LightSync: Unsynchronized Visual Communication over Screen-Camera Links

Wenjun Hu
Microsoft Research Asia
wenjun@microsoft.com

Hao Gu*
Microsoft Research Asia and USTC
v-haogu@microsoft.com

Qifan Pu*
University of Washington and USTC
qp@cs.washington.edu

ABSTRACT

A key challenge for smartphone based visual communication over screen-camera links is imperfect frame synchronization. The difficulty arises from frame rate diversity and variability due to camera capability, lighting conditions, and system factors. On the 4 smartphone cameras we tested, the frame rate varies between 8 and 30 frames per second. If the transmit frame rate is too high, the receiver might lose original frames or capture mixed frames, which are normally not decodable. Previous systems simply reduce the effective screen frame rate to be half the camera frame capture rate, to guarantee receiving a decodable frame every other frame. This under-utilizes the transmitter side capacity and is inefficient.

We achieve frame synchronization with LightSync, which features in-frame color tracking to decode imperfect frames and a linear erasure code across frames to recover lost frames. LightSync allows smooth communication between the screen and the camera at any combination of the transmit and receive frame rates, as long as the receive rate is at least half the transmit rate. This means that each receiver can scale the decoding performance with its own camera capability. Across several phones, our system can more than double the average throughput compared to previous approaches.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Wireless Communication

Keywords

Screen-camera links, unsynchronized communication, per-line tracking, inter-frame erasure coding

1. INTRODUCTION

Smartphone possession is rising quickly, fostering growing interest in and efforts to enrich smartphone based user interaction and experience. Perhaps one of the most common scenarios is acquiring information via the phone, e.g., in retail and tourism. For example, QR codes are now widely adopted for advertising, because many smartphones ship with a built-in 2D barcode scanner or can be installed with such an app easily. Many brochures often come with one QR code per product[6]. There are even initiatives like Gibraltar targeting tourists with Wikipedia QR codes[1], where a QR code is placed next to each point of interest, and scanning the code will take the user to the corresponding Wikipedia page. In China, for example, it is increasingly common for websites to show several QR codes on a page and even TV adverts include QR codes. Mary Meeker's latest predictions on mobile Internet trends report 4 times more QR codes being scanned in March 2013 than a year before[4].

However, the capacity of a single, static 2D barcode is very limited. Indeed, existing QR codes often encode a single URL, and require cellular or WiFi connectivity for access to additional information. On the other hand, public displays are also widely available, and, when combined with smartphone cameras, form communication links that can complement RF wireless connectivity, as suggested in previous works[10, 15]. Therefore, we can imagine embedding not only a web link, but also the full information on the linked page, in a video of 2D barcodes. Screen-camera links enjoy several advantages compared to traditional RF wireless connectivity. They are free, directional, and free of interference. There is no overhead in link setup and management, and therefore it is well suited to one-time transfer of a small amount of information. Consider the Gibraltar initiative again. The text in the entire Wikipedia page for Gibraltar can be stored in a `.txt` file of 37.8 KB, while the text for Gibraltarian cuisine is only 4.6 KB in size. Transferring either text represents a suitable workload for the screen-camera link. We can envision a display playing the barcode video continuously or on-demand, and recording the video on the smartphone will download the Wikipedia page directly.

To achieve a high capacity, real screen-camera communication systems must address such challenges as imperfect frame synchronization, image distortion, and computational complexity. While previous research often focused on reducing image quality distortion to increase the per-frame capacity, we aim to solve the basic synchronization problem in such a system to push the frame rate. This is orthogonal to per-frame improvement. In other words, we consider temporal domain coding, compared to spatial domain coding in most previous works.

Frame synchronization is a prerequisite for effective communication. The camera must be able to detect the displayed frames from individual captured video frames. In a Gibraltar like scenario, there is an additional requirement that the same video playback rate should work with any

*Part of the work was performed while Hao Gu and Qifan Pu were research interns at Microsoft Research Asia.

MobiCom'13, September 30–October 4, Miami, FL, USA.

http://dx.doi.org/10.1145/2500423.2500437 ...$15.00.

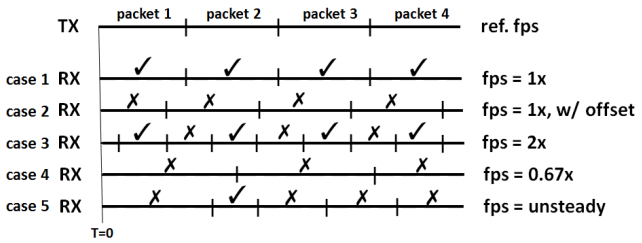


Figure 1: Mismatched frame rates between TX-RX pairs. "|" marks the boundary between two consecutive frames. ✗: "mixed" frames, ✓: "clean" frames. Inter-frame intervals are omitted.

phone camera that wishes to record the video. Implicitly, this is a broadcast system.

Lack of synchronization is mainly caused by mismatched frame rates and a phase offset between the screen and the camera. Figure 1 shows several examples. Typically, only frames staying within the period of one transmission are *clean* (marked "✓"), and thus decodable. If a captured frame spans different TX frames, it becomes a *mixed* frame (marked "✗"). Worse, smartphone camera sensors typically employ a *rolling shutter*, which scans a scene one line at a time. This makes the frame mixing vary on a per-line basis. Such a frame has to be dropped usually. We might also lose frames due to discrete shutter sampling, if the screen's display frame rate is higher than the camera's capture rate. Only after receiving all the transmitted frames 'cleanly' can a receiver successfully decode the original message (cases 1 and 3).

The difficulty in frame synchronization arises from frame rate diversity and variability. With a traditional coding scheme, a receiver should capture frames at a rate at least twice the transmitter's display frame rate, to guarantee receiving all frames cleanly (case 3). Unfortunately, in terms of rate capability the opposite often happens in practice. Modern LCD or AMOLED screens usually have a 60 fps rate or at least 30 fps, while cameras have lower and more diverse frame rates of up to 30 fps. For example, HTC One X[2] and Nokia Lumia 920[5] support video recording at up to 24 and 30 fps respectively. Worse, many smartphone cameras exhibit unsteady frame rates due to hardware design, firmware settings, and API constraints. Our tests on several phones (Section 2.3) show that the camera will automatically adjust its frame rate according to the perceived lighting conditions. We observe heterogeneous frame rates for the same phone across videos, and for the same video across phones. Therefore, it is neither desirable nor possible to control the frame rate at the application level.

To achieve frame synchronization, previous systems like PixNet [15] and COBRA [9] simply reduce the screen frame rate in the hope that the receiver can pick out a good frame every other frame. Langlotz et al [12] embed 2D barcodes into time-multiplexed 4D barcodes by repeating frames on multiple, orthogonal color channels, essentially limiting the transmitter's frame rate.

However, there are several fundamental drawbacks with these approaches. First, these approaches are inefficient. The transmitter side capacity is under-utilized, while the receiver drops a large number of mixed frames that actually contain much information. Second, they assume an initial sender setup based on receiver capability, which in practice requires feedback and needs to tailor to individual receivers. Third, these approaches are still unreliable in the face of a fluctuating receive rate, which makes it difficult to work with a general purpose phone camera. For COBRA, the authors fixed the receive rate via a firmware change.

Instead, we argue that the display rate could be higher. The key to displaying frames more quickly is to decode imperfect frames and recover any lost frames on the camera.

We design LightSync to achieve faster and more flexible frame synchronization. LightSync allows each phone to receive at its own capacity. Across links, this avoids the system being bottlenecked by the least capable smartphone camera.

LightSync features two main components. First, it adopts in-frame per-line color tracking to decode heterogeneous frame mixes due to the rolling shutter. Second, it employs inter-frame erasure coding and frame-based tracking to recover lost frames and guard against incorrectly decoded frames. This provides reliable information transfer. Our design also takes into account the computational capability of typical smartphones to enable real-time decoding. In our system, all captured frames contain useful information and can contribute to the overall frame recovery. Note that our design mainly addresses frame synchronization, and therefore could be viewed as an outer code over a base 2D barcode per frame. In that sense, our scheme is orthogonal to existing 2D barcodes like QR codes, PixNet, and COBRA. Furthermore, the notions of per-line tracking and inter-frame coding are generically applicable to screen-smartphone links, even though our current design targets the one-way transfer scenario.

We implement LightSync as a frame generator for the display and an Android phone app to decode the frames in real time. Our system works with off-the-shelf devices and existing general-purpose APIs.

To compare more broadly, we also capture the test barcode video on more phones, and run an offline decoder on the desktop. We find that our test phones can decode the video irrespective of the display frame rate. This means that the screen can display at a high rate so that each phone can get the best performance in line with its capability. Compared to the base line, we achieve up to 2.3× the average throughput across phones.

To summarize, we make the following contributions:

First, we characterize frame rate variability on several smartphones and the challenges to frame synchronization for successful decoding. We address the above challenges with an inter-frame erasure code.

Second, we use efficient in-frame per-line color tracking to resolve artifacts caused by the rolling shutter, which enables us to decode mixed frames. As a result, any received frame is decodable and useful.

Third, with the per-line tracking and the inter-frame code, the screen-camera communication can proceed smoothly at any combination of the transmit and receive frame rates, provided the receive rate is at least half the transmit rate. Since our one-way communication scenario is implicitly a broadcast system, our techniques allow each receiver to scale the performance with its own camera capability and avoid being bottlenecked by the least capable device.

2. SYNCHRONIZATION CHALLENGES

We start with an overview of the typical screen-camera communication process and discuss the main challenges for unsynchronized communication. Although we motivated such communication with a one-way transfer scenario, the issues discussed in this section are broadly applicable to these links.

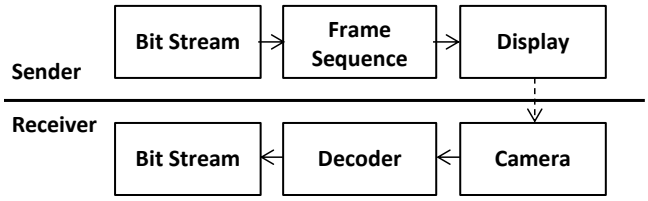


Figure 2: Screen-camera communication.

2.1 Screen-camera communication overview

With the screen acting as the transmitter and the camera the receiver, a pair of screen and camera forms a communication channel.

Figure 2 shows the typical encoding and decoding processes of such a system. This resembles the typical process in RF communication, although specific techniques differ when mapping bits to signals, generating and sampling signals. Hence we can reason about many issues in the same way. For example, we need mechanisms for frame detection and channel compensation. An immediate issue is the frame rate at the transmitter and the receiver respectively, measured in frames per second (fps). Naturally, they need to match in order for both sides to synchronize on the same frames. In particular, conventional decoding mechanisms require a distinct displayed frame to be detected within a captured video frame to ensure successful decoding.

2.2 Display (Transmit) frame rates

Modern LCD or AMOLED screens can typically display content at 30 fps or 60 for high definition videos, although there could be minor variances. With pre-generated video frames, the main cause to the variance is the interplay between when a new frame is written to the LCD's frame buffer and the LCD rendering the current frame. Generating the video frames on the fly, for example from a compressed format, would further add to the variation.

Given the typical video configurations, however, it is more common for videos to be played at 24 or 30 fps, in line with the human visual perception capability. Although this is not a hard limit for our system, off-the-shelf screens and cameras are normally designed with this in mind.

2.3 Capture (Receive) frame rates

Variability and unsteady inter-frame intervals. Compared to the transmit side, the received frame rate exhibits more variability for several reasons, and such variability affects the captured image quality.

The camera needs sufficient light exposure to capture legible frames. The time it takes to harvest sufficient light depends on the sensor hardware sensitivity and the lighting conditions, in particular, the contrast and intensity levels. Therefore, it is necessary to adopt a different frame rate given specific contrast and intensity levels, and cameras normally adjust this rate automatically to ensure the visual quality for the captured video.

When saving captured frames to the storage, we can also run into system factors. This is specific to cameras fitted on smartphones and tablets. There might be interruptions due to background processes. On our Huawei phone, we sometimes observe an inter-frame interval twice the usual length. There are also artifacts due to the API. For example, our HTC One X camera appears to record videos at close to 24 fps, but the camera callback API can only support saving the frames at 15 to 20 fps. Further, the callback slows down on the One X when there is insufficient memory available in the system. The frame rate on the Samsung Galaxy S III[7] is only stable if we lock the CPU to its maximum frequency with its `setCPU` app. Otherwise, it fluctuates hugely between 21 to 29 for an entirely white foreground, averaging around 25 fps. Even when the frame rate appears steady, the inter-frame interval still varies. Figure 3 shows examples of inter-frame intervals on two phones. The intervals measured on the One X oscillate significantly, which can also introduce arbitrary phase offsets between the transmit and receive sides. Although these issues are artificial and could be addressed with API or firmware fixes, they highlight the complexity of working with off-the-shelf devices.

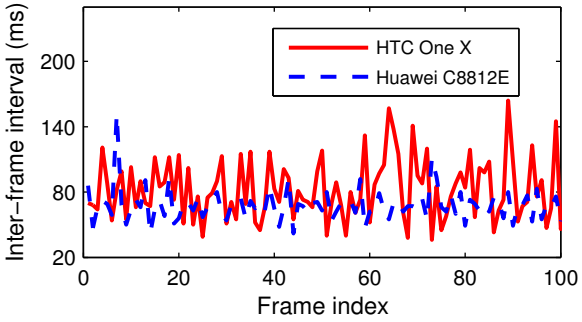


Figure 3: Example inter-frame intervals.

As exposure time increases with decreasing frame rate, there are several implications on the recorded image quality. While one might expect some minimum average intensity, the effect is non-uniform across each image, depending on the dynamic range of the camera sensor. For example, the probability of over-exposure in some part of the image increases. Moreover, the image becomes more susceptible to blur due to any camera motion during prolonged exposure.

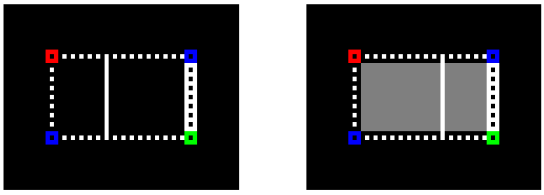
Operation range and diversity across devices. The maximum frame rate is determined by the hardware quality, for example, the sensor quality and the amount of heat the device can handle. Table 1 lists the specification of several smartphones. A Nokia Lumia 920, which boasts of great camera quality, can support up to 30 fps, whereas a Huawei Ascend C8812E[3] can only support up to 15 fps. At the low end, the minimum frame rate depends on how much light can be captured in the focal area. Both phones support at least 8 fps, though not for the same images.

Since it is typical to have a certain barcode pattern, we perform a simple experiment to assess the achievable frame rate in practice. We make four test barcode videos (Figure 4), display them on a Dell LCD (2007WFP) turned to the lowest brightness, and record the videos on several phones, placing the phone at a distance to capture the entire video frame[1]. In each video frame, we set the code area of the barcode to black, gray, or white, and the surrounding color to white or black. This changes the average intensity of each video frame, and the camera automatically compensates for that by adjusting the frame rate. Table 2 summarizes the results. We calculate the *effective frame rate* by counting how many camera API callback function calls can be made per second. There are two interesting high-level findings. First, the Lumia 920 can achieve 30 fps even for the darkest video.

[1]We have verified that brightness beyond the focal area does not affect the results.

Table 1: The smartphones used in the experiments and their specifications.

Model	Display size	Video camera	OS	CPU
Ascend C8812E	480 x 800, 4 in	720p, 15fps	Android 4.0.4	1 GHz
HTC One X	720 x 1280, 4.7 in	1080p, 24fps	Android 4.0.3	Quad-core 1.5 GHz
Galaxy S III	720 x 1280, 4.8 in	1080p, 30fps	Android 4.0.4	Quad-core 1.4 GHz, ARM Cortex-A9
Lumia 920	768 x 1280, 4.5 in	1080p, 30fps	Windows 8	Dual-core 1.5 GHz Krait



(a) Video 1 (b) Video 2

(c) Video 3 (d) Video 4

Figure 4: Snapshots of test videos for receive frame rates. The column in the center area moves to the right by one block per frame. Barcodes in Video 4 are surrounded by some white space.

Table 2: Recording frame rates per second.

Video	1	2	3	4
Ascend C8812	8	10	15	15
HTC One X	13	15	15	15
Galaxy SIII	17	26	30	30
Lumia 920	30	27	30	29

Second, we can ensure a minimum rate even on the low-end phone. The latter is helpful, since frames captured at a very low frame rate are more susceptible to blur or over-exposure, and are harder to decode correctly. Note that the results for Lumia 920 exactly show frame rates are not steady.

To conclude, we observe heterogeneous frame rates for the same phone across videos, and for the same videos across phones. By adopting certain contrast features in the barcode design, we have some control for each phone, although we still need to deal with a range of different frame rates across phones even for the same raw video.

In previous works[15, 9], the frame rate on the receiving phone was fixed to 30 through firmware hacking. The above result clearly shows that a fixed rate may be undesirable, for breaking the camera's adaptive lighting compensation algorithm and potentially producing low-quality images.

Sensitivity. We next assess the sensitivity of the frame capture rate and the generality of the above observations. Our findings are summarized below:

1. As long as the intensity of the focal area is stable, the frame rate hardly changes. Within each camera's operating frame rate range, the camera may only support a few rates.
2. Screen brightness has little effect, since even the minimum brightness on our test LCD is bright enough given an average intensity of the focal area.
3. The distance between the screen and the phone only matters to the extent of the average intensity of the focal area.
4. Different environments, such as outdoor, shopping mall, and office, mainly feature different ambient light settings. The typical displays we can find in these environments are bright enough to stabilize the frame rate, although the image quality differs.

2.4 Frame synchronization

Received frame patterns. Given various potential combinations of the transmit and receive frame rates, we perform a simple experiment to study the received frame pattern. We play Video 4 at several playback rates, and record the video with the Ascend phone. Note that the inter-frame intervals on the receiver side essentially introduce random phase offsets between the two sides.

Figure 5 shows the received frame pattern. For each received frame, i.e., a captured image on the camera, we can detect which originally displayed frames are present[2], and plot a circle for each detected frame. When two or more detected frame indices correspond to the same received frame index, this captured frame is a mixture of two or more displayed frames. Any gap indicates a missed frame, i.e., neither of the consecutive received frames captured the displayed frames corresponding to the gap.

At 7 fps (slightly less than $\frac{15}{2}$), we capture at least one complete frame every other frame. The 'other frame' normally has split original frames due to a phase offset.

When the transmit rate approaches 15 fps, the receive rate, almost every frame is a mix of two frames, but there are occasional missed lines. Split frames are common, and the component frames at the top and bottom might differ.

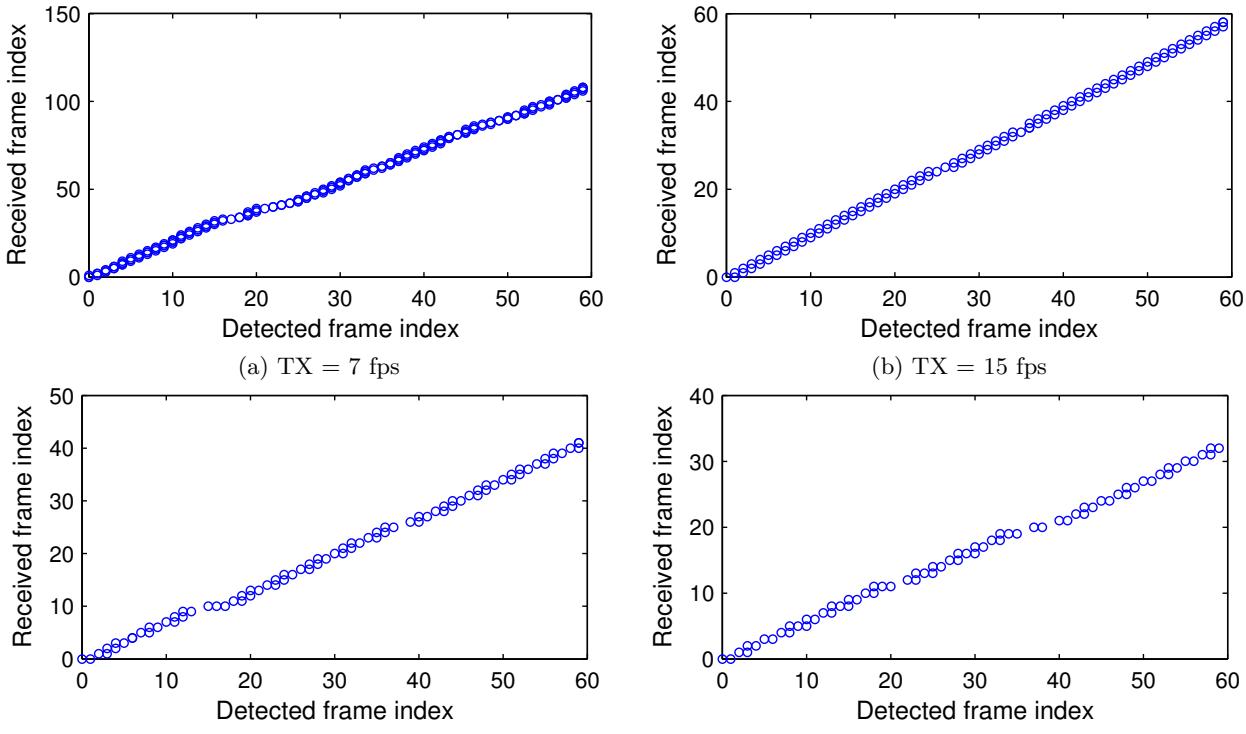
As the transmit rate increases further, we start to capture 3-frame mixes and miss frames at times. However, consecutive losses or 3-frame mixes are very rare. Missing lines or frames are caused by discrete shutter sampling or delayed return from the camera callback function. In a 3-frame mix, the middle frame is present in its entirety.

Since the Ascend records videos at 15 fps, detecting the same number of displayed frames requires receiving far more frames at a low transmit rate. Therefore, the scales of the vertical axis are different across the subfigures of Figure 5.

Rolling shutter. If we examine the mixed frames more closely, we see that the extent of overlap can vary by line, even when the same component frames are present at both the top and the bottom. This is caused by a *rolling shutter*, meaning that the sensor scans the scene line by line to synthesize the complete image. This is specific to CMOS camera sensors, which are the norm for smartphone cameras due to their low energy footprint. Newer smartphone models try to reduce the shutter lag to mitigate this effect. In contrast, CCD sensors, typically used for high-end cameras, capture the whole image at a time.

Frame synchronization. Conventionally, only completely clear frames can be decoded, and therefore the goal of frame

[2]This is part of the LightSync decoding process, detailed in Section 3.3.



(a) TX = 7 fps

(b) TX = 15 fps

(c) TX = 20 fps

(d) TX = 25 fps

Figure 5: Received frame patterns on the Ascend.

synchronization is to adjust the frame rates on both sides to capture enough such frames to decode all original information. In the light of the above received frame patterns, the typical frame synchronization approach is to have the receive capture frames twice as fast as the display, in the hope of ensuring a good frame every other frame. This also appears to fit in with the Nyquist sampling requirement.

However, this approach is wasteful, since a large number of imperfect frames are dropped, which could provide useful information. Further, since each frame is a discrete sample already, in theory we could have equal frame rates on both sides. The key to achieving this is to decode imperfect frames and recover lost frames. The latter can be easily addressed with erasure coding across the original frames, which can also help recover incorrectly decoded frames. The former is complicated by inherent image quality issues.

2.5 Challenges in decoding imperfect frames

Heterogeneous mixes. If each frame was *homogeneous*, then we would only need a per-frame tracker to record the proportion of each component frame.

Unfortunately, there are two practical issues. First, we often obtain mixtures between different frames in the same captured frame, e.g., the top half is from original frames 1 and 2, while the bottom half is from 2 and 3.

Second, given the rolling shutter, we need to track the proportion of mixture variation per line even for the same component frames.

Inherent image quality issues. Image quality is generally lower for smartphone cameras than for regular cameras due to form factor and cost constraints. This is reflected in such issues as color distortion around the focus within an image and the dynamic range of the color spectrum. Compensating for these issues becomes more challenging for mixed frames.

When mixing colors, there is a tradeoff between being able to distinguish all mixtures and ensuring the result color stays within the dynamic range. Most notably, having too many white blocks mix will cause over-exposure at the mixed position. Therefore, designing for many-component mixed frame is near impossible.

The main solution is to discretize the color levels used, i.e., use *sparse* colors so that we can disambiguate their combinations. This means we cannot expect much 'soft information' between the discrete levels.

3. LightSync DESIGN

LightSync achieves fast frame synchronization by making it possible to decode imperfect frames. Specifically, we address rolling shutter with multiple tracking bars and per-line tracking within each frame. We adopt a linear erasure code across the original frames to recover lost frames and leverage partial or mixed frames. In some sense, frames are synchronized by default in LightSync. Practically, synchronization (or resynchronization) is achieved as long as we can identify the component frames of each received frame.

3.1 Line-based overlap tracking

With the rolling shutter effect, frame mixes vary by proportion at different lines, sometimes involving different frames. We resolve these with standard color bars, which form an implicit encoding pattern.

Encoding pattern. Generally, we need to insert tracking bars in each frame in the shutter rolling direction (perpendicular to the scanning direction). These can track the percentage overlap at each line by showing the reference colors. Our current design is based on findings from the experiments in the previous section. We target a frame capture rate that is at least half of the display rate.

To recap, we only need to resolve mixed frames with up to 3 component frames, i.e., each block may have contributions from up to 3 blocks. In such a mixed frame, the middle frame always takes up 50% of the mixture, whereas the re-

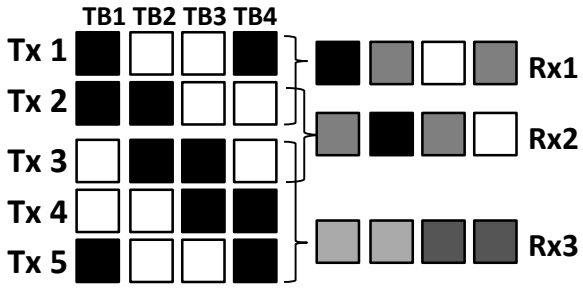


Figure 6: Per-frame tracking bar pattern.

maining 50% is split between the first and the third frames. Further, color distortion on the Huawei phone only allows us to reliably use two grayscale levels as the base color blocks, i.e., black and white. Thus, there are only 8 possible color combinations at each block. All white and all black are easy to detect and do not vary by line. White-black-white and black-white-black will both produce 50% gray at any line, and therefore are two further trivial cases. The remaining four combinations may produce varying results depending on the color composition and need to be tracked.

With this in mind, we place four bars in each sent frame, two white and two black. Figure 6 illustrates the color pattern. Each vertical line represents a tracking bar, while each horizontal line represents the color pattern of a line in a frame. The color pattern shifts by one bar in each successive displayed frame, and the overall pattern across frames repeats every four frames. As a result, all four combinations are present in any received frame, and the four blocks in each line serve as reference colors for per-frame decoding. They are analogous to training sequences in traditional RF wireless communication.

Decoding mixed frames. For any block in a mixed frame, the decoding algorithm proceeds in two stages.

First, we determine the exact displayed frames present at this block. This is needed if the top and bottom frame indices differ or one of the frame index blocks is too dim, and only needs to be done once per line. Given the tracking bar encoding pattern, if the first or the last frame is missing, one of the bars will be completely black, and another completely white. Further, the positions of these two depend on which frame is missing. Consider the first three frames in Figure 6, if the first frame is missing (Rx 2), TB4 will be white, and TB2 will be black. If the third frame is missing (Rx 1), TB1 will be black, and TB3 will be black. This process implicitly detects lost lines from the displayed frames, because blocks from those lines will remain unknown after per-frame decoding. By identifying the component frames per line, we are also able to handle practically any phase offset between the displayed and the captured frames.

Second, we compare the code block color with the reference colors to determine the code block sequence in the original frames concerned. There are three known colors, white, black, and 50% gray, and four tracking block colors. If the code block is matched to only one reference color except the gray, the block sequence can be unambiguously decoded. Otherwise, we look up the block at the same position in the previous captured frame to help resolve the ambiguity. If that still fails, we mark this block unknown, and leave it to the cross-frame correction stage, combined with undoing the erasure code.

We determine a color match if the code block and the reference colors are within a small range of each other. Given color distortion due to the lens, readings for the same color at the center and at the edge of the image might differ by 5 to 10%. Therefore, for any two-component mix, the range is set to 10%. We can decode a 3-component mix with less than 10% contribution from the third frame as if a two-component mix. To allow for this slight additional error, we relax the range by a further 5% for 3-component mixes. Relaxing the range too much would cause too much unnecessary ambiguity, and hence we keep it tight.

In theory, we can calculate the exact percentage mixture from the tracking bars. However, color distortion makes this generally impossible. Instead, the inference based approach is simpler, and allows us to decode multiple displayed frames from a single mixed frame captured.

Implicit cross-frame error correction. The tracking bar pattern embeds error correction capability, and we leverage this to ensure correct decoding. The color thresholds are set such that if the code block color matches that of only one tracking block, the original blocks concerned are guaranteed to follow the pattern of the matched tracking bar and can be decoded correctly. Further inference based on already decoded blocks is then guaranteed to yield correct decoding results. Extending this further, the decoding results from decoding the inter-frame erasure code are also guaranteed correct. This resembles hard-decision decoding.

3.2 Inter-frame erasure coding

Erasure coding in LightSync. One of the key ideas of LightSync is to apply erasure coding across the original frames, so that each sent frame may contain information from multiple original frames. This also means that any partial or mixed frame is just a version of coded frame and can be decoded as long as we keep track of the coding coefficients. Therefore, once we receive enough linearly independent frames, we should be able to recover all original frames. This makes it easy to recover lost frames.

Ideally, each encoded frame should be a linear combination of all original frames, while linearly independent of any other encoded frame. This would allow us to recover all N original frames from any N encoded frames. However, recovering the original frames requires solving a simultaneous linear system, and the computational complexity increases quickly with N. Therefore, we opt for a simpler, suboptimal coding scheme.

Our previous study shows that frame losses are rare, and there are no more than two consecutive losses given the typical frame rate range on our phones. Furthermore, we expect the capture rate to be at least half the display rate, which means that we expect to receive all frames after the transmitter 'repeats' all frames once. Therefore, we only need a very simple form of erasure coding to recover such losses with low computational overhead.

For our purposes, each frame can simply be treated as a stream of bits, and our erasure coding process produces a coded bit stream of the same length. Therefore, the inter-frame coding is independent of the base code per frame, and can be viewed as an outer code over a 2D barcode. The coded bits within a frame can then be mapped to colored blocks and the blocks arranged in some pattern following the per-frame base code. Optimizations for the base code, such as the blur-aware color ordering in COBRA, can equally apply

here. We might even be able to apply such techniques in the temporal domain, although we leave this to future work.

Low-complexity coding. We use a sparse and deterministic coding matrix for low complexity recovery.

First, we simply transmit the original frames verbatim as the first half of our frame sequence. Second, we generate encoded frames to complete the second half. We divide all original frames to groups of three consecutive frames. This is because we expect no more than one frame loss per three frames, and coding together few frames will speed up decoding. For each group of original frames F_1, F_2, and F_3, the coded frames are $F_1 \oplus F_2$, $F_2 \oplus F_3$, and $F_1 \oplus F_3$.

Our coding matrix is quite suboptimal in guarding against linear dependency, since the second half of the sequence is generated from the original frames in the first place. However, we trade off suboptimal coding for low-complexity decoding. Further, even seemingly linearly dependent later frames are still helpful when per-frame decoding did not succeed 100%. We will show examples in Section 5.

We apply the matrix to bits at the same position of the frames within each coding group, which implies no coding within or across lines. Coding within or across lines of a frame offers little additional redundancy in the event of a lost frame, since the entire frame is lost anyway[3].

Stopping criteria. For our main scenario, the display simply plays the entire code frame sequence in a looping fashion. The phone client stops recording the video once enough frames have been successfully decoded.

Having the original frames first is helpful if the video is played on-demand, because a high-rate phone is less likely to lose the first few frames, and would need to target a shorter decoding delay. Later iterations can be viewed as auxiliary transmissions to provide incremental redundancy.

If we had a unicast scenario and a duplex communication channel, the receiver could acknowledge receipt of all frames to stop further transmissions.

Frame tracking and on-demand decoding. We insert a few fields in each frame to keep track of the sent frames. First, each frame carries a frame sequence length, which is twice the total frame count. This gives an indication of how many frames need to be captured, although the exact number of frames needed might deviate from this total depending on the decoding performance. Second, each frame carries the sent frame indices within the sequence of all coded frames, from which we can determine the component original frames.

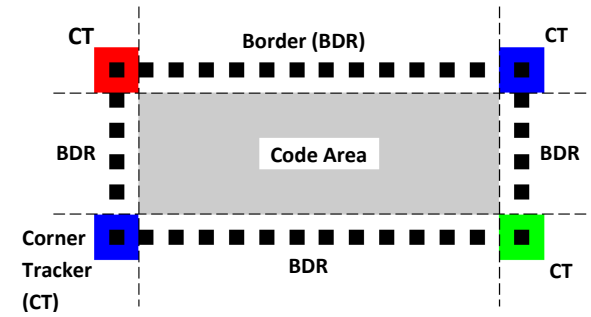
Once we have the coded frames and the rows of the coding matrix these correspond to, decoding is straightforward. For our coding scheme, we only need a few XORs to recover the original frames. Since we often lose only part of a frame, we perform decoding only for the lines with missing information.

Note that we are effectively encoding and decoding in a line-by-line fashion within each frame. Therefore, the current design is suboptimal, because we still transmit whole frames. In future, we can extend the design to mimic transmitting and receiving per line, analogous to partial packet recovery in wireless networks[11].

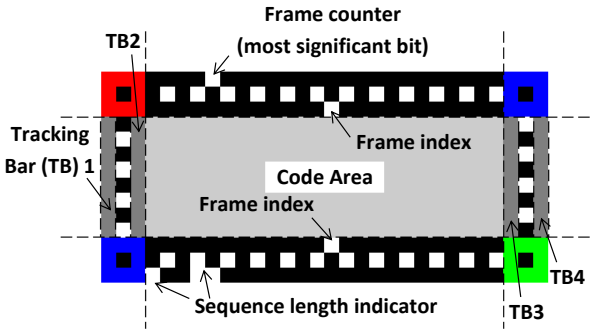
3.3 End-to-end system flow

Frame format. Our techniques are not related to the per-frame layout, so we can mostly follow existing designs, e.g.,

[3]They can provide more protection against perspective distortion, but this is beyond the focus of the current work.



(a) Corner and border design in COBRA



(b) Additional fields for LightSync

Figure 7: Frame layout.

that for COBRA, since COBRA includes optimizations for fast code block extraction[4]. Figure 7 shows an example frame. The corners and borders are basically the same as in COBRA, except that we alter the vertical border pattern slightly. Our border blocks per line have different colors, which aids color calibration. Although not apparent in Figure 7(b), we do surround each barcode with some white space to ensure the average light intensity of the frame. This follows the observations in Table 2.

Generally speaking, the tracking bars need to be perpendicular to the scanning direction (i.e., in the shutter rolling direction), whereas the frame tracking fields should be in bars parallel to the scanning direction. We currently arrange the code block in a landscape orientation, and hence the shutter scanning direction is horizontal.

We fit the frame tracking fields and the tracking bars in the spare space around the border blocks to avoid additional overhead. The four tracking bars are placed to the left and right of the two vertical border bars.

There are four frame-tracking fields in the bars around the top and bottom border bars. These index the sent frame within the entire coded frame sequence. They reset when the frame sequence is repeated. We use white block on black bars, since the camera is more receptive to white and better preserves the frame information this way.

The *coded frame sequence length* is represented in *binary* in the space below the bottom border bar, and is the same in all frames. This is twice the number of the original frames. The bar above the top border shows *the most significant bits of the frame index*. It is represented in *unary* to be distinguishable even in a mixed frame. The bar immediately below the top border bar shows in *unary* the *frame index modulo the bar length*. A mixed frame will have multiple blocks set in this bar to indicate the component frames. The

[4]We will discuss how to improve the layout in Section 5.

bar immediately above the bottom border bar shows the same information. This is because the mixture components can change within a frame.

Encoding and transmission. We start with a bit stream divided into frames, and generate an erasure coded bit stream. Frames are then generated following some bit-to-block mapping, such as that for COBRA or data matrix. Any applicable code block optimizations can be added too. This is our base code. We then add frame tracking fields and the line-based tracking bars to each encoded frame. Finally, these frames are turned into a video and played in a looping fashion at a given frame rate.

Reception and decoding. Decoding bootstraps by identifying where in the coded sequence the first frame is. Note that part of the first received frame may need to be discarded, since there might be insufficient information of the previous frames. We start with per-frame decoding and then invoke cross-frame correction after receiving half a sequence.

Per-frame decoding proceeds in two stages. For each captured frame, we first detect the corners and the borders, and calculate all code block positions following the block localization algorithms in COBRA. We then extract the color values at these locations. Note that each position corresponds to a single pixel, but we average the color readings of a 3×3 square centered at this position as the reading for the block. This can reduce errors due to color distortion.

In the second stage, we find the frame tracking blocks to record the component frame index information. If we determine that the received frame contains information of any original frames that have not been fully decoded, we decode the code blocks in this newly received frame line by line. At the start of each line, we first calibrate the data block by comparing its reading to the black and white border blocks in the same line, and then decode the block by comparing its color to those of the tracking blocks.

Once the decoder detects a second half of the video sequence in the received frame, we start cross-frame correction after decoding each new frame by undoing the erasure code and filling in any missing blocks at the relevant positions if possible. Frame capture and decoding both terminate if all original frames have been decoded. The blocks are then mapped to bits to output a bit stream.

API. LightSync can serve as a piece of middleware, taking a bit stream as input at the transmitter, and producing a decoded bit stream as output at the receiver. An application written on top of LightSync just needs to generate and interpret the bit stream accordingly.

3.4 Discussion

In signal processing terms, unsynchronized communication in our scenario is essentially a form of *undersampling*, where the sampling rate is lower than the Nyquist rate. Mixed frames correspond to *aliasing*. The reason and condition for being able to resolve such aliasing in our setting is that our raw signals, the blocks taking on black or white, are *sparse* in the available color spectrum. This ensures a small number of possibilities when aliasing occurs, such that we can still disambiguate many raw signal combinations that produce the aliases.

The use of tracking bars and linear erasure codes is generic, although our current design takes into account the particular challenges and requirements of the current system. In theory, the design would scale to apply directly over orthogonal color channels. We will explore this in future work.

As technology improves, we might expect to adopt a simpler design yet capable of higher performance. For example, if we had CCD camera sensors employing *global shutters*, the per-line tracking could be significantly simplified. The frame counters can almost fully determine the component frames, and the vertical tracking bars only need to detect the transition across frames.

Given we focus on resolving frames temporally, we assume a reasonable image quality for each received frame so that we can extract the code blocks per frame. In other words, we expect the afore-mentioned first stage of per-frame decoding to succeed[5]. Mechanisms to guard against poor image quality, possibly caused by motion-induced blur or poor lens focus, would also improve the robustness of our scheme, though they are orthogonal to our main focus.

4. LightSync IMPLEMENTATION

We implement LightSync to work with a standard LCD and an Android phone app.

Encoder. For the transmitter part of the system, we write a frame generator in Matlab. It takes a given bit stream, and produces encoded frames following a pre-specified bit-to-block mapping (default is 0 to black and 1 to white) and the frame layout described in the previous section. The frames are then turned into a video, to be played at a specified frame rate and displayed on the screen.

Decoder. The decoder is implemented in Java as a phone app using Android SDK 2.3.3 and Android development tools. The app runs two threads. One thread obtains the recorded frames using the camera callback API, and puts them in a queue to be processed. The other thread takes the frame at the front of the queue and decodes it in real time without saving the image. Decoding results for the frame are saved, which may involve code blocks in up to 3 original frames, even if there are unresolved blocks at this stage. Both threads are stopped when all original frames have been decoded completely.

As we explained earlier, the output of LightSync can be piped into an actual application that will interpret the bit stream decoded in a meaningful way. For our experiments, however, we simply make the decoder a standalone app.

Changing the interfaces slightly, we also turn the online app into an offline desktop decoder that can take timestamped captured video frames and decode them. This helps us analyze the captured frames and the decoding results.

Optimizations for real-time processing. Our design for the erasure coding and the tracking bars have already taken into account the real-time processing requirements. In addition, we take three measures to achieve real-time decoding.

First, we minimize the number of pixels to process. Specifically, we limit the initial search area for the corners. Once the corners are located, we calculate the central pixel locations of the remaining blocks, including borders, tracking blocks, and data blocks. For each block, we only process the 9 pixels centered around the calculated location to extract the color value. From the second frame onwards, we

[5]Otherwise, the received frame will simply be dropped and the corresponding component frames considered missing. Although LightSync can handle such situations, it is not optimized for them.

also use locations from the previous frame as initial references. For example, we use the previous corner locations, and only briefly search around these to confirm or re-locate the corners. The number of border blocks on each side is only counted when processing the first frame.

Second, we avoid using any large high-dimensional array in processing the images, since Java does not provide efficient operations with such arrays.

Third, we use a single thread to complete all decoding operations[6]. This avoids unnecessary context switches.

Since our current implementation already meets the real-time target, we do not explore further optimizations, even though some operations could be sped up more.

5. EVALUATION

5.1 Experiment setup

Hardware. We use a Dell LCD (2007WFP) as the display and several smartphones. Table 1 lists the phones and their specifications. Note that the published specifications describe video capture frame rates in the recording mode, with the video saved directly to a compressed format. In contrast, our app runs in the preview mode to access the color information of the individual pixels in the raw images, and so might see a different capture rate.

Workload. Our workload is a file transfer that can be encoded into a 30-frame barcode video. Once erasure coding is applied, this turns into a 60-frame sequence played repeatedly. This means that the phone camera would only need to record for 1 to 2 seconds, which is a reasonable duration to expect the user to hold the camera for. Furthermore, the total of 60 frames in a sequence is a suitable lowest common denominator for our cross-phone experiments later in the section, so that we can capture a few rounds of the video given the available memory on each phone.

We play the video at 10 to 27 fps with `mplayer`. All phones try to recover the file from recorded video frames. The playback rates of 28 to 30 fps cannot be precisely controlled and are hence omitted. All the video players we tried use rounding when trying to achieve a certain playback rate. For 27 fps, the inter-frame gap is almost exactly 37 ms, and fairly steady. For 30 fps, the gaps should be 33.3 ms, but more variable in practice. These cause the display to sometimes skip (part of) a frame, which cannot be recovered on the receiving side. Although our erasure code can handle this to some extent, the frame skipping cannot be controlled and causes too much randomness across experiment runs.

Unless otherwise stated, we use a data area of 36×20 blocks per frame, 10×10 pixels per block. Including the borders and corners, each frame has 38×26 blocks. The phone is placed around 10 cm from the Dell LCD, though the exact distance varies slightly by the phone based on their individual display sizes. Note that these parameters are not hard operating constraints for LightSync, but do affect the image quality. In a real scenario, we can support larger distances by using much larger code blocks and displaying much larger barcodes to achieve acceptable image quality.

36×20 is much smaller a code area size than was considered in PixNet or COBRA. While our techniques scale with the code area size, the image quality does not. We experimented with such sizes as 60×40 and 48×26, and each suited a specific phone. When more blocks are packed, the block size decreases quickly in the captured frame, and we find that corner detection fails easily in the face of various lens effects (for example, color distortion on the Ascend and moiré pattern on the HTC One X). Larger corner and border blocks would be needed to offset such distortions. Since frame layout optimization is beyond the scope of this work and we assume a reasonable image quality, we stick to the small code area size for most experiments.

[6]A small number of threads may also be fine.

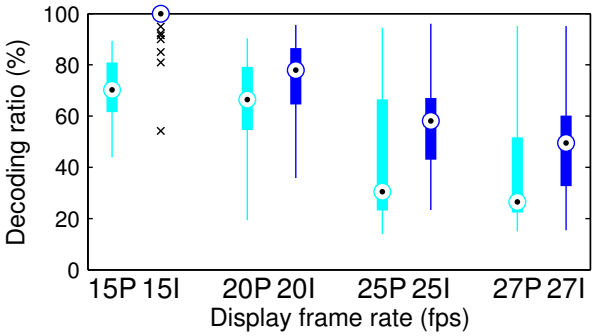


Figure 8: Per-frame decoding performance.

Metric. The main metric is the effective frame rate, seen in the transfer completion time. Minimizing the completion time can improve user experience, in that the user only needs to hold the camera for a shorter time. This would also reduce the chance of capturing blurred images due to hand motion. We will also calculate the equivalent throughput.

Since our implementation goal is to have the client application operate in real time, we time individual operations to assess their computational complexity.

5.2 LightSync microbenchmarks

We first evaluate the individual performance of the in-frame tracking and erasure coding as much as possible. We will also time each stage in the decoding process. These microbenchmarks are performed on the Ascend unless otherwise stated, which represents a low-end phone with an average quality camera. The CPU and memory are also at the low end among currently available smartphones.

Performance without erasure coding. In this experiment, we assess how well we can identify the component blocks of any received block by comparing its color to those of the corresponding tracking blocks. We collect 150 received frames for each setting on the One X, and calculate the per-frame decoding ratio (Figure 8). There are two boxplots per frame rate setting, one for *only* using tracking bars within a frame (suffixed with 'P'), and the other for additionally using the previous frame to resolve some ambiguous block combinations (suffixed with 'I'). Since we cannot control the frame mixing patterns, the received frames are composed from mixtures of different original frames and block patterns, and therefore large variances are seen for all these boxplots. Not surprisingly, decoding becomes more challenging with increasing display rate, due to more missing and 3-component mixed frames. Cross-frame disambiguation and correction are essential to decode the remaining blocks.

Efficiency of the erasure code. Our code is strictly suboptimal in guarding against linear dependency between received frames, so we evaluate its efficiency by considering the performance variation with the received frame pattern.

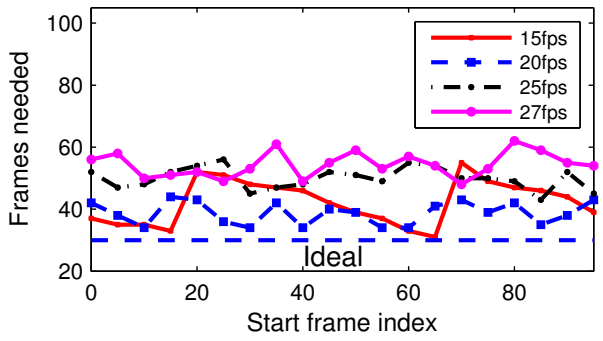


Figure 9: Efficiency of the inter-frame code.

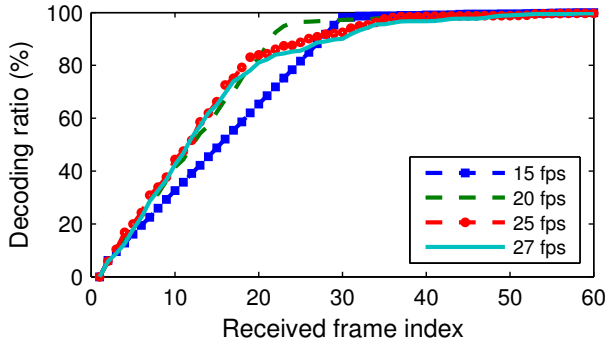


Figure 10: Decoding progress as frames are received.

We collect more than 100 frames for display rates at 15, 20, 25, and 27 fps, and start decoding from different points in the received sequence. Figure 9 plots the number of frames needed for successful decoding.

Recall the Ascend phone supports 15 fps, and therefore the received frame patterns are different at these display rates. Ideally, decoding should complete when we receive 30 frames. When the display frame rate is 15, the phone only occasionally misses a frame, and per-frame decoding often succeeds completely. Therefore, how many extra frames are needed depends on the position of the missing frame, and it is uniformly distributed between 1 and 30. As the display rate increases, per-frame decoding becomes more challenging and the decoding ratio drops. This is reflected in many more frames are needed, sometimes to complete a few blocks. However, the variance also decreases compared to that at 15-fps video playback, since most frames contribute at least a few lines of information.

Although the exact lines look different for a different phone, the general observation still holds. Figure 10 shows the decoding ratio of the blocks *in the 30 original frames* as more frames are received for a comparable run on the One X across display frame rates. We assume decoding only starts when the first original frame in the video is detected in a received frame. Initially the higher the display rate, the faster the growth rate of the decoding ratio. This is because a received frame is more likely to contain more component frames at a high display rate, and decoding a received block means several original video blocks are successfully decoded. Once the erasure code frames are detected, the decoding ratio grows more slowly for all display rates, because the decoder is now only filling in missing blocks.

Computational complexity and memory usage. Our target is to decode each frame within 33 ms to support 30 fps, the highest receive frame rate observed so far. Table 3 lists the time taken for each operation. The first frame takes longer because corner detection is the slowest part of the whole process. Later frames can build on the corner positions identified in the first frame, and only slight adjustment is needed. Border detection is essentially a no-op from the second frame onwards.

Table 3: Processing time breakdown in ms.

Operation	1st frame	Later
Corner detection	27	4
Border block count	6	1
Block extraction	9	10
Per-frame decoding	8	10
Linear decoding per frame	No-op	1
Per-frame total	50	26

The client app decodes the captured frames on-the-fly without saving the frames. Therefore, the memory requirement mainly includes caching the raw frame being processed and storing information for the frames already decoded. The former is about 200 KB for the Ascend (and 1 MB for the HTC One X). For the latter, each frame contains 36×20 data blocks, each mapping to 1 or 2 bits. If per-frame decoding is not successful, we mark the block 'unknown'. In all we only need 1 bit of state information per block after decoding, and therefore around 720 bits per frame. This is at most double the amount of raw bits carried in each video frame.

Sensitivity to distortion. The linear coding part of LightSync is entirely independent of the common distortions applicable to screen-camera communication.

Due to following the frame layout in COBRA, we cannot effectively assess how such distortions affect the tracking bars. This is because if the corners and borders are not sufficiently affected, neither will the tracking bars. Conversely, decoding cannot even start without successfully identifying the corner and border locations. In this sense, however, the current corner and border design acts as a natural guard to ensure the resilience of decoding using tracking bars.

Generally speaking, ambient light or screen brightness has little effect provided the border blocks used for calibration can be identified. The tracking bars can be affected by perspective distortion if a data block cannot locate its reference tracking blocks in the same line. However, we can extend the inter-frame coding to code across lines for further in-frame protection, and we will explore this in future work.

5.3 Performance comparison

Performance across phones. For this set of experiments, we capture all frames and decode offline. This allows us to analyze the images in more detail and to compare image quality across devices. Decoding stops when all 30 original frames have been successfully recovered. We calculate the effective frame capture rate as the number of frames needed to complete decoding over the time taken to capture these frames. Since we have previously shown that the processing is fast enough, we simply calculate the throughput from the effective frame rate and the per-frame capacity. We also run the app on all the Android phones to verify that the offline performance results match the real-time performance.

Figure 11 shows the effective frame capture rates achieved by each phone. The phone's actual capture rate is marked in the legend. All phones can achieve a decent frame rate at any display rate, which meets our initial goal of unsynchronized communication. Not surprisingly, the highest effective rate is

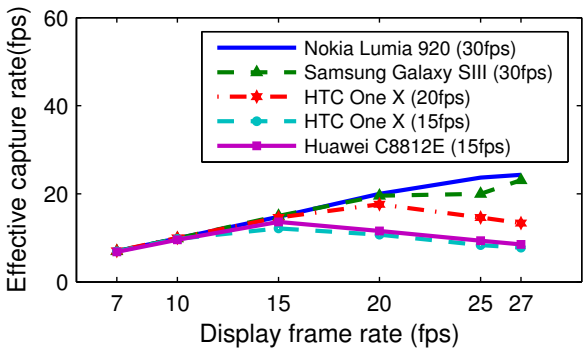


Figure 11: Effective capture rates on each phone.

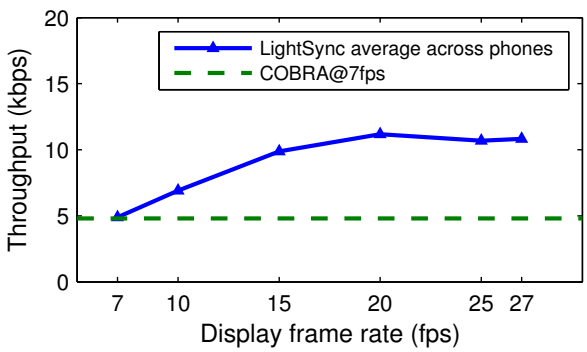


Figure 12: Average throughput across phones.

achieved when the display rate is comparable to the phone's actual frame rate. This is because per-frame decoding performance degrades and frame loss occurs for higher display rates. It takes extra frames and time to recover even a few undecoded blocks, due to either per-frame decoding failure or even a single lost frame. Note that we are interested in the shapes of the lines rather than the relative heights of them. The former shows how LightSync scales with increasing disparity between the display and capture frames rates, whereas the latter is determined by the hardware quality.

Alternative schemes. We next compare LightSync against COBRA and a simplified version of Unsynchronized 4D barcodes (4D barcodes for short). We omit PixNet in the comparison, since it was shown to be too computationally expensive for current smartphones[9].

COBRA is essentially our base code. The original proposal includes code optimizations based on color ordering and sender motion. We skip these since they do not matter in our setting. Note, however, that COBRA uses 4 colors, whereas LightSync is only in monochrome now, and therefore COBRA has twice the per-frame capacity as LightSync.

For this comparison, we assess the average rate across phones at the same display rate, emulating our broadcast scenario. The decoding algorithm in COBRA means it can only decode frames captured at a rate at least double that of the display rate. Therefore, when the lowest-quality phones can only capture at 15 fps, the video playback rate can be no higher than 7 fps. For a higher display rate, the low-fps phones capturing a COBRA-coded video will simply see a sequence of mixed frames and unable to decode much, unless the blocks at the same positions in consecutive frames happen to have the same color. This cannot be resolved simply by caching captured frames and decoding offline, and means that COBRA can only be run for the 7 fps screen rate for all phones. Figure 12 shows that the average throughput across phones LightSync can support is up to 2.3× that for COBRA. In terms of decoding time, the main additional decoding step in LightSync is undoing the inter-frame coding, which takes negligible time.

The basic idea of 4D barcode is to repeat frames on multiple, orthogonal color channels. Since the original design cannot support real-time processing[9], we adapt it to be based on COBRA like frame layout but encoding data in colored blocks in the original spirit. However, we observe that it cannot be decoded even when displayed at 7 fps.

4D barcode performs poorly for three reasons. First, red, green, and blue are not orthogonal in practice. Decoding is practically impossible when the patterns on different color channels interfere, and different color distortion behavior across phones makes it nearly impossible to correct for the colors generically. This could in theory be corrected in software if the colors are calibrated precisely per device. However, it is not practical for the type of scenarios we are targeting. The original design included color rectification and specific block patterns that could alleviate the color interference. Second, even when the colors can be differentiated, the frames can only be decoded if the display frame rate is no more than $\frac{3}{2}$ of the frame capture rate. The original paper evaluated their system with animated `.gif`, whose rate is 10 fps. At a higher display rate, we face the same issue with mixed frames as for COBRA, which cannot be decoded using the conventional approach. Third, the throughput of 4D barcode even in the ideal case is equivalent to using a monochrome channel at the same frame capture rate.

6. RELATED WORK

Visible Light Communications (VLC). Our work falls within Visible Light Communication. The technology was first explored by VLCC (Visible Light Communication Consortium) in Japan[13]. Since then, there have been many efforts exploring various hardware and techniques (e.g., [14, 8]). However, they generally involve LEDs as transmitters and photodiodes as receivers, and a significant focus is on modulation via pulses. [10] uses pixelated wireless optical channels, though. In comparison, we use general-purpose LCDs and smartphone cameras for communication. Furthermore, our encoding mechanism mainly involves operating over groups of pixels spatially, but the pixels can also take on different colors to encode bits.

Screen-camera links. As a specific type of VLC, LCD-camera communication tends to be one-way, involving showing a video to transmit information. We already referred to PixNet, COBRA, and 4D barcode. Building on building on [10], PixNet is designed for LCDs and high-end cameras, and proposes a high-throughput 2D barcode per frame modulated using 2D OFDM. COBRA is for phone-to-phone optical streaming. The authors design a color barcode with a layout optimized for low-complexity processing and mobility-aware adaptive code block arrangement to be blur-resilient. To draw an analogy with traditional RF propagation based wireless communications, PixNet is analogous to designing an OFDM-based modulation scheme over BPSK for individual frames, whereas COBRA optimizes the frame format while using a simpler bit-to-waveform mapping as the modulation scheme over the entire channel.

LightSync is orthogonal, in that we address frame synchronization and aim to increase the frame rate. The analo-

gous task in traditional wireless is to decode undersampled frames, e.g., when consecutive frames are sent on 20 MHz channels, but sampled at less than 40 MHz and received as a single frame. 4D barcode targets the same issue by extending a 2D barcode across time on orthogonal color channels. We have explained in Section 5 that the design is suboptimal.

VRCodes[18] follows a further orthogonal avenue by exploring unobtrusive barcode design that is imperceptible to human eyes. The authors leverage rolling shutter to mimic a high frequency change of selected colors, such that human eyes would only perceive the mixture of both colors at all times. The idea could be combined with LightSync, if we replace black and white with suitable colors.

Note that the encoding and decoding goals and mechanisms for communication over screen-camera pairs are substantially different from conventional video coding and computer vision work. Video coding and transmission aims to use a small number of bits to describe and convey the original scene information, in the hope of reconstructing the scenes with little or acceptable quality degradation. Vision work tries to interpret the captured images, which is an open-ended process. In contrast, all the screen-camera communication work modulates a bit stream to well-defined colored pixel blocks, which are then demodulated accordingly to retrieve the original bit stream. This is simply software-defined radio in the visible light frequency bands.

Visual interactions. There has long been interest in enabling visual interactions with phone cameras[16, 17], including to assist local communication, e.g., by obtaining a security token with the camera. Although we share the same ultimate goal of enriching visual interaction, we aim to enhance the visual communication capability.

There are many 2D barcodes, such as QR code or data matrix. These codes can be printed on any surface, but is static. In comparison, our scenario requires a screen, but can display dynamic content.

Optical Character Recognition (OCR) could be another alternative to 2D barcodes, although current OCR technologies can only recognize simple patterns.

Short-range RF wireless technologies. Latest smartphones typically support Bluetooth and Near Field Communication (NFC), and can communicate with another device fitted with the needed chips. These can power such applications as file transfer and contact-less payment.

We view screen-camera based visual communication as complementary to NFC, because we are adding communication capability to devices that may not be NFC-capable. Visual links can sometimes be more efficient than NFC due to non-interference with similar wireless devices.

7. CONCLUSION

In this work, we show how to achieve unsynchronized communication in real time between LCDs and phone cameras with LightSync. Our techniques mainly include in-frame per-line color tracking to decode mixed frames and inter-frame erasure coding to recover lost frames. Our implementation works with off-the-shelf screens and smartphones without any OS or firmware changes. It also work with other 'auto' settings of the smartphone camera as much as possible, to allow the user to seamlessly switch between normal camera usage and such visual communication. The frame rate can also adjust automatically based on the average intensity of each frame without breaking the system.

With LightSync, we are able to decode information even when the display frame rate is higher than the camera frame capture rate. This shifts the performance bottleneck to the individual receiving camera's hardware capability, instead of the display frame rates. As a result, when the same content is displayed to a range of phones, the average throughput across phones can more than double and is not constrained by the least capable phone, unlike in previous approaches. Techniques like LightSync provide primitives for more sophisticated visual communication scenarios.

Acknowledgments

We are very grateful to Heran Lin for helping with the experiments during the revision of the paper and to the anonymous reviewers for very insightful comments that helped improve the paper.

8. REFERENCES

[1] Gibraltar targets tourists with Wikipedia QR codes. http://www.bbc.co.uk/news/technology-19544299.

[2] HTC One X specification. http://www.gsmarena.com/htc_one_x-4320.php.

[3] Huawei Ascend C8812E specification. http://pdadb.net/index.php?m=specs&id=3532&c=huawei_ascend_c8812.

[4] Mary Meeker's 2013 Internet Trends Report. http://allthingsd.com/20130529/mary-meekers-internet-trends-report-is-back-at-d11-slides/.

[5] Nokia Lumia 920 specification. http://www.gsmarena.com/nokia_lumia_920-4967.php.

[6] QR codes improve shopping experience. http://www.iogear.com/blog/2011/03/15/qr-codes-improve-shopping-experience/.

[7] Samsung Galaxy S III specification. http://www.phonearena.com/phones/Samsung-Galaxy-S-III_id6330.

[8] A. Ashok, M. Gruteser, N. Mandayam, J. Silva, M. Varga, and K. Dana. Challenge: Mobile optical networks through visual MIMO. In *Proceedings of MobiCom*, 2010.

[9] T. Hao, R. Zhou, and G. Xing. COBRA: Color Barcode Streaming for Smartphone Systems. In *Proceedings of MobiSys*, 2012.

[10] S. Hranilovic and F. R. Kschischang. A Pixelated MIMO Wireless Optical Communication System. *IEEE Journal of Selected Topics in Quantum Electronics*, 2006.

[11] K. Jamieson and H. Balakrishnan. PPR: Partial packet recovery for wireless networks. In *ACM SIGCOMM*, 2007.

[12] T. Langlotz and O. Bimber. Unsynchronized 4D Barcodes Coding and Decoding Time-Multiplexed 2D Colorcodes. In *Proceedings of the International Symposium on Visual Computing*, 2007.

[13] M. Nakagawa. The world of visible optical communication - It opens with LED — Light Communications. *Kogyo Chosakai Publishing, Inc.*

[14] G. Pang, T. Kwan, H. Liu, and C. Chan. LED Wireless. *IEEE Industry Applications Magazine*, 2002.

[15] S. D. Perli, N. Ahmed, and D. Katabi. PixNet: LCD-camera pairs as communication links. In *Proceedings of MobiCom*, 2010.

[16] M. Rohs. Real-world Interaction with Camera-Phones. In *Proceedings of the International Symposium on Ubiquitous Computing Systems*, 2004.

[17] P. Vartiainen, S. Chande, and K. Rämö. Mobile Visual Interaction: Enhancing Local communication and Collaboration with Visual Interaction. In *Proceedings of the International Conference on Mobile and Ubiquitous Multimedia*, 2006.

[18] G. Woo, A. Lippman, and R. Raskar. VRCodes: Unobtrusive and Active Visual Codes for Interaction by Exploiting Rolling Shutter. In *IEEE International Symposium on Mixed and Augmented Reality*, 2012.