced in [22] and [23] and the frame rate study. The human eye can tolerate a delay of less than $\frac{1}{24}$s (41.67 ms). If we can achieve a time offset below 41.67 ms after synchronization, it would align well with human viewing habits. However, without synchronization, the temporal offset in standard live streaming services can be as large as a few seconds. Our experiments, as outlined in Section VII-F, revealed a synchronization error exceeding 400 milliseconds when using Amazon IVS's synchronization solution. We aim to surpass this performance. The devices that we employ include a PC with an Intel Core i7-10710U CPU running Windows 10, a MacBook Pro (16-inch, 2019) 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 dB so that the embedded signal after processing can be totally unaware of by audiences even in such a silent environment. Considering the consistency of the experiment, the default settings for the article are as follows: we run the streamer side on MacBook and the audience side on Google Chrome on PC. We also discuss the versatility of our solution in the following Section VII-C.

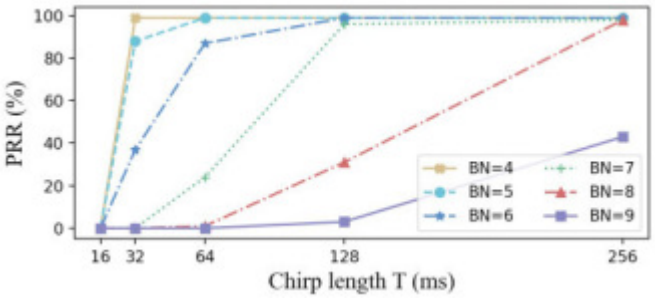


Fig. 9. The probability of successful packet reception (PRR) under different BN and T settings. As T increases and BN decreases, PRR also increases.

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. 9 shows the relationship among 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 the 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 IV, we have shown that the data rate is proportionate to BN and inversely proportional to T, and we

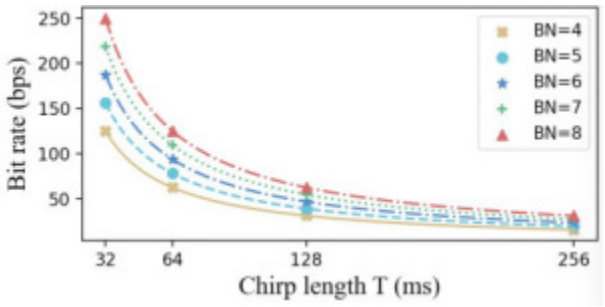


Fig. 10. Data rate under different BN and T settings. As T increases and BN decreases, data rate also decreases.

TABLE I

DATA RATE UNDER DIFFERENT BN AND T SETTINGS WITH A PRR OVER 95%. GIVEN THIS PRR REQUIREMENT, LSYNC IS CAPABLE OF PROVIDING A DATA RATE OF 125 BPS

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%. GIVEN THIS PRR REQUIREMENT, LSYNC IS CAPABLE OF PROVIDING A DATA RATE OF 156.25 BPS

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%

depict the relationship in Fig 10. Based on Fig 9 and Fig 10, we summarize the greatest data rate that our system can provide when PRR is greater than 95% in Table I. We list the result 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 [14] 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. 11) 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 IV, 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.

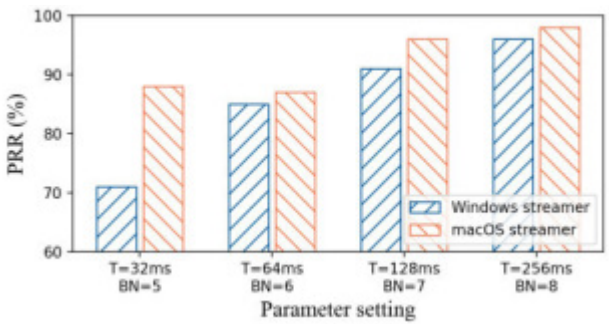


Fig. 11. Packet reception ratio with different streamer environments. The streamer side implementation on macOS offers a slightly better PRR performance compared to Windows.

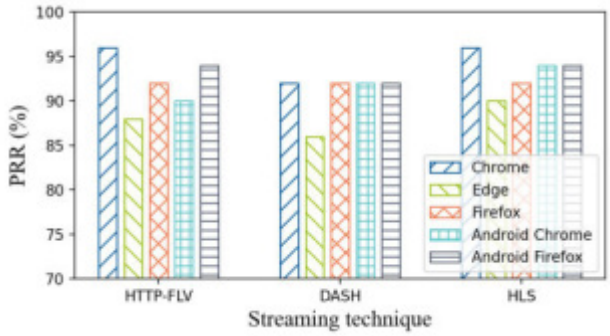


Fig. 12. Packet reception ratio with different streaming protocols and browsers on the audience side, when $T = 128ms,\ BN = 7$. In terms of streaming protocols, the performances are similar, with a slightly lower probability of PRR observed when using DASH. With regards to browser settings, LSync performs optimally on Google Chrome but has the poorest performance on Edge.

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 different mainstream browsers and the above three protocols. Fig. 12 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 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 being different.

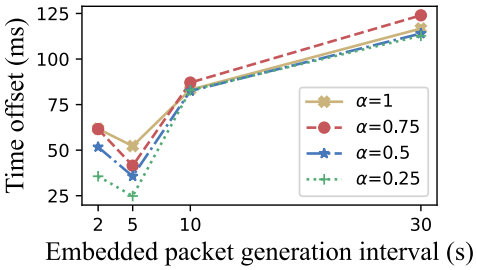


Fig. 13. The average time offset between information and live stream after synchronization. Setting the update interval to 5 s results in the shortest time offset between the information and the live stream. Additionally, the offset decreased as we reduced the value of α, with a possible minimum of 24.83 ms.

LSync works best on Google Chrome, where the PRRs are 96%, 92%, and 96% when tested with HTTP-FLV, DASH, and HLS. Nevertheless, the PRRs slightly drop to 88%, 86%, and 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 iPhones, the results show that this method does work on ordinary mobile devices, and it should work on iPhones in the future.

D. Synchronization Precision

We evaluate the precision of the timeline-synchronization mechanism proposed in Section VI, using the time offset between information and live stream as the measure of accuracy. To test our mechanism, we vary the embedded signal generation interval on the streamer side to 2 s, 5 s, 10 s, and 30 s, while setting the modulation parameters $T = 128ms$ and $BN = 7$. We acquire the local timestamp just before sending an embedded packet and sent it to the audience side through the WebSocket [46]. The sending of the WebSocket and the packet in the live stream occurr simultaneously.

On the audience side, we demodulate the audio signal and periodically update the live stream latency using Eq. 2, while scheduling the received WebSocket information to show up after $interval_{i\&l}$. We vary the parameter α from 1 to 0.25. We calculate the time offset between the WebSocket information shown-up time point and the start point of the corresponding packet in the live stream.

As shown in Fig. 13, when the update interval is set to 5 s, the delay between the information and the live stream is the shortest among the four interval settings. If the interval is too great, the last renewed latency may not reflect the current network situation, limiting the synchronization precision. Conversely, if the interval is too small, the frequent latency updating process may not accurately reflect the more general network delay, thus hampering synchronization accuracy. A packet generation interval of 5 s produces the lowest time offset, which decreases as we reduce the value of α. By considering historical measurements, we mitigate the effect of abrupt changes in the live stream latency.

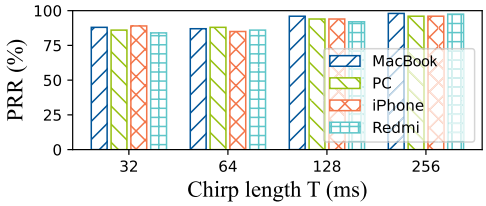


Fig. 14. Packet reception ratio on Google Chrome on various types of devices.

With the configuration of $\alpha = 0.25$ and updating interval equaling 5 s, the average time offset between the information and the live stream was as low as 24.83 ms, less than 41.67 ms. Therefore, our synchronization mechanism is well-suited to people's viewing habits.

E. Robustness

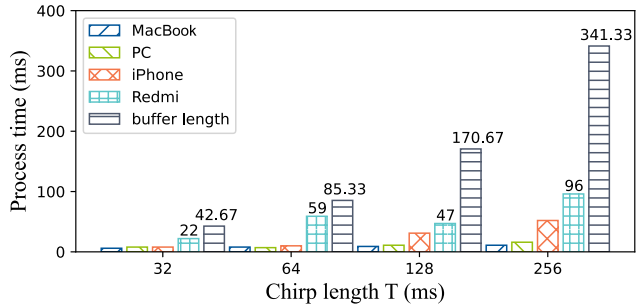
In order to comprehensively evaluate our proposed LSync system, we consider critical factors that include variation in the capability of devices, signal interference, and network congestion, exploring their substantial influence on reception reliability and overall system performance.

Performance across devices. To evaluate the system's performance considering the variations in device capabilities, we measure the PRR and processing delay when we implement the audience side on Google Chrome on various devices including the PC, MacBook, Redmi, and iPhone. The experiments are carried out under the configuration shown in Table II. Although the audio stream cannot be processed due to the bug [41] on iPhone, we record a clip with the same quality as the live stream and utilize the browser on iPhone to process it in the same way.

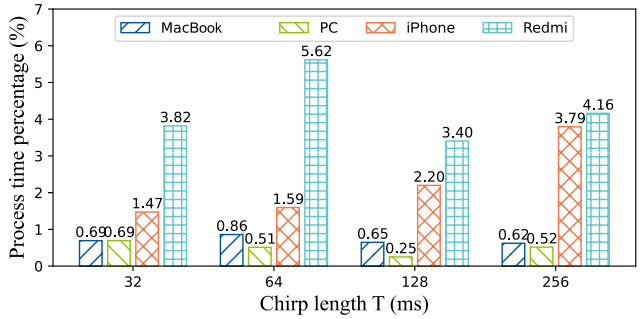
Fig. 14 shows the PRR on different kinds of devices. Even though the capability of each device varies, the reception accuracy of the embedded signal exhibits little difference.

Previous work in [11], [12], [14], and [13] 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.

Fig. 15a 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. Even on the least performing mobile device, the Redmi, 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 every single buffer. Therefore, during a period when the buffer is receiving the next audio signal section, the last received section can be entirely processed. To further illustrate our processing efficiency, we calculate the mean processing time for different settings. Fig. 15b 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



(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. 15. Processing time for each audio buffer in diverse devices under different settings. The longest processing time for each buffer is shorter than the buffer length, thereby ensuring real-time audio processing. In addition, the average processing time for each buffer is less than 6% of the buffer length.

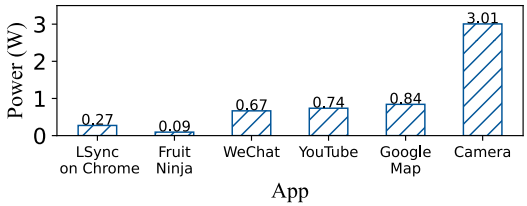
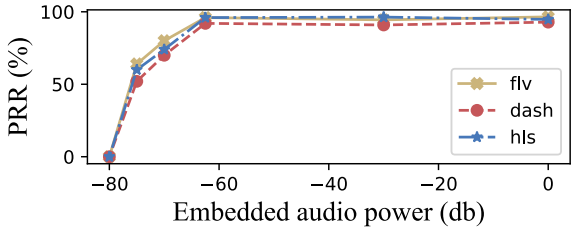


Fig. 16. Power consumption of LSync and other applications on Redmi mobile phone.

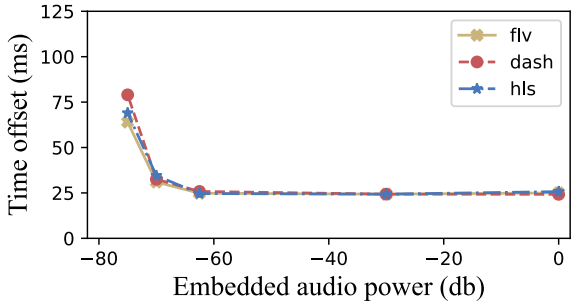
on mobile phones, the processing takes up a bit more time but less than 6% on average for each buffer. According to our experimental results, not only real-time signal processing is achieved, but also it is highly efficient even on mobile devices with poor computational performance.

In addition, we conducted a power consumption comparison of the LSync audience side with various other applications on the Redmi mobile phone, which included Fruit Ninja (a lightweight game), the background-running WeChat, YouTube video playback, Google Maps, and the Camera app for taking photos. Each application runs for 300 seconds, after which we gathered energy consumption data and calculated its power consumption. As illustrated in Fig. 16, when running on Chrome, LSync exhibited significantly lower power consumption compared to the other applications, except for the Fruit Ninja. This result highlights the efficiency of LSync's signal-processing design, indicating that LSync will not cause a significant drain on the mobile device's battery.

Signal interference. To validate our assertion made in Section III-B, where we mentioned selecting the CSS technique for data modulation due to its ability to be demodulated



(a) Embedded packet reception ratio



(b) Average time offset between information and live stream after synchronization

Fig. 17. LSync system performance including embedded packet reception reliability and the average time offset between information and live stream after synchronization with different interference strengths.

at a low power level and its resilience to interference, we conducted experiments at various interference strength levels. We measured both packet reception reliability and synchronization performance. As for the experiment configuration, the PC and MacBook are connected to the same WLAN. The streamer side runs on a MacBook, and the audience side runs on Google Chrome on PC. To represent various levels of interference strength, we kept the power of the standard audio cable at 0 dB while adjusting the power of the virtual audio cable with embedded packet insertion from 0 to -80 dB. The experiments are carried out with the configuration of $T = 128ms$, $BN = 7$ and the embedded packet generation interval $= 5\,\mathrm{s}$, $\alpha = 0.25$.

The experimental results are presented in Fig. 17. These results show a similar pattern across three different stream protocols. According to the experiments, both PRR and synchronization precision remain stable when the embedded audio power exceeds -62.5 dB, which is approximately 10^{-6} of the standard audio power. However, when the embedded audio power weakens to -65 dB and -70 dB, the PRR drops to approximately 75% and 60%, respectively. The increased packet loss leads to actual time intervals between successfully received packets exceeding the 5-second sending interval, resulting in a corresponding decline in synchronization precision. However, even when the embedded audio power decreases to -75 dB, the synchronization error remains within $80\,\mathrm{ms}$, and our system continues to function. At an embedded audio power level as low as -80 dB, no packets can be successfully recognized and demodulated. Under this condition, our system cannot operate properly.

The experiments above confirm the robustness of the CSS modulation technique against interference. LSync performs flawlessly when the power of the embedded signal exceeds 10^{-6} of the standard audio. Even when the power diminishes

to as low as −75 dB, LSync continues to operate, albeit with a slight reduction in synchronization precision.

Network congestion. In the context of experimental validation of LSync technology, network congestion stands out as a critical factor to consider. By deliberately introducing and quantifying network congestion in controlled experiments, we can assess the robustness and performance of LSync.

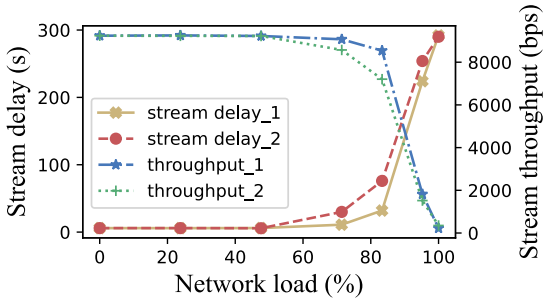
The experimental setup remains consistent with the configuration described in the preceding section on signal interference, with the power of the virtual audio cable equaling −62.5 dB. To evaluate the impact of network congestion on the stream pushing and pulling stages respectively, we carry out two sets of experiments as follows:

1) The live streaming server is implemented on PC. The audience side pulls the stream locally and network congestion only influences the stream-pushing process.
2) The live streaming server is implemented on MacBook and thus the streaming-pulling process is affected by network fluctuations.

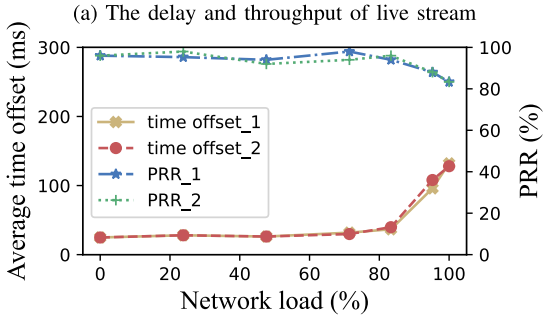
We employ the iPerf3 tool [47] to generate UDP background traffic at varying data rates from the MacBook to the PC, in order to quantify network congestion.

To start with, we measure the UDP bandwidth available between the two devices using iPerf3. The average bandwidth is 419 Mbps. Due to network fluctuations, the instantaneous available bandwidth ranges from 122 Mbps to 549 Mbps. Following this, we systematically vary the UDP background traffic bandwidth from 0 to 420 Mbps. Subsequently, we monitor the delay and throughput of the live stream and evaluate the performance of the LSync system.

Fig. 18 presents the experimental results. Line numbers 1 and 2 correspond to the aforementioned experimental configuration respectively. The network load represents the ratio of the bandwidth used by the current UDP background traffic to the total average 419 Mbps bandwidth. The higher it is, the more severe the network congestion will be. The figure shows that in both scenarios, the UDP background traffic has a similar influence on the system performance. As depicted in Fig. 18a, when the network load surpasses 83%, the average throughput of the live stream experiences a sharp decline, accompanied by an increase in delay. The LSync system performs accordingly as shown in Fig. 18a. The embedded packet reception reliability and the synchronization precision get lower as the UDP traffic increases. When the network load is already substantial, the live stream experiences stuttering. Subsequent streams have to wait until there is sufficient available bandwidth to continue transmission and playback. If the stream freezing occurs exactly at the location of embedded audio packets, it leads to data packet corruption, resulting in reduced packet reception rates. However, due to the short length of each packet, there is a higher probability that it can remain intact within the continuous transmission of the live stream, and the PRR exceeds 80%. On the other hand, even though the embedded packets are generated every 5 seconds, the actual time intervals between received packets become significantly longer due to the stream stuttering. Consequently, the average time offset between information and the live



(a) The delay and throughput of live stream



(b) The LSync system performance

Fig. 18. The delay in 5 minutes and throughput of the live stream as well as the LSync system performance including embedded packet reception reliability and the average time offset between information and live stream after synchronization with increasing UDP background traffic.

stream after synchronization increases, as demonstrated in the previous Section VII-D.

In summary, under conditions where all other traffic utilizes less than 83% of the average total available bandwidth, the live stream can be transmitted smoothly, and our LSync system demonstrates excellent performance. However, in cases where the live stream experiences stuttering due to increased network congestion, the LSync system experiences a slight reduction in packet reception reliability and synchronization precision.

F. Comparative Analysis

To comprehensively assess the strengths and weaknesses of LSync in comparison to other available solutions, we conducted a comprehensive comparative analysis involving existing solutions, such as Amazon IVS and Alibaba Cloud live stream service, as well as insights from prior research papers. This approach enhances the depth of our evaluation, providing a more comprehensive understanding of LSync's effectiveness and potential limitations.

Amazon IVS employs timed metadata to enable users to integrate synchronized interactive features into video applications. It utilizes ID3 timed metadata to transmit customized data within a live stream at specific time points. The streaming process is visualized in Fig. 19. In our experimental setup, we utilized the standard OBS Studio for streaming, and the AWS CLI for timed metadata insertion. On the audience side, an event is triggered whenever playback reaches a segment with embedded metadata, using the Amazon IVS Player SDK [8]. When implementing the Alibaba Cloud solution, it requires a customized version of OBS Studio to support SEI frame insertion functionality as depicted in Fig. 20. The SEI frame containing information is inherently synchronized

TABLE III
COMPARISON AMONG LSYNC AND COTS STREAM SYNCHRONIZING SOLUTIONS

Solution	Amozon IVS	Alibaba Cloud	LSync
Synchronizing solution	ID3 tag in audio	SEI frame in video	Embedded audio
Synchronization precision	Low (452 ms)	High (<10 ms)	**High** (24.83 ms)
Robustness	Low	Low	**Medium**
Protocol compatibility with FLV, HLS, and DASH	Low	Medium	**High**
Streamer Independence	High	Low	**High**
Server Independence	Low	Low	**High**

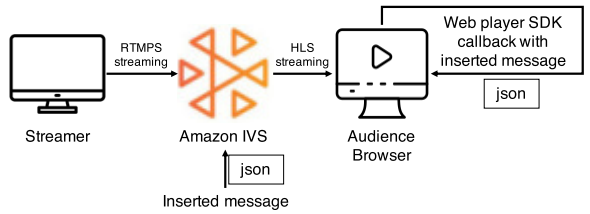


Fig. 19. Information synchronization with Amazon IVS based on ID3 tags.

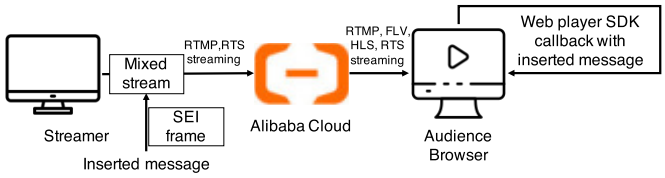


Fig. 20. Information synchronization with Alibaba Cloud based on SEI frame insertion.

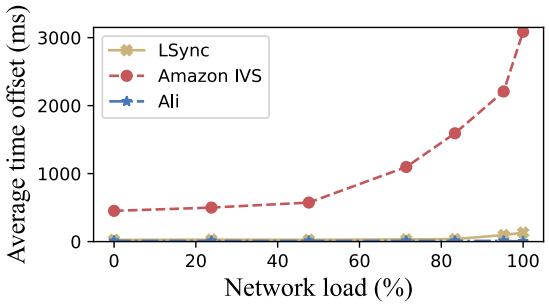


Fig. 21. The comparison of average time offset between information and live stream after synchronization for different solutions.

with the live stream. We assessed the average information synchronization precision for both COTS solutions and compared them with our proposed LSync solution.

The comparative assessment results are summarized in Table III. Besides, The comparison of average time offset between information and live stream after synchronization for the three solutions under different network congestion condition is shown in Fig. 21. Among the evaluated solutions, Alibaba's approach, utilizing SEI frame insertion, demonstrates the highest synchronization precision due to its intrinsic information-stream synchronization. In contrast, Amazon's solution, which relies on ID3 tags, exhibits the lowest synchronization precision when aligning transmitted metadata with the stream. Moreover, its performance drastically decreases as network congestion intensifies. The timed metadata uses ID3 tags embedded in the audio segments for synchronization. The mismatch between the data and the segment as well as the length of each segment can cause a relatively large misalignment. LSync shows significantly improved synchronization precision compared to Amazon IVS. However, there is still room for enhancement to catch up with the SEI frame insertion approach.

In terms of robustness, the SEI interpolation approach may become less reliable when there are potential video frame losses. Meanwhile, since ID3 tags are typically stored within the audio data portion, the transmission is susceptible to audio segment loss or corruption, dependent on network connection stability and audio encoding quality. Based on our experiments, we've observed that the information packet loss rate can skyrocket to a staggering 60% during periods of severe network congestion. Despite the potential for packet loss or network interruptions that affect the transmission of embedded audio packets in LSync, the system maintains functionality with a slight reduction in synchronization precision, albeit with decreased PRR and longer time intervals between received embedded packets.

From a system versatility perspective, LSync outperforms the other solutions. While LSync works seamlessly with FLV, DASH, and HLS, Amazon IVS only supports HLS stream playback, and Alibaba Cloud lacks support for DASH. On the streamer side, both Amazon IVS and LSync offer versatility in streamer application choices, whereas Alibaba Cloud requires a specific, modified OBS Studio for SEI frame embedding or the development of a custom streamer application based on their provided SDK. Regarding the server, users have the flexibility to choose cloud service providers with LSync, whereas Amazon and Alibaba provide no such freedom and their live streaming services entail additional costs.

From a technical standpoint, LSync can be classified as a form of hidden acoustic channel communication technique. To evaluate its capabilities, we conducted a comparative analysis with previous research papers.

The summarized results are presented in Table IV. LSync distinguishes itself by being the pioneer in implementing the CSS modulation technique for concealed acoustic channel communication. Notably, it stands alone in refraining from utilizing the near-ultrasound bands with frequencies exceeding 17 kHz for data embedding. This is essential since such near-ultrasound bands would be filtered out by an AAC encoder, which is a standard component of live broadcast software. Consequently, other research methods cannot be directly applied to livestream services. Moreover, LSync exhibits an exceptional channel utilization of 156.25 bps/kHz, surpassing Tagscreen by almost sixfold and Dolphin by fourfold. Nevertheless, there is potential for further enhancement in LSync's performance. This includes the possibility of expanding the bandwidth of the embedded audio into lower frequency ranges and applying more powerful filtering processes to enhance its quality. These improvements can lead to increased throughput, making LSync more competitive for concurrent live streaming and data transmission.

TABLE IV
COMPARISON WITH HIDDEN ACOUSTIC CHANNEL COMMUNICATION RESEARCH PAPERS

Solution	Hyewon [11]	Ka [12]	Dolphin [13]	Tagscreen [14]	LSync
Modulation	BOK	QOK	OFDM	BOK	**CSS**
Channel (kHz)	19.5-22	18.5-19.5	8-20	18-20	**14-15**
Channel Utilization (bps/kHz)	6.4	15	41	25	**156.25**

VIII. CONCLUSION

In this work, we present LSync, for the synchronization of traditional live streams and other media content. A key innovation is to insert a 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 $1\,\mathrm{kHz}$. The proposed synchronization mechanism reaches a precision of $24.83\,\mathrm{ms}$ on average, that fits people's viewing habits. 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.

REFERENCES

[1] Z. Yin, C. Wu, Z. Yang, and Y. Liu, "Peer-to-peer indoor navigation using smartphones," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 5, pp. 1141–1153, May 2017.

[2] W. Gu, Z. Yang, L. Shangguan, W. Sun, K. Jin, and Y. Liu, "Intelligent sleep stage mining service with smartphones," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, Sep. 2014, pp. 649–660.

[3] *Rain Classroom*. Accessed: Nov. 1, 2023. [Online]. Available: https://www.yuketang.cn/

[4] *HQ Trivia*. Accessed: Nov. 1, 2023. [Online]. Available: https://en.wikipedia.org/wiki/HQ_(video_game)

[5] *Live Streaming Process*. Accessed: Nov. 1, 2023. [Online]. Available: https://www.dacast.com/blog/what-is-live-streaming/

[6] W. Jiang, S. S. Ge, and D. Li, "Fixed-time-synchronized control: A system-dimension-categorized approach," *Sci. China Inf. Sci.*, vol. 66, no. 7, Jul. 2023, Art. no. 172203.

[7] *ID3*. Accessed: Nov. 1, 2023. [Online]. Available: https://en.wikipedia.org/wiki/ID3

[8] *Amazon IVS User Guide*. Accessed: Nov. 1, 2023. [Online]. Available: https://docs.aws.amazon.com/ivs/latest/LowLatencyUserGuide/metadata.html

[9] *Solution of Alibaba Cloud for Live Quiz*. Accessed: Nov. 1, 2023. [Online]. Available: https://developer.aliyun.com/article/394552

[10] 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 *Proc. 7th ACM SIGCHI Symp. Eng. Interact. Comput. Syst.*, Jun. 2015, pp. 174–179.

[11] H. Lee, T. H. Kim, J. W. Choi, and S. Choi, "Chirp signal-based aerial acoustic communication for smart devices," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 2407–2415.

[12] S. Ka et al., "Near-ultrasound communication for TV's 2nd screen services," in *Proc. 22nd Annu. Int. Conf. Mobile Comput. Netw.*, Oct. 2016, pp. 42–54.

[13] 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 *Proc. 22nd Annu. Int. Conf. Mobile Comput. Netw.*, Oct. 2016, pp. 29–41.

[14] Q. Lin, L. Yang, and Y. Liu, "TagScreen: Synchronizing social televisions through hidden sound markers," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2017, pp. 1–9.

[15] X. Chen, X. Wu, X.-Y. Li, X. Ji, Y. He, and Y. Liu, "Privacy-aware high-quality map generation with participatory sensing," *IEEE Trans. Mobile Comput.*, vol. 15, no. 3, pp. 719–732, Mar. 2016.

[16] Z. Yang, L. Jian, C. Wu, and Y. Liu, "Beyond triangle inequality: Sifting noisy and outlier distance measurements for localization," *ACM Trans. Sensor Netw.*, vol. 9, no. 2, pp. 1–20, 2013.

[17] 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 *Proc. 13th Annu. Int. Conf. Mobile Syst., Appl., Services*, May 2015, pp. 181–195.

[18] T. Li, C. An, X. Xiao, A. T. Campbell, and X. Zhou, "Real-time screen-camera communication behind any scene," in *Proc. 13th Annu. Int. Conf. Mobile Syst., Appl., Services*, May 2015, pp. 197–211.

[19] S. Shi, L. Chen, W. Hu, and M. Gruteser, "Reading between lines: High-rate, non-intrusive visual codes within regular videos via ImplicitCode," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, Sep. 2015, pp. 157–168.

[20] V. Nguyen et al., "High-rate flicker-free screen-camera communication with spatially adaptive embedding," in *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.

[21] M. Izz, Z. Li, H. Liu, Y. Chen, and F. Li, "Uber-in-light: Unobtrusive visible light communication leveraging complementary color channel," in *Proc. IEEE INFOCOM 35th Annu. IEEE Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.

[22] A. Mitchell and M. Mitchell, *Visual Effects for Film and Television*. New York, NY, USA: Taylor & Francis, 2004.

[23] *Frame Rate: A Beginner's Guide*. Accessed: Nov. 1, 2023. [Online]. Available: https://www.techsmith.com/blog/frame-rate-beginners-guide/

[24] *WebRTC*. Accessed: Nov. 1, 2023. [Online]. Available: https://webrtc.org/

[25] *Google Meet*. Accessed: Nov. 1, 2023. [Online]. Available: https://meet.google.com/

[26] *Facebook Messenger*. Accessed: Nov. 1, 2023. [Online]. Available: https://www.messenger.com/

[27] *Prevent WebRTC From Leaking Local IP Address*. Accessed: Nov. 1, 2023. [Online]. Available: https://github.com/gorhill/uBlock/wiki/Prevent-WebRTC-from-leaking-local-IP-address

[28] M. Bosi, "ISO/IEC MPEG-2 advanced audio coding," *J. Audio Eng. Soc.*, vol. 45, no. 10, pp. 789–814, 1997.

[29] *OBS Studio*. Accessed: Nov. 1, 2023. [Online]. Available: https://obsproject.com/

[30] *SBR White Paper*. Accessed: Nov. 1, 2023. [Online]. Available: http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/drm/SBR_White_Paper_v1.pdf

[31] *What is FLAC*. Accessed: May 13, 2024. [Online]. Available: https://xiph.org/flac/

[32] C. Li, H. Chen, Z. Wang, Y. Sun, X. Li, and T. Qin, "Two-stage constructions for the rate-compatible shortened polar codes," *Tsinghua Sci. Technol.*, vol. 28, no. 2, pp. 269–282, Apr. 2023.

[33] F. Dang, P. Zhou, Z. Li, and Y. Liu, "NFC-enabled attack on cyber physical systems: A practical case study," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, May 2017, pp. 289–294.

[34] *Web Audio API*. Accessed: Nov. 1, 2023. [Online]. Available: https://developer.mozilla.org/en-U.S./docs/Web/API/Web_Audio_API

[35] *Web Audio BiquadFilterNode*. Accessed: Nov. 1, 2023. [Online]. Available: https://developer.mozilla.org/zh-CN/docs/Web/API/BiquadFilterNode

[36] Y. Lin, W. Dong, Y. Gao, and T. Gu, "SateLoc: A virtual fingerprinting approach to outdoor Lora localization using satellite images," *ACM Trans. Sensor Netw.*, vol. 17, no. 4, pp. 1–28, Nov. 2021.

[37] D. Lin, Q. Wang, W. Min, J. Xu, and Z. Zhang, "A survey on energy-efficient strategies in static wireless sensor networks," *ACM Trans. Sensor Netw.*, vol. 17, no. 1, pp. 1–48, Feb. 2021.

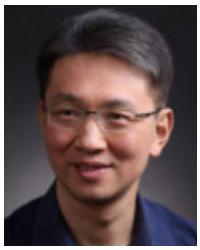
[38] *Cyclic Redundancy Check*. Accessed: Nov. 1, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Cyclic_redundancy_check
[39] *Real-time Messaging Protocol (RTMP) Specification*. Accessed: Nov. 1, 2023. [Online]. Available: https://helpx.adobe.com/adobe-media-server/dev/stream-live-media-rtmp.html
[40] *VB-Cable Virtual Audio Device*. Accessed: Nov. 1, 2023. [Online]. Available: https://vb-audio.com/Cable/
[41] *Safari CreateMediaElementSource Broken on HLS/m3u8 Video Sources*. Accessed: Nov. 1, 2023. [Online]. Available: https://stackoverflow.com/questions/60889426/is-safari-createmediaelementsource-broken-on-hls-m3u8-video-sources
[42] *Node Media Server*. Accessed: Nov. 1, 2023. [Online]. Available: https://github.com/illuspas/Node-Media-Server
[43] *Flash Video*. Accessed: Nov. 1, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Flash_Video
[44] *HTTP Live Streaming*. Accessed: Nov. 1, 2023. [Online]. Available: https://developer.apple.com/streaming/
[45] *Dynamic Adaptive Streaming Over HTTP*. Accessed: Nov. 1, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Dynamic_Adaptive_Streaming_over_HTTP
[46] *Websockets*. Accessed: Nov. 1, 2023. [Online]. Available: https://websockets.readthedocs.io/en/stable/
[47] *IPerf-The Ultimate Speed Test Tool for TCP, UDP and SCTP*. Accessed: Nov. 1, 2023. [Online]. Available: https://iperf.fr/

Fan Dang (Member, IEEE) received the B.E. and Ph.D. degrees in software engineering from Tsinghua University, Beijing, in 2013 and 2018, respectively. He is a Research Assistant Professor with the Global Innovation Exchange, Tsinghua University. His research interests include industrial intelligence, edge computing, and mobile security. He is a member of ACM.

Yifan Xu (Student Member, IEEE) received the B.S. degree in software engineering from Tsinghua University, Beijing, in 2020, where he is currently pursuing the Ph.D. degree with the School of Software. His research interests include the industrial internet, the Internet of Things, and network scheduling.

Rongwu Xu received the B.E. degree from the Department of Computer Science and Technology, Tsinghua University, in 2022, where he is currently pursuing the M.S. degree with the Institute for Interdisciplinary Information Sciences. His research focuses on applying machine-learning techniques to solve network security and privacy problems.

Xinlei Chen (Member, IEEE) received the B.E. and M.S. degrees in electronic engineering from Tsinghua University, China, in 2009 and 2012, respectively, and the Ph.D. degree in electrical engineering from Carnegie Mellon University, USA. He is an Assistant Professor with Shenzhen International Graduate School, Tsinghua University. He was a Post-Doctoral Research Associate with the Electrical Engineering Department, Carnegie Mellon University, from 2018 to 2020. His research interests lie in AIoT, pervasive computing, and cyber-physical systems. He has won several awards from top-tier conferences, including the Best Poster Award from IEEE/ACM IPSN, the Best Demo Award from ACM SenSys, and the Best Paper Award from CPD Workshop of ACM UbiComp.



Yunhao Liu (Fellow, IEEE) is a Professor with the Department of Automation and the Dean of the Global Innovation Exchange, Tsinghua University, Beijing. He was a Chang Jiang Professor and the Dean with the School of Software, Tsinghua University, from 2013 to 2017. From 2018 to 2019, he was the Chairperson Designee with the Department of Computer Science and Engineering, Michigan State University. His research interests include the Internet of Things, wireless sensor networks, indoor localization, the industrial internet, and cloud computing. He is a Fellow of ACM.