# SphericRTC: A System for Content-Adaptive Real-Time 360-Degree Video Communication

Shuoqian Wang
SUNY Binghamton
swang130@binghamton.edu

Xiaoyang Zhang
SUNY Binghamton
xzhan211@binghamton.edu

Mengbai Xiao
The Ohio State University
xiao.736@osu.edu

Kenneth Chiu
SUNY Binghamton
kchiu@binghamton.edu

Yao Liu
SUNY Binghamton
yaoliu@binghamton.edu

## ABSTRACT

We present the SphericRTC system for real-time 360-degree video communication. 360-degree video allows the viewer to observe the environment in any direction from the camera location. This more-immersive streaming experience allows users to more-efficiently exchange information and can be beneficial in the real-time setting. Our system applies a novel approach to select representations of 360-degree frames to allow efficient, content-adaptive delivery. The system performs joint content and bitrate adaptation in real-time by offloading expensive transformation operations to the GPU via CUDA. The system demonstrates that the multiple sub-components – viewport feedback, representation selection, and joint content and bitrate adaptation – can be effectively integrated within a single framework. Compared to a baseline implementation, views in SphericRTC have consistently higher visual quality. The median Viewport-PSNR of such views is 2.25 dB higher than views in the baseline system.

## CCS CONCEPTS

• **Information systems → Multimedia streaming**; **Web conferencing**; • **Human-centered computing → Virtual reality**.

## KEYWORDS

real-time communication; WebRTC; 360-degree video; content-adaptive; CUDA

**ACM Reference Format:**
Shuoqian Wang, Xiaoyang Zhang, Mengbai Xiao, Kenneth Chiu, and Yao Liu. 2020. SphericRTC: A System for Content-Adaptive Real-Time 360-Degree Video Communication. In *Proceedings of the 28th ACM International Conference on Multimedia (MM '20), October 12–16, 2020, Seattle, WA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3394171.3413999

## 1 INTRODUCTION

360-degree videos allow users to navigate recorded scenes in three degrees of freedom (3-DoF). The 360-degree video medium is not only useful for delivering pre-recorded content, but can also allow users to more-naturally experience real-time events, e.g., teleconferencing, telepresence, and human-controlled robotic navigation.

A core challenge in 360-degree video delivery involves delivering the frames required for immersive experience in a bandwidth-efficient way. Current implementations can suffer from both low quality user-views and high bandwidth consumption. Both drawbacks can be mitigated by reducing the number of transmitted, but unviewed, pixels on 360-degree frames.

A wide range of approaches have been proposed to address wasted bandwidth in the 360-degree video streaming setting. Tiling approaches, e.g., [23, 38], transmit only a subset of the spatial portions of the omnidirectional frame that will be observed by users. Oriented projections [6, 8, 50] attempt to transmit representations with higher pixel densities in regions likely to be viewed by users. Both tiling and oriented projection approaches require accurate predictions of the users' view-orientation to operate effectively. In the on-demand streaming setting, many such prediction approaches have been proposed [23, 36, 38, 44, 45].

Bandwidth-efficient operation in the real-time 360-degree video communication setting is significantly more challenging than in the on-demand streaming settings. In the real-time setting, much less time is available to convert collected 360-degree segments to more efficient tile-based or oriented projection segments. In addition, given that new content is generated on-the-fly, traces of view orientations from many users over a single sequence of video frames cannot be used to predict the current user's view.

This paper describes a practical end-to-end real-time 360-degree video communication system, which we call SphericRTC. SphericRTC applies a combination of computational tools and algorithms to achieve bandwidth-efficient streaming in the real-time setting. SphericRTC achieves bandwidth efficiency and improves view quality by processing raw segments into content-adaptive oriented projections. To perform this adaptation step in real-time, SphericRTC offloads these transformations to the GPU. Specifically, this work makes the following contributions:

- We design an oriented projection approach toward 360-degree content adaptation suitable for real-time processing. This approach includes a novel strategy for selecting the best oriented projection parameters for a given real-time adaptation bitrate.
- We show that, in our system, the simple approach of using the most-recent reported user orientations is effective for generating frame representations for high-quality views. This approach is


*MM '20, October 12–16, 2020, Seattle, WA, USA*

ACM ISBN 978-1-4503-7988-5/20/10…$15.00
https://doi.org/10.1145/3394171.3413999

effective due to the low latency between reporting user's view orientation and consumption of the generated frames.
- We implement a CUDA library to create adaptive representations in real-time and integrate this library into our SphericRTC framework.
- Our complete system integrates viewport feedback, adaptation and transformation components. This integration demonstrates that managing information flow from video receiver to the video collector and between prediction and adaption components are possible in a practical system.
- Results from extensive experiments show that SphericRTC consistently outperforms a baseline system: The median Viewport-PSNR of views rendered in SphericRTC is 2.25 dB higher than the views rendered in the baseline system.

## 2 BACKGROUND

### 2.1 360-Degree Video

360-degree videos present users with omnidirectional content, allowing them to freely explore all orientations emanating from the camera position. To effectively transmit 360-degree videos, the omnidirectional content is first projected onto a 2D plane. While many projection schemes exist, the ***equirectangular projection*** [11] is the one that is the most simple and widely used. Transmitting 360-degree video streams typically include full omnidirectional frames. However, although all pixels in the omnidirectional frame would be transmitted, only the portion in the field of view (FoV) is rendered and displayed to the viewer. This results in a large number of wasted (i.e., unviewed) pixels. 360-degree video viewing devices' FoVs are typically between 90° and 120°. To illustrate the amount of wasted bandwidth, rendering a 100°×100° FoV requires only approximately 15% of pixels on an equirectangular frame. A direct consequence of wasted bandwidth is bad visual quality of the rendered views. For example, suppose the available network bandwidth can support streaming of an equirectangular-projected video of 1080p, the effective resolution of rendered views is only equivalent to 480p or even 360p.

To address the bandwidth inefficiency, a number of solutions have been proposed to reduce the wasted data in the on-demand and live video streaming settings. These approaches include tiling [21–23, 28, 33, 35, 37, 38, 43, 44, 49] and oriented projections [6, 8, 50]. The high-level idea is to prepare representations of the 360-degree videos such that it is possible to only transmit data within predicted viewport in high quality.

### 2.2 Real-Time Video Communication

Real-time video communication is widely used for online video chatting and video conferencing. Services such as Apple's FaceTime, Google Hangouts, and Zoom allow video streams to be transmitted with hundreds of milliseconds round-trip time [46, 48]. Developers can also leverage the open-source WebRTC framework [13] to add real-time video communication capabilities to their applications on various platforms.

Real-time video communication, however, should be differentiated from the term "live video streaming". In "live streaming", broadcaster-to-viewer latency is on the order of tens of seconds [31, 47]. For example, Liu et al. found that the median broadcaster-to-viewer latency of 360-degree live streaming is 37.1 seconds on YouTube and 18.7 seconds on Facebook [31]. With less stringent latency requirements, live video streaming can take advantage of per-segment encoding (batching a few seconds of video before encoding) and reliable TCP-based transmission. On the other hand, for real-time video communication, captured video frames should be encoded and transmitted as soon as possible without batching.

A key challenge with real-time video communication is the ultra-low latency required by interactive video, which helps create the illusion that the video receiver is located in the same place as the video collector. To do so, collected video frames are encoded and then transmitted via the real-time transport protocol (RTP) [39] typically over the UDP transport layer protocol.

### 2.3 WebRTC

WebRTC is an open-source framework for real-time video communication [13]. When transmitting videos over WebRTC, the video collector encodes frames in RTP packets. The video receiver records information about every RTP packet it receives and transmits the information in a real-time transport control protocol (RTCP) message (Transport-wide Feedback) back to the video collector. The ***bandwidth controller*** at the video collector calculates metrics such as inter-packet delay variation, queuing delay, and packet loss. These metrics are then used to compute the ***target sending bitrate*** for transmitting video content to the video receiver [3, 5]. The frame encoder then uses this target sending bitrate to adjust the ***quantization parameter*** (*qp*). This *qp* parameter is further used by the bandwidth controller. The bandwidth controller adjusts ***target resolution*** of captured frames so that transmitted frames more-closely match the target sending bitrate. To dynamically adapt the video's sending bitrate based on the network conditions, the bandwidth controller adjusts the <*estimated bandwidth/target sending bitrate, target resolution*> of transmitted video streams based on network statistics collected from the video receiver. Separate from the main audio/video stream, WebRTC can also be used to set up a data channel using the stream control transmission protocol (SCTP) [4].

Vanilla WebRTC does not provide special support for 360-degree videos. To provide real-time communication, 360-degree videos have to be treated the same way as traditional 2D videos. For example, consider a 360-degree camera connected to a video collector computer via a wired connection, the camera outputs video frames in the equirectangular projection. These equirectangular-projected frames will be encoded and transmitted by WebRTC via the secure real-time transport protocol (SRTP) [2]. During transmission, target sending bitrate of the video collector/sender will be adapted based on the delay and packet loss feedback from the receiver [17].

A main drawback with this solution, however, is that a large percentage of pixels on the equirectangular-projected frame is not viewed, causing wasted bandwidth and low quality of rendered views. In this paper, we propose SphericRTC that achieves bandwidth efficiency and improves view quality by using content-adaptive oriented projections. We compare the performance of SphericRTC with the vanilla WebRTC baseline in evaluation.

## 3 DESIGN

### 3.1 Overview of SphericRTC

The core idea in SphericRTC's design involves transforming the video content in a way that reduces unviewed pixels. In the real-time setting, this transformation must take place within a tolerable frame delivery threshold. SphericRTC performs this content adaptation using i) the most recently reported view-orientation and ii) a bitrate adaptation decision derived from network statistics.

Figure 1 provides an overview of SphericRTC components. Internally, SphericRTC is built on top of WebRTC. With vanilla WebRTC, the video collector transmits video stream to the video receiver, and the video receiver reports network statistics back to the video collector to perform bitrate adaptation. To enable content adaptation, in SphericRTC, the video receiver reports back the additional "**Viewport Feedback**", i.e., the user's view orientations while consuming the 360-degree video stream, to the video collector.

In the "**Representation Selection**" component, content-bitrate adaptation decisions are made based on collected network statistics and users' viewing data. For content adaptation, SphericRTC uses ***oriented projections*** for representing 360-degree video frames. Oriented projections devote more pixels in the projected frame to areas on the sphere close to a target pixel-concentration orientation. Decisions made by the "Representation Selection" component are forwarded to the "**Frame Processing**" component, which spatially transforms the camera-captured frame into representations with different pixel-concentration areas and frame resolutions.

In the remainder of this section, we first discuss how the "Representation Selection" component makes adaptation decisions by choosing appropriate oriented projection parameters. We then discuss both how to synchronize selected per-frame projection parameters with the video receiver as well as a variety of other design considerations.

### 3.2 Analysis of Oriented Projections

An "oriented projection" uses more pixels on the video frame to represent a selected target pixel-concentration area than other areas in the frame [50]. After transformation to an oriented projection, the frame will appear as if it "magnifies" the content near the pixel-concentration direction. When the user's view is near this target orientation, views rendered from oriented frames will be of higher quality compared to views from a standard projection of the same resolution. However, views farther away from the target orientation can be in lower quality compared to views from a comparable standard projection. We use the ***offset spherical projection*** to create oriented projections in SphericRTC.

*3.2.1 Offset Spherical Projection.* The offset spherical projection can be parameterized by a single offset vector $\vec{d}$, $\|\vec{d}\| \in [0, 1)$. This offset vector describes the oriented projection's pixel-concentration direction. An offset projection can be created starting with pixels from a 360-degree video frame are mapped to the surface of a unit sphere. Each pixel can be represented by a unit vector $\vec{p}$ (i.e., $\|\vec{p}\| = 1$) from the center of the sphere to the pixel position. All pixels on the sphere are "offset" by $\vec{d}$ by applying the following transformation: $\vec{p'} = \frac{\vec{p}-\vec{d}}{\|\vec{p}-\vec{d}\|}$, creating a new (normalized) vector, $\vec{p'}$. Figure 2 illustrates how the offset vector $\vec{d}$ is applied to pixels on a sphere.

As pixels are re-projected to new locations on the sphere, we obtain a new sphere. We can then project pixels on the new sphere to a 2D plane, e.g., using the equirectangular projection, for 2D video encoding. Figure 3(a) shows an original equirectangular-projected frame with no offset applied. Figures 3(b), 3(c), and 3(d) show the resulting offset equirectangular projections after applying offset vectors with different magnitudes.

*3.2.2 Magnification Analysis.* By applying the offset, content on the sphere near the offset direction will be magnified. In Figure 2, red-tinted pixels represent magnified areas of the sphere while content in the blue-tinted pixels indicate areas of the sphere that were reduced to occupy smaller areas.

If we assume that the angle between a pixel vector $\vec{p}$ on the sphere and the offset vector $\vec{d}$ is $\theta$, then after applying the offset, the angle between the new vector $\vec{p'}$ and the offset vector becomes:

$$f(\theta) = arctan\frac{\sin\theta}{-\|\vec{d}\| + \cos\theta} \tag{1}$$

Consider a small FoV of $2 \cdot \Delta$, the magnified FoV will become $2 \cdot \Delta' = 2 \cdot (f(\theta + \Delta) - f(\theta))$. We can thus calculate the magnifying value as:

$$\frac{\Delta'}{\Delta} = \frac{f(\theta + \Delta) - f(\theta)}{f(\Delta) - f(0)} \tag{2}$$

Per-pixel magnification can be thus approached by (3), which can be calculated as the derivative of $f(\theta)$ as (4):

$$\lim_{\Delta \to 0} \frac{f(\theta + \Delta) - f(\theta)}{\Delta} \tag{3}$$

$$g(\theta) = f'(\theta) = \frac{\frac{\cos\theta}{-\|\vec{d}\|+\cos\theta} + (\frac{\sin\theta}{-\|\vec{d}\|+\cos\theta})^2}{1 + (\frac{\sin\theta}{-\|\vec{d}\|+\cos\theta})^2} \tag{4}$$

In this function, the greatest value is at $\theta = 0$, i.e., when the pixel is in the same direction as the offset vector $\vec{d}$, and the maximum magnification can be calculated as:

$$g(0) = \frac{1}{-\|\vec{d}\| + 1} \tag{5}$$

### 3.3 Joint Content and Bitrate Adaptation

Using offset projections for content adaptation requires us to determine the best offset vector, $\vec{d}$, for transforming the 360-degree content. Furthermore, we can represent $\vec{d}$ as the product of a unit vector, $\vec{o}$ (the offset direction), pointing from the center of the sphere to a point on the spherical surface, and a scalar, $m \in [0, 1)$ (the offset magnitude), representing the magnitude of the offset vector. That is, $\vec{d} = m \cdot \vec{o}$. The content adaptation task then becomes two tasks: selection of offset direction and selection of offset magnitude.

*3.3.1 Offset Direction Determination.* Content magnification is maximal in the pixel-concentration direction pointed by the offset vector. Therefore, to maximize the visual quality of views rendered at the video receiver's side, the *offset direction* should be selected to match the viewer's future view orientation. Typically, a prediction algorithm would be used to predict the future orientation. Here, real-time video communication allows us to take advantage of the

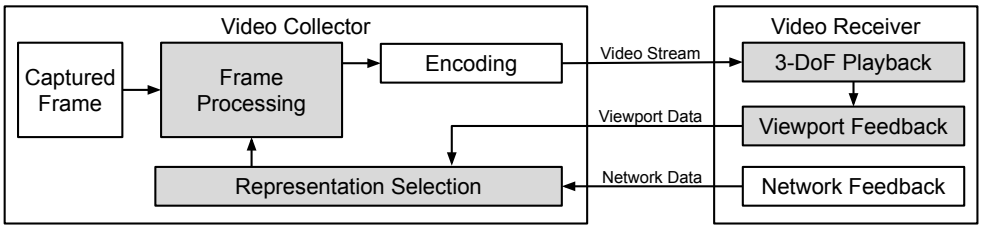



**Figure 1: High-level design of SphericRTC. The grey-shaded boxes show components added or modified by SphericRTC from the original WebRTC implementation.**

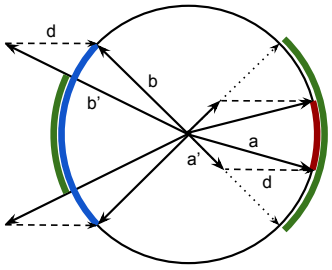
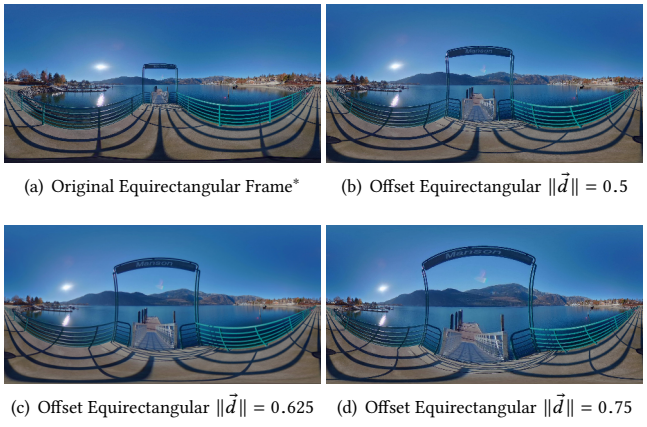



**Figure 2: Illustration of the Offset Spherical Projection**



(a) Original Equirectangular Frame* (b) Offset Equirectangular $\|\vec{d}\| = 0.5$

(c) Offset Equirectangular $\|\vec{d}\| = 0.625$ (d) Offset Equirectangular $\|\vec{d}\| = 0.75$

**Figure 3: By applying offsets with different magnitudes, contents near the frame center are magnified to different levels. *Photo by Timothy Oldfield on Unsplash: https://unsplash.com/photos/luufnHoChRU.**

ultra-low end-to-end latency. Given that the view orientation will not change too drastically during a small time period, we propose a simple approach that predicts view orientation of a future frame as the most recent received view orientation from the video receiver. This predicted view orientation is then used as the offset direction for adapting the frame content.

*3.3.2 Offset Magnitude Determination.* To select the best *offset magnitude*, we have to take into account the bitrate adaptation decision made by the WebRTC bandwidth controller. The bandwidth controller adjusts the *target resolution* according to the quantization parameter *qp* returned from encoder, and *qp* is adjusted based on the content to be encoded and the *target bitrate* selected by the bandwidth controller. Our content adaptation selects an appropriate offset magnitude whenever the target resolution changes. This offset magnitude is chosen such that the pixel-concentration area's pixel density matches the pixel density of the camera-supplied frame. That is, our approach fixes the pixel density in the concentration area and degrades densities away from the concentration area if the bandwidth controller changes the target resolution.

We adaptively select offset magnitude based on the following quantities: (1) the resolution of the captured frame, i.e., the frame provided by the 360-degree camera, and (2) the target resolution selected by the bandwidth controller. If the estimated bandwidth allows frames in the original resolution to be transmitted, then no content magnification is permitted, and the offset magnitude should be set to 0. We thus use the following equation to determine the offset magnitude:

$$m = \|\vec{d}\| = 1 - \frac{1}{g(0)} = 1 - \max(\frac{w'}{w}, \frac{h'}{h}) \quad (6)$$

Here, $w'$ and $h'$ represent the width and height of the target resolution selected by WebRTC's bandwidth controller. $w$ and $h$ represent the width and height of the original resolution of frame provided by the video collector's 360-degree camera. Note that this equation assumes that the original, camera-supplied frame and the adapted frame use the same underlying base spherical projection, e.g., the equirectangular projection. If not, $w$ and $h$ need to be adjusted accordingly.

To summarize, the "Representation Selection" component in SphericRTC makes adaptation decisions based on both the predicted view orientation (for using as the offset direction) and the target resolution selected by WebRTC's bandwidth controller (for selecting the offset magnitude). This dependence of offset magnitude on target bitrate is a form of joint adaptation.

## 3.4 Transmitting Offset Direction to the Receiver

As the video collector makes decisions regarding the bitrate adaptation and the content adaptation, the video receiver must also understand how to correctly render received frames in the offset projections. While the offset magnitude can be inferred from the received frame's resolution, one must know the offset direction used for creating the offset projection in order to correctly render views at the video receiver side.

Unlike on-demand and live streaming where video receivers request video segments that are pre-encoded and can contain one or a few seconds of video content, in real-time communication, frames are encoded and sent frame-by-frame to the receiver. As a result, out-of-band transmission of metadata, e.g., the media presentation description (MPD) document used in on-demand and live streaming, is not feasible in real-time scenario.

Instead, we choose an in-band solution, sending the offset direction information with each frame. In this way, the video receiver can extract required information needed for view rendering from the received frame directly. We will discuss our implementation leveraging RTP header extension in Section 4.3.

### 3.5 Other Considerations

Once the "Representation Selection" component has determined the parameters of the offset projection to encode, the "Frame Processing" component will apply the transformation to the frame captured from the 360-degree camera accordingly. SphericRTC uses the equirectangular projection as the underlying mapping of spherical pixels to the 2D frame. This design choice is based on the following reasons:

First, many 360-degree cameras natively output the equirectangular frames. Thus, we can represent the original, camera-supplied frame as a special case of the offset equirectangular projection, i.e., with the offset magnitude being 0. In this way, no frame transformation is needed if the network bandwidth is sufficient for transmitting the frame in its camera-supplied resolution.

Second, if we use other spherical projections such as the cubemap projection, the projected content on the 2D rectangular plane may no longer be continuous. Content discontinuity may negatively affect the quality of encoded pixels under a same target encoding bitrate. For example, the cubemap projection projects pixels on a sphere onto six faces of a cube. However, even with the "baseball layout" [20], the boundary between top and bottom portions of the projection is not continuous. Discontinuous content makes it more difficult for the encoder to find similar blocks within the frame to copy from, and thus requires more bits to encode to a same quality.

## 4 IMPLEMENTATION

We implement SphericRTC based on the open-source framework, WebRTC. Figure 4 shows an overview of modules added or modified in the SphericRTC implementation. Three new modules are added: (1) the "View Orientation Transmitter" module at the video receiver side provides viewport data feedback to the video collector, (2) the "Offset Direction Determination" module chooses the best offset direction based on feedback from the video receiver, and (3) the "FastTransform360" library performs fast frame transformation.

In addition, to integrate these three new modules with the WebRTC code base, we further modified the source code of a number of objects in WebRTC. These modified WebRTC objects are drawn in dashed boxes in Figure 4.

Overall, the view orientation transmitter is written in about 100 lines of code in Javascript, the offset direction determination module is written in about 200 lines of code in C, the FastTransform360 library is written in about 1,000 lines of CUDA code and 600 lines of code in C++ as an interface. Moreover, about 600 lines of code in C++ is added to the WebRTC code base[1].

### 4.1 View Orientation Transmission and Offset Direction Determination

To select the best offset direction for content adaptation, the video collector needs to know where the viewer at the video receiver side is going to look at. To do so, the video receiver sends back user's view orientation data to the video collector and let the offset direction determination module perform the prediction.

[1]We make SphericRTC source code available at: https://github.com/bingsyslab/SphericRTC

To reliably transmit user's view orientation data to the video collector, we use the WebRTC data channel [4] which is implemented using the `usrsctp` library and transmitted over SCTP. At the video collector, the offset direction determination module is implemented as a shared-memory server using the POSIX shared memory APIs [7]. It receives data sent from the video receiver, performs view prediction, and writes the latest prediction to the shared memory space. In this work, we simply predict future views as the last-received view orientation. However, a more complex view prediction scheme can be easily incorporated into our shared-memory server. When SphericRTC needs the offset direction, it fetches from the piece of memory. With the producer-consumer modeled shared-memory server, the video collector can always get the latest predicted offset direction.

### 4.2 The FastTransform360 Library (FT360)

Facebook has open-sourced a 360-degree video frame processing library, Transform360 [9], that uses the CPU for its transformation operations. However, we found that CPU-based frame transformation cannot satisfy the real-time latency and throughput requirements.

To this end, we implement a FastTransform360 (FT360) library to perform fast frame processing based on joint content and bitrate adaptation decisions. FT360 is compiled as a shared library using CUDA [10] with a C++ interface. It takes frame inputs in YUVI420 byte array format, computes a mapping table, performs sampling using the CUDA sampling module, `tex2D()`, and outputs the transformed frame in YUVI420 format. The FT360 library can perform scaling, rotation, offset, and/or re-projection from input equirectangular-projected frames to desired frames in various base spherical projections.

### 4.3 RTP Header Extension

To provide in-band transmission of offset direction information to the video receiver, we use RTP header extension to include the information and send it to the video receiver with the frame simultaneously. To transmit frames over the network, encoded frames are split into multiple RTP packets and transmitted via UDP. To reconstruct the frame at the video receiver side, each RTP packet includes a header that includes a timestamp as the frame ID. Furthermore, RTP also allows a header extension mechanism that can carry additional information in the RTP header. We thus take advantage of this and add 5 bytes of add-on extension. Figure 5 shows the new RTP header generated by SphericRTC. The add-on extension is used for encoding the offset direction information into four bytes (*uint*8): yaw, pitch, roll, and extra. Both "pitch" and "roll" have 180 unique integer values and can be represented using one *uint*8, while "yaw" has 360 unique integer values, and we use an extra *uint*8 for it. Overall, the total number of offset directions allowed in our SphericRTC is $360 \times 180 \times 180 = 11,664,000$.

## 5 EVALUATION

We conducted extensive experiments comparing SphericRTC with the naive implementation of real-time 360-degree video communication via vanilla WebRTC. We focus our performance evaluation

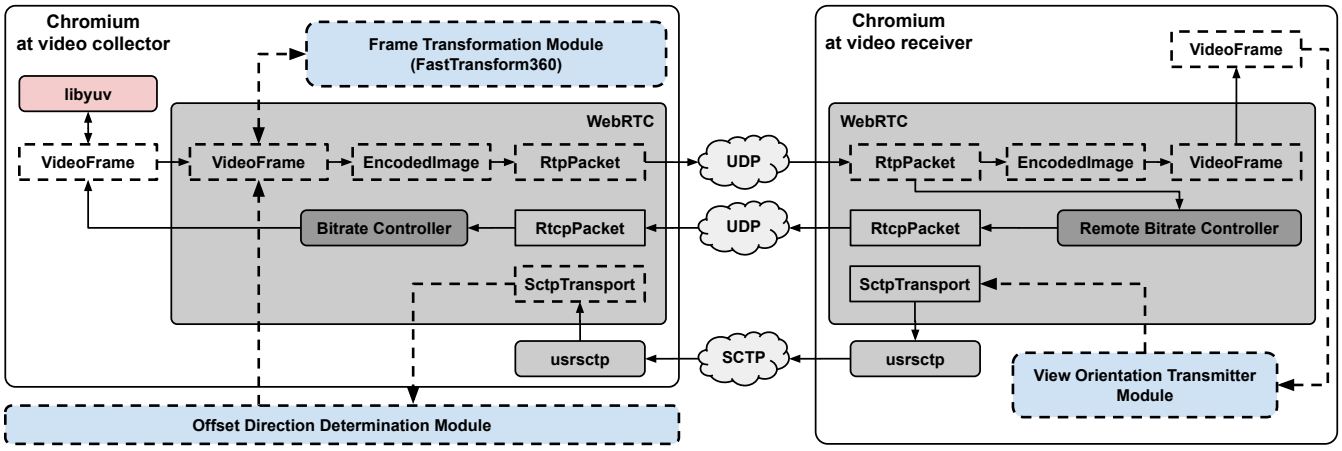



**Figure 4: Overview of SphericRTC implementation. Modules in dashed boxes are added or modified by SphericRTC. The `libyuv` library is used in vanilla WebRTC for frame scaling. With SphericRTC, frame scaling and transformation are all performed via FastTransform360 instead.**

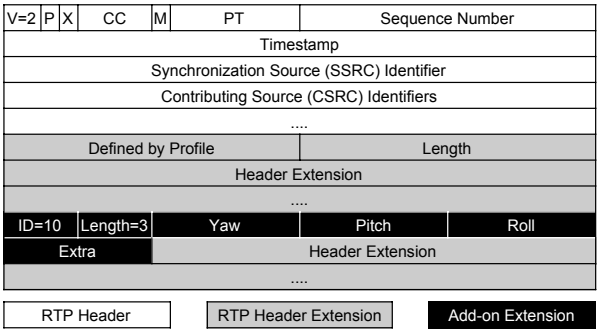



**Figure 5: To support in-band transmission of offset directions, we added 5 bytes to the RTP header.**

on if SphericRTC can satisfy the real-time communication requirements, and if the content adaptation provided by SphericRTC can indeed improve the visual quality of views rendered to the end users.

## 5.1 Evaluation Methodology

*5.1.1 Dataset.* To conduct repeatable experiments for fair comparison between SphericRTC and vanilla WebRTC, we used 360-degree videos and user head movement traces from two publicly-available datasets [19, 42]. For each dataset, we used 5 videos for experiments, and for each video, 50 seconds of video content and 20 users' head movement traces are used, creating in total 10,000 seconds of videos for each test setting. These videos are pre-decoded and saved in `.y4m` [15] format – a byte array format. The `.y4m` file can be read as a virtual camera by the Chromium browser, simulating the frame capture of a physical camera [14].

*5.1.2 Workflow.* The workflows for SphericRTC and vanilla WebRTC are similar. We first establish a peer-to-peer WebRTC connection between the video collector and the video receiver [12]. Once the WebRTC connection is established, the captured frames will be transformed and encoded at the collector side. Here, we record all the captured frames in YUVI420 format before encoding, which is the last place we can keep frames lossless. Encoded frames are then encrypted, split into RTP packets, and sent to the receiver. At the receiver side, the RTP packets are reassembled, decrypted, and passed to the video decoder. During transmission, some of the frames may not be received due to packet loss or may be dropped due to network delay (e.g., the frame arrives too late). All the frames are also recorded in YUVI420 format after decoding, for performance evaluation. Afterwards, frames are rendered by WebGL shaders implemented in OpenGL Shading Language (GLSL).

For SphericRTC, additional steps are needed. The offset direction determination module at the video collector passes the latest received view-orientation as the predicted offset direction. In the mean time, it also calculates the offset magnitude based on selected target resolution via joint content and bitrate adaptation. The FastTransform360 library then transforms frames to desired offset direction, offset magnitude, and target resolution, and passes them to the encoder. The video receiver extracts the offset direction from the RTP packets and infers the offset magnitude from the resolution of received frames.

*5.1.3 Signaling Server.* To help establishing the peer-to-peer connection between the video collector and the video receiver, we use a third-party signaling server [12]. In our experiments, we also use this signaling server to store user head movement traces. In this way, the video receiver can fetch the trace file from the third-party server and send the user's trace data to the collector in a desired interval. While we selected the interval of 500 ms in our experiments, this interval can be adjusted depending on the network latency, frame content, and user behavior.

*5.1.4 Frame Matching and Rendering.* Frames recorded at the video collector side are used to generate the "ground truth views", and frames recorded at the video receiver side are used to generate the "receiver-side rendered views". To compare "receiver-side rendered views" with "ground truth views", we must first match frames recorded at both sides. To do so, we utilized timestamps inside the

Chromium browser and SphericRTC/WebRTC. From when a frame is captured to when it is packetized in RTP packets, there exist four levels of timestamps identifying a frame. The RTP timestamp is the only timestamp shared by both sides. To address this issue, all the timestamps are stored at the video collector side as three translation tables. At the video receiver side, while each decoded frame is only associated with its RTP timestamp, by using the translation tables, we can find matches between receiver-side decoded frames with corresponding captured frames.

We rendered views from each pair of matching frames and calculated the visual quality metrics between receiver-observed views and ground truth views. To render views based on user's view orientation, we used an open-source tool, FFmpeg360 [32], for OpenGL-based view rendering. We implemented new shaders for rendering frames in the offset equirectangular projection. We report visual quality results in two objective metrics: viewport peak signal-to-noise ratio (Viewport-PSNR) and viewport structural similarity [41] (Viewport-SSIM).

## 5.2 Experiment Setup

To evaluate SphericRTC's performance under different network scenarios, we used `tc` [1] for adjusting the network bandwidth, latency, and packet loss rate in different settings. Specifically, for performance under different network bandwidth, we experimented with 3 test settings: {500 Kbps, 1 Mbps, and no constraint}. Note that vanilla WebRTC consumes approximately 2.5 Mbps bandwidth when there is no bandwidth constraint. For latency evaluations, we experimented with the following 3 settings: {additional 500 ms one-way network delay, additional 200 ms one-way, and no additional delay}. Note that with our experiment setup, if no additional delay is added, the end-to-end delay between the video collector and the receiver, is approximately 200 ms one-way. This 200 ms one-way delay includes frame processing time, frame encoding time, network transmission delay, frame decoding time, and frame recording times which are required by our evaluation methodology. Finally, we also consider 3 packet loss rate settings: {fixed 5% loss, fixed 2% loss, and nature packet loss}.

## 5.3 Frame Processing Time

We first evaluate if FastTransform360 can perform frame transformation with low latency and high throughput. We compare the per-frame processing time of FastTransform360 with the open-source tool Transform360 on a same machine with an Intel i7-7700K CPU without overclocking and an NVIDIA GeForce GTX 1080 GPU. In the experiments, both FastTransform360 (integrated with SphericRTC) and Transform360 were used for converting input 3840×1920 equirectangular-projected frames to output frames in a fixed resolution of 2880 × 1440 in the offset equirectangular projection. Figure 6 shows the per-frame processing time logged in our experiments. Results show that the median frame processing time of CUDA-based FastTransform360 library is 10.099 ms, reaching 99 frames per second (FPS) processing throughput. On the other hand, the median processing time of CPU-based Transform360 is 60.951 ms.

## 5.4 Viewport Quality

For view quality comparison, we used frames recorded at both the video collector and the video receiver sides and users' head

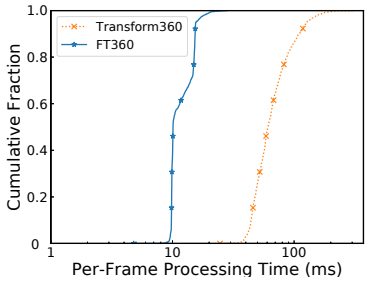



**Figure 6: FastTransform360 runs significantly faster than Transform360. Note that the x-axis is drawn in log scale.**

movement traces to render viewports with 100°×100° FoV. Here, we consider views rendered from frames recorded at the video collector as "ground truth views" since these frames are not encoded or transformed and are recorded in the lossless manner. We compare "receiver-side rendered views" with "ground truth views" and report the distribution of Viewport-PSNR and Viewport-SSIM.

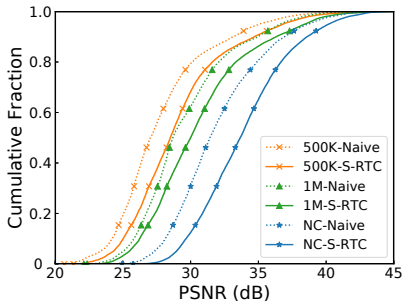
Figures 7(a) and 7(b) show the viewport quality comparison of various bandwidth settings. In these figures, SphericRTC results are denoted as "S-RTC". Results show that SphericRTC consistently outperforms naive WebRTC-based implementation (i.e., vanilla WebRTC). When there is no network constraint, the median Viewport-PSNR obtained by SphericRTC is 33.66 dB, 2.25 dB higher than the naive solution. The median Viewport-SSIM obtained by SphericRTC is 0.915, while the median Viewport-SSIM obtained by vanilla WebRTC is only 0.890. With 500 Kbps and 1 Mbps bandwidth constraints, SphericRTC still renders views of higher quality, achieving 1.46 dB and 1.31 dB improvement in median Viewport-PSNR, respectively.

When there is no bandwidth constraint but additional network delay, our results show that the viewport quality decreases. This is because WebRTC's bandwidth controller reduces the target sending bitrate in response to long network delay. Increased network latency also means it takes longer for receiver-side viewport feedback to arrive. As a result, the actual view orientation may be more deviated from the offset orientation used for representing the frame. We expect using a more complex view prediction scheme can alleviate this problem. Even reduced target sending bitrate and less accurate view prediction, results show that SphericRTC still consistently performs better than the naive WebRTC implementation. Detailed results are show in Figures 8(a) and 8(b).

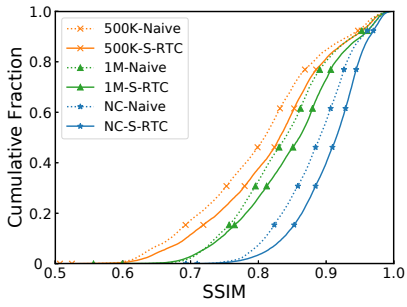
Finally, when there is no bandwidth constraint or additional network delay but with increased (2%) packet loss rate, the median Viewport-PSNR of SphericRTC drops only 0.4 dB. The same can be observed for vanilla WebRTC as well. This indicates that SphericRTC/WebRTC is not very sensitive to a moderate level of packet loss. However, when the packet loss rate increases to 5%, both SphericRTC and WebRTC perform much worse. In all scenarios, SphericRTC consistently outperforms the naive WebRTC implementation as indicated in Figures 9(a) and 9(b).

# 6 RELATED WORK

Real-time 360-degree video communication has many practical use cases. For example, Heshmat et al. set up a telepresence robot that

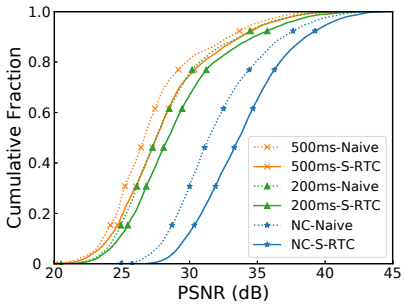



(a) Viewport-PSNR (dB)

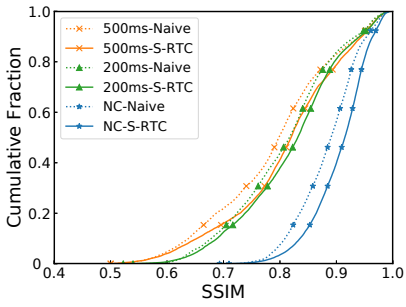



(b) Viewport-SSIM

**Figure 7: Viewport quality comparison under different network bandwidth settings {500 Kbps, 1 Mbps, no constraint}. S-RTC means SphericRTC.**

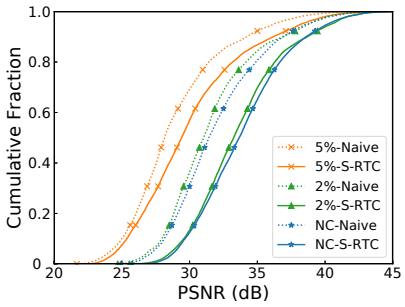



(a) Viewport-PSNR (dB)

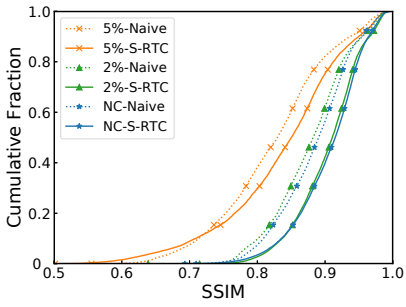



(b) Viewport-SSIM

**Figure 8: Viewport quality comparison under different additional network delay settings {500 ms, 200 ms, no additional delay}. S-RTC means SphericRTC.**




(a) Viewport-PSNR (dB)




(b) Viewport-SSIM

**Figure 9: Viewport quality comparison under different packet loss rate settings {5%, 2%, natural}. S-RTC means SphericRTC.**

can deliver 360-degree videos to a remote viewer via WebRTC to provide shared outdoor activity experiences [29]. Ha et al. proposed a telepresence wheelchair with 360-degree views delivered over WebRTC [27]. Researchers have developed systems enabling other types immersive media communication, such as virtual reality (VR) conferencing, in real-time [24–26]. For example, in [25], Gunkel et al. used an RGB-D camera to capture photo-realistic representations of users. They developed a system to transmit the captured representations via WebRTC to remote VR conference participants.

To support 360-degree live streaming, researchers have proposed solutions for improving stitching performance and encoding performance. For example, Chen et al. designed LiveTexture that uses an event-driven stitching algorithm for tiled 360-degree video streaming [18], and Lee et al. proposed foveated 360-degree video stitching [30]. Ballard et al. proposed RATS that uses GPU-based hardware encoder for encoding tiled 360-degree videos [16]. Midoglu et al. proposed a framework that leverages RATS for evaluating 360-degree video streaming performance [34]. These works focus on improving the stitching and encoding performance, while we focus on improving the quality of user-observed views by transforming and transmitting 360-degree video frames in adaptive representations.

Recently, Shi et al. proposed Freedom to support bandwidth-efficient delivery of VR content, including 360-degree videos, from nearby mobile edge cloud (MEC) servers [40]. To save bandwidth, the MEC server generates and encodes Visible Area with Margin (VAM) frames on the GPU. These VAM frames contain only part of an omnidirectional frame, meaning that portions of the user view outside of the VAM will not be rendered correctly.

## 7 CONCLUSION

360-degree video allows a more-immersive experience compared to the traditional video format. However, delivering 360-degree video requires significant network bandwidth. In the real-time setting, any approach to mitigate this high-bandwidth requirement must be applied within the real-time latency constraints.

SphericRTC applies a novel approach to allow efficient, content-adaptive 360-degree video communication within real-time limitations. The system demonstrates that multiple components, view orientation feedback, offset direction/magnitude selection, and fast frame transformation can be effectively integrated within a single practical system. The SphericRTC approach is validated by its performance: the median Viewport-PSNR in SphericRTC is 2.25 dB higher than that of the baseline system.

## 8 ACKNOWLEDGEMENT

We appreciate constructive comments from anonymous referees. This work is partially supported by National Science Foundation under grants CNS-1618931, CNS-1943250, and OAC-1761963.

# REFERENCES


[1] 2001. tc(8) — Linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[2] 2004. The Secure Real-time Transport Protocol (SRTP). https://tools.ietf.org/html/rfc3711.

[3] 2015. Transport-wide Congestion Control. https://tools.ietf.org/html/draft-holmer-rmcat-transport-wide-cc-extensions-01.

[4] 2015. WebRTC Data Channels. https://tools.ietf.org/html/draft-ietf-rtcweb-data-channel-13.

[5] 2016. Google Congestion Control Algorithm. https://tools.ietf.org/html/draft-ietf-rmcat-gcc-02.

[6] 2016. Next-generation video encoding techniques for 360 video and VR. https://code.facebook.com/posts/1126354007399553/next-generation-video-encoding-techniques-for-360-video-and-vr/.

[7] 2016. Overview of POSIX shared memory. http://man7.org/linux/man-pages/man7/shm_overview.7.html.

[8] 2017. End-to-end optimizations for dynamic streaming. https://code.facebook.com/posts/637561796428084/end-to-end-optimizations-for-dynamic-streaming/.

[9] 2019. Transform360. https://github.com/facebook/transform360.

[10] 2020. CUDA. https://developer.nvidia.com/cuda-zone.

[11] 2020. Equirectangular Projection. http://mathworld.wolfram.com/EquirectangularProjection.html.

[12] 2020. Signaling and video calling. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling.

[13] 2020. WebRTC. https://webrtc.org/.

[14] 2020. WebRTC Testing. http://webrtc.github.io/webrtc-org/testing/.

[15] 2020. YUV4MPEG2. https://wiki.multimedia.cx/index.php/YUV4MPEG2.

[16] Trevor Ballard, Carsten Griwodz, Ralf Steinmetz, and Amr Rizk. 2019. RATS: adaptive 360-degree live streaming. In *Proceedings of the 10th ACM Multimedia Systems Conference*. 308–311.

[17] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. 2016. Analysis and design of the google congestion control for web real-time communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems*. ACM, 13.

[18] Bo Chen, Zhisheng Yan, Haiming Jin, and Klara Nahrstedt. 2019. Event-driven stitching for tile-based live 360 video streaming. In *Proceedings of the 10th ACM Multimedia Systems Conference*. 1–12.

[19] Xavier Corbillon, Francesca De Simone, and Gwendal Simon. 2017. 360-degree video head movement dataset. In *Proceedings of the 8th ACM on Multimedia Systems Conference*. 199–204.

[20] Xavier Corbillon, Francesca De Simone, Gwendal Simon, and Pascal Frossard. 2018. Dynamic adaptive streaming for multi-viewpoint omnidirectional videos. In *Proceedings of the 9th ACM Multimedia Systems Conference*. 237–249.

[21] Xavier Corbillon, Gwendal Simon, Alisa Devlic, and Jacob Chakareski. 2017. Viewport-adaptive navigable 360-degree video delivery. In *Communications (ICC), 2017 IEEE International Conference on*. IEEE, 1–7.

[22] Mario Graf, Christian Timmerer, and Christopher Mueller. 2017. Towards bandwidth efficient adaptive streaming of omnidirectional video over http: Design, implementation, and evaluation. In *Proceedings of the 8th ACM on Multimedia Systems Conference*. ACM, 261–271.

[23] Yu Guan, Chengyuan Zheng, Xinggong Zhang, Zongming Guo, and Junchen Jiang. 2019. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *Proceedings of the ACM Special Interest Group on Data Communication*. 394–407.

[24] Simon Gunkel, Martin Prins, Hans Stokking, and Omar Niamut. 2017. *WebVR meets WebRTC: Towards 360-degree social VR experiences*. IEEE.

[25] Simon NB Gunkel, Marleen DW Dohmen, Hans Stokkinq, and Omar Niamut. 2019. 360-Degree Photo-realistic VR Conferencing. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. IEEE, 946–947.

[26] Simon NB Gunkel, Martin Prins, Hans Stokking, and Omar Niamut. 2017. Social VR platform: Building 360-degree shared VR spaces. In *Adjunct Publication of the 2017 ACM International Conference on Interactive Experiences for TV and Online Video*. 83–84.

[27] Van Kha Ly Ha, Rifai Chai, Hung T Nguyen, et al. 2020. A Telepresence Wheelchair with 360-Degree Vision Using WebRTC. *Applied Sciences* 10, 1 (2020), 369.

[28] Jian He, Mubashir Adnan Qureshi, Lili Qiu, Jin Li, Feng Li, and Lei Han. 2018. Rubiks: Practical 360-Degree Streaming for Smartphones. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 482–494.

[29] Yasamin Heshmat, Brennan Jones, Xiaoxuan Xiong, Carman Neustaedter, Anthony Tang, Bernhard E Riecke, and Lillian Yang. 2018. Geocaching with a beam: Shared outdoor activities through a telepresence robot with 360 degree viewing. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–13.

[30] Wei-Tse Lee, Hsin-I Chen, Ming-Shiuan Chen, I-Chao Shen, and Bing-Yu Chen. 2017. High-resolution 360 Video Foveated Stitching for Real-time VR. In *Computer Graphics Forum*, Vol. 36. Wiley Online Library, 115–123.

[31] Xing Liu, Bo Han, Feng Qian, and Matteo Varvello. 2019. LIME: understanding commercial 360° live video streaming services. In *Proceedings of the 10th ACM Multimedia Systems Conference*. ACM, 154–164.

[32] Yao Liu, Chao Zhou, Shuoqian Wang, and Mengbai Xiao. 2019. FFmpeg360 for 360-degree videos: edge-based transcoding, view rendering, and visual quality comparison: poster. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 337–339.

[33] Anahita Mahzari, Afshin Taghavi Nasrabadi, Aliehsan Samiei, and Ravi Prakash. 2018. Fov-aware edge caching for adaptive 360 video streaming. In *Proceedings of the 26th ACM international conference on Multimedia*. 173–181.

[34] Cise Midoglu, Özgü Alay, and Carsten Griwodz. 2019. Evaluation Framework for Real-Time Adaptive 360-Degree Video Streaming over 5G Networks. In *Proceedings of the 2019 on Wireless of the Students, by the Students, and for the Students Workshop*. 6–8.

[35] Afshin Taghavi Nasrabadi, Anahita Mahzari, Joseph D Beshay, and Ravi Prakash. 2017. Adaptive 360-degree video streaming using scalable video coding. In *Proceedings of the 2017 ACM on Multimedia Conference*. ACM, 1689–1697.

[36] Afshin Taghavi Nasrabadi, Aliehsan Samiei, and Ravi Prakash. 2020. Viewport prediction for 360° videos: a clustering approach. In *Proceedings of the 30th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. 34–39.

[37] Stefano Petrangeli, Viswanathan Swaminathan, Mohammad Hosseini, and Filip De Turck. 2017. An http/2-based adaptive streaming framework for 360 virtual reality videos. In *Proceedings of the 2017 ACM on Multimedia Conference*. ACM, 306–314.

[38] Feng Qian, Bo Han, Qingyang Xiao, and Vijay Gopalakrishnan. 2018. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. 99–114.

[39] Henning Schulzrinne, Steven Casner, R Frederick, and Van Jacobson. 2003. RFC3550: RTP: A transport protocol for real-time applications.

[40] Shu Shi, Varun Gupta, and Rittwik Jana. 2019. Freedom: Fast recovery enhanced vr delivery over mobile networks. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. 130–141.

[41] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[42] Chenglei Wu, Zhihao Tan, Zhi Wang, and Shiqiang Yang. 2017. A dataset for exploring user behaviors in VR spherical video streaming. In *Proceedings of the 8th ACM on Multimedia Systems Conference*. 193–198.

[43] Lan Xie, Zhimin Xu, Yixuan Ban, Xinggong Zhang, and Zongming Guo. 2017. 360probdash: Improving qoe of 360 video streaming using tile-based http adaptive streaming. In *Proceedings of the 25th ACM international conference on Multimedia*. 315–323.

[44] Lan Xie, Xinggong Zhang, and Zongming Guo. 2018. Cls: A cross-user learning based system for improving qoe in 360-degree video adaptive streaming. In *Proceedings of the 26th ACM international conference on Multimedia*. 564–572.

[45] Tan Xu, Bo Han, and Feng Qian. 2019. Analyzing viewport prediction under different VR interactions. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 165–171.

[46] Yang Xu, Chenguang Yu, Jingjiang Li, and Yong Liu. 2012. Video telephony for end-consumers: measurement study of Google+, iChat, and Skype. In *Proceedings of the 2012 Internet Measurement Conference*. ACM, 371–384.

[47] Jun Yi, Shiqing Luo, and Zhisheng Yan. 2019. A measurement study of YouTube 360° live video streaming. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM, 49–54.

[48] Chenguang Yu, Yang Xu, Bo Liu, and Yong Liu. 2014. "Can you SEE me now?" A measurement study of mobile video calls. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 1456–1464.

[49] Alireza Zare, Alireza Aminlou, Miska M Hannuksela, and Moncef Gabbouj. 2016. HEVC-compliant tile-based streaming of panoramic video for virtual reality applications. In *Proceedings of the 2016 ACM on Multimedia Conference*. ACM, 601–605.

[50] Chao Zhou, Zhenhua Li, and Yao Liu. 2017. A measurement study of oculus 360 degree video streaming. In *Proceedings of the 8th ACM on Multimedia Systems Conference*. 27–37.