DTP: Deadline-aware Transport Protocol

Hang Shi
Tsinghua University
shi-h15@mails.tsinghua.edu.cn

Yong Cui
Tsinghua University
cuiyong@mail.tsinghua.edu.cn

Feng Qian
University of Minnesota - Twin Cities
fengqian@umn.edu

Yuming Hu
Tsinghua University
hym17@mails.tsinghua.edu.cn

ABSTRACT

More and more applications have deadline requirements for their data delivery such as 360° video, cloud VR gaming and autonomous driving. Those applications usually are bandwidth hungry. Fortunately, the data of those applications can be split into multiple blocks with different priorities making it possible to reduce the bandwidth consumption by prioritizing some blocks over others. However, the existing transport layer is too primitive to accomplish that. So those applications are forced to build their own customized and complex wheels. In this work, we propose Deadline-aware Transport Protocol (DTP) to provide deliver-before-deadline service. The application expresses the deadline and metadata of the data to DTP. Then DTP tries to meet the requirement by scheduling blocks. Compared to existing protocols, DTP provides meaningful service and reduces the burden of the application developer.

CCS CONCEPTS

• **Networks → Transport protocols**; **Network protocol design**; **Public Internet**.

KEYWORDS

Deadline, Multiplexed stream transport, QUIC

ACM Reference Format:
Hang Shi, Yong Cui, Feng Qian, and Yuming Hu. 2019. DTP: Deadline-aware Transport Protocol. In *3rd Asia-Pacific Workshop on Networking 2019 (APNet '19), August 17–18, 2019, Beijing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3343180.3343191

APNet '19, August 17–18, 2019, Beijing, China

ACM ISBN 978-1-4503-7635-8/19/08…$15.00
https://doi.org/10.1145/3343180.3343191

1 INTRODUCTION

A wide spectrum of emerging applications such as VR, AR, autonomous driving, ultra-low latency gaming, and large-scale IoT systems are being quickly commercialized. Networking plays an increasingly important role in supporting these applications as fostered by the ubiquitous wireless access in the last mile, the proliferation of the edge/cloud infrastructures, and the advance of computation-intensive machine learning and AI. For example, a remote cloud pre-renders HD VR scenes allowing the client to fetch and render them at a very low cost [6]; AR applications offload sophisticated object recognition tasks to a nearby edge [7]; self-driving cars can communicate among themselves to make road detection and path planning safer and more accurate [15]. All these tasks need underlying support from robust, reliable, and high-performance networking.

Compared to traditional network-intense applications such as web, FTP, and multimedia streaming, the above emerging applications exhibit several unique differences from the perspective of network traffic workload and QoE requirements.

(1) They need to deliver multiple concurrent blocks with varying priorities. Take VR content delivery as an example. In order to reduce the network bandwidth consumption, recent systems [3, 13] spatially divide the scene into tiles, and selectively transfer the tiles based on the viewer's orientation. Therefore, the priority of a tile depends on the likelihood of it being perceived by the viewer.
(2) The applications' content delivery is delay-sensitive with clearly definable deadlines. For VR content delivery, each tile's deadline is its playback time minus local processing time; for 3D environment sharing among connected vehicles, the delivery deadline of a 3D object (*e.g.,* an incoming vehicle) can be calculated based on the object's distance and moving speed (the motion vector [15]). If the content cannot be delivered by the deadline, the delivery efforts may be wasteful.
(3) The application should be capable of dynamically adjusting the priority and the deadline, as well as canceling a data block even after it is passed to the transport

layer's send buffer. This may occur when, for example, the VR server has estimated a more accurate head pose position, or when a self-driving car updates a 3D object's moving trajectory, distance, or speed.

In §2, we showcase more examples that call for the above features. Admittedly, they can be realized at the application layer. Many existing systems indeed did that, with the downside being the additional development overhead and inflated application code base. Realizing all transport features in the application layer also violates layered principle in networking, making the application hard to develop and maintain. In addition, realizing some of the features such as canceling a piece of committed data may even require highly intrusive modification such as injecting an OS kernel module [13].

Motivated by the above use cases, in this position paper, we propose DTP (Deadline-aware Transport Protocol), a transport protocol to support the aforementioned features: content priority, deadline, dynamic adjusting them, and canceling enqueued data. DTP thus lets developers focus on the application logic (*e.g.,* how to compute the deadline) instead of the common transport-layer mechanism required by the applications (*e.g.,* how to enforce the deadline). We use QUIC as the base transport protocol due to its increasing popularity, its full user-space implementation, and its support of concurrent streams[1] that provide a useful building block for DTP.

Despite the straightforward idea, developing DTP and integrating it into QUIC is challenging in several aspects. How to properly design the application API? How to make DTP's data block based content delivery compatible with QUIC's byte stream oriented paradigm? How to allow a block to be safely canceled? How to design a scheduler that jointly considers the blocks' priority and deadline? We provide high-level solution directions in the remainder of this paper.

Overall, the contributions made by this paper is two-fold. First, we identify common transport-layer requirements from emerging applications such as AR, VR, and autonomous driving. Second, we propose DTP and sketch its key design. We are currently implementing DTP, and plan to thoroughly evaluate it on real applications and to make our implementation open-source.

2 BACKGROUND AND MOTIVATION

Emerging applications such as 360° video and cloud VR/gaming has the deadline requirement for its data transfer. To get a smooth user experience, they use similar techniques such as viewport prediction and transfer prioritization. We first conduct case studies of those applications and then motivate our new transport protocol.

2.1 Case study 1: 360° video streaming

360° video is gaining popularity on video platforms such as YouTube and Facebook. User is free to change her view direction when watching the video and get an immersive experience. It is realized by projecting the panoramic screen onto the display based on the user's viewport. Many researchers developed tile-based streaming[9, 14] for 360° video. The panoramic scene is divided into many tiles. Instead of streaming the whole panoramic scene, it only streams those tiles which fall into the user's viewport[13], or stream in-viewport tiles in high quality compared to non-viewport tiles[3]. Then on the client side, the tiles are get decoded in parallel and combined together. The delay of one tile will stall the whole video streaming otherwise user will see the fragmented content.

The tile-based solution is simple and effective in reducing the bandwidth requirement and smoothing the playback experience. One key module of the tile-based streaming is to rank and determine the tile set to stream. The tile to stream should be chosen based on the attention of the user. The more likely it is to be viewed by the user, the more important is the tile. So researchers develop various viewport adaptive methods[3, 13, 14] to facilitate the tile-based streaming. One common approach is to predict the user's head movement to determine which tiles a user is going to watch in the next few seconds then request those tiles. To cope with the prediction error, the server needs to be able to update rankings of the tile set based on the most recent prediction result.

2.2 Case study 2: cloud VR gaming

Virtual reality can provide user completely immersive experience. Due to the computing power and thermal limit of the mobile phone, it is hard to realize high-quality VR experience. To overcome those limitations, many studies try to leverage the cloud/edge computation power to render high-quality VR content and stream it back to the mobile device. They split the VR scene to multiple slices and do parallel rendering, encoding and transferring. This is similar to the 360° video. The difference with 360° video is VR game content can be split into dynamic foreground object and relatively static background scene. They have different movement patterns. The foreground object is interactive and sometimes maybe manipulated by the user. However, the background scene is relatively static and only change when user trigger movement in the virtual world and the speed of the virtual world movement is not the same as the user movement in the real world unlike the viewport change triggered by head movement.

Based on this observation, Zeqi *et al.* [6], Teemu *et al.* [5] develop cooperative rendering architecture called Furion and CloudVR. The mobile device is responsible to render the

dynamic foreground object and the cloud server handles the background scene rendering. CloudVR supports dynamic object placement. When the user begins to interact with an object, the object will be fetched from the server to be rendered locally. Furion only has a single foreground interactive object but it does prediction and pre-fetch of the background scene based on user location and movement in the virtual world. Luyang *et al.* [8] also develop a remote rendering untethered VR system. It uses the display VSync signal to synchronize the client displaying and remote server rendering. Missing the VSync signal can lead to severe display delay. This VSync delay can be viewed as the deadline of the object transmission.

2.3 Case study 3: cooperative augmented vehicular reality

Autonomous vehicles use a variety of sensors such as LiDAR, Radar and stereo cameras to sense the surroundings. These sensors scan the surrounding environment periodically. However, they have limited range and only have line-of-sight visibility so they cannot perceive far away or occluded objects. To overcome those limitations, Hang *et al.* [15] proposed the Augmented Vehicular Reality(AVR), which extends the visual horizon of vehicles by wirelessly sharing visual information between each other.

Sharing the visual information wirelessly between vehicle faces several challenges. First, the bandwidth requirement of transmitting full visual information is far exceeding the current Vehicle-to-Vehicle wireless technology. To reduce the bandwidth requirement of communication, the sender can prioritize some objects to others. The visual information can be separated into two parts: static 3D map and dynamic objects. Those dynamic objects can be split into multiple tiers based on their distance/velocity. Then the sender can choose to send only the critical objects to another vehicle, saving the bandwidth.

Second, the latency requirement of that information is stringent because the vehicle is fast moving and needs to react quickly. By applying motion prediction on some objects such as cars and human, it can avoid sending them. The motion prediction can also facilitate the prioritization of objects.

2.4 Common transport requirements of emerging real-time applications

We summarize the common transport requirements of those applications as follows.

Deadline requirement for its block-based data transmission. Those applications all generate and process the data in block fashion and each block has a clear deadline requirement. A partial delivered block is useless for those applications. Each block can be independently processed. In 360° video, each tile in one GOP is a block and the deadline of tiles and objects is its render time minus local processing time. In cloud VR gaming, each background scene and foreground object is a block. Missing deadline will cause the blank or stalled scene. In autonomous driving, each 3D object is a block and its deadline can be calculated based on its distance and moving speed.

Multiple blocks with different priorities and deadlines. In those applications, there are many blocks need to be transferred. Those blocks have different impact on application performance making it possible to reduce the bandwidth consumption by prioritizing some blocks over others.

In 360° video, the priority of the tile can be set according to the user's viewport. The tile in the center of the viewport has a higher priority compared to peripheral tiles.

In cloud VR gaming, the foreground objects and background scene are separate blocks. Foreground objects have a higher priority than the background scene. The deadlines of foreground objects are tighter than one of the background scene. Different foreground objects can also have different priorities and deadlines based on the possibility of user interaction.

In autonomous driving, different dynamic objects have different priorities. One way to set the priority of an object is to set it by how dangerous it is to other vehicles if they do not have the sight of that object. For example, the fast oncoming vehicle has a higher priority to the following vehicle than a slow-moving and far-away bicycle.

Predictive fetching and correction of the prediction. To get a smooth user experience, those applications usually use predictive fetching/rendering based on possible movement trajectory. These predictions may have errors and need to be corrected immediately.

In 360° video, the tile priority is determined based on the viewport. And the viewport is constantly changing. It is necessary to use the user head movement trace to predict the viewport in the next few seconds. When the prediction is updated, the tile priority should be updated too. If the user completely turns its attention away from the previous prediction, then the tile transmission should be canceled too.

In cloud VR gaming, the background scene is fetched based on the avatar location and possible movement direction. The foreground object is also dynamically changed based on the user interaction intention. Similar to 360° video they both involve the prediction and prefetch.

In autonomous driving, only part of the frame which is out of line-of-sight is shared with other vehicles. The relative position the occlusion is constantly changing. It is necessary to predict the occlusion status to determine the priority of the

object. The movements of other objects need to be predicted to determine its priority too. Those predictions may be wrong and should be updated too.

2.5 Lack of support in current transport protocols

In short, those applications need a transport solution which can provide a dynamic, multi-block based and deliver-before-deadline service. No existing transport protocol can provide such a service. TCP and UDP are single-stream oriented. And it is not aware of the deadline of the data transmission. QUIC has multi-stream support. Each stream can be assigned with a priority according to its RFC[4]. However, it lacks the deadline support and the priority usage in the scheduler is not well studied too. QUIC is also providing reliable delivery which conflicts with the real-time nature of the above applications. Those applications prefer the data freshness over reliability. Real-time Transport Protocol(RTP)[16] does favor freshness over reliability but it is not aware of the deadline and does not provide block-based delivery either. The lack of support of current transport protocols are summarized in table 1.

Features	TCP	UDP	RTP	QUIC	Ideal
Block-based	No	Yes	No	No	Yes
Deadline-aware	No	No	No	No	Yes
Timeliness	No	No	Yes	No	Yes
Prioritization	No	No	Static	Static	Dynamic

Table 1: Comparison of different transport protocols

Without proper transport protocol support, applications are forced to build their own wheels. Each application has to handle the prioritization, sending and acknowledging of data *etc.* which are tedious and complex, resulting in a bloated code base. Otherwise, they just use some off-the-shelf protocols such as TCP which results in the inferior performance. We propose Deadline-aware Transport Protocol (DTP) to provide such a service. DTP follows the classic principle of separating policy and mechanism. Application provide the requirement and necessary metadata of the data to transport layer. The metadata indicates the policy for how the data should be transmitted. Transport layer provides the common mechanism to implement the policy. The details of the policy execution process is encapsulated at the lower level, which frees the application from the micro management job of the data delivery.

3 DESIGN

DTP is extended based on QUIC because QUIC provides many useful building blocks including full encryption, flexible congestion control and multiplexing without head-of-line blocking *etc.* QUIC is also a user space library and has many well-supported implementations. In this section, we discuss the design of DTP.

3.1 Architecture

The sender side architecture is shown in section 3.1. When block along with its metadata is sent from the application to the transport layer, it will be put into a dedicated buffer. Then scheduler will choose the block to send. After that, the packetizer will divide those blocks into packets. A congestion control module is responsible to send packets, collects ACK, do packet loss detection. Then it will put the lost data back to the retransmission queue of each block. And send back network status such as bandwidth and RTT to the scheduler to facilitate the schedule decision.

We made several changes to QUIC architecture. First and foremost, DTP extend the QUIC to support the block based delivery without changing the wire-format. QUIC RFC[4] requires QUIC implementation to support the stream prioritization. So an intuitive way of mapping block to stream is to put blocks with same priority into one stream. In this way, we can reuse most part of a QUIC scheduler for prioritization. However, this mapping faces an inherent problem. QUIC's reliable byte stream nature conflicts the timeliness and dynamic of block delivery. Suppose one block in the middle of stream 1 passed its deadline and is decided to be dropped. There is no easy way to mark the specified block as obsolete. Existing extension to add unreliable or partial reliable transmission to QUIC cannot accomplish that. Lubashev *et al.* [10] propose to mark parts of the stream(to a certain byte offset) as unreliable. Both solutions cannot specify a byte range to be unreliable. This solution also introduces many other complexities. Such as the conflict between max block size and the max stream data size for flow control. What if the total block size exceeding the flow control limit of the stream? And we also need another framing sub-protocol(specifying delimiter) to assemble blocks to stream. This will introduce extra processing and transport overhead.

We take another simple approach: mapping block to QUIC stream one to one. If the block is decided to be dropped, we just utilize the standard stream cancellation process of existing QUIC protocol, which is sending the *RESET_STREAM* frame to cancel the stream. The *RESET_STREAM* frame will trigger the flow control update as the normal QUIC. Using this mapping, we can reuse the max stream data size as the max block size too so don't need to introduce a new transport parameter during the handshake. We can also map the block id to stream id without breaking the stream id semantic using eq. (1). The stream type bits is defined in QUIC RFC[4]. By using the FIN bit of the STREAM frame, the receiver can easily determine whether the block transmission is finished.

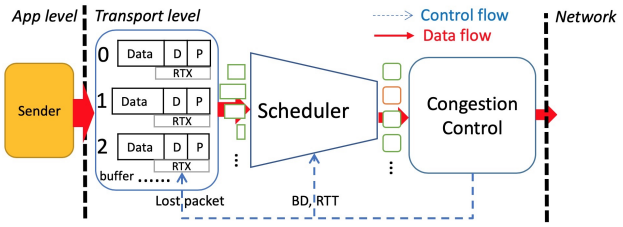

Figure 1: The architecture of DTP

```
Sender:
int send(int fd, void *block, size_t size, int id,
int priority, int deadline);
int update(int fd, int id, int priority, int deadline);
int retreat(int fd, int id);
Receiver:
int receive(int fd, void *block, size_t size, int *id);
```

Figure 2: Interfaces of DTP

$$stream\_id = (block\_id << 2)\ \&\ stream\_type\_bits \quad (1)$$

Second, we disable the multiplexing of different streams into one packet as suggested in QUIC RFC[4]. The reason is twofold. First is to avoid one packet loss blocks multiple blocks' progress. Second is to simplify the retransmission process when one of blocks get dropped. Without multiplexing, retransmission of the lost packet will involve fewer payload changes.

Third, the scheduler obtains networking estimate from the congestion control module. The bandwidth can be calculated using the congestion window (like Cubic) or the delivery rate (like BBR). We set the default congestion control algorithm to BBR because it won't fill the in-network buffer.

3.2 API

As pointed out in section 2.4, what applications really need is a block-based delivery. When we talk about the deadline, the meaningful deadline to the application is the block completion time *i.e.,* the time between when the block is generated at the sender and when the block is submitted to the application at the receiver. We extend the BSD socket API to let application attach metadata along with the data block, as is shown in fig. 2. Those metadata is listed as follows.

(1) Each block has a unique identifier, represented by an integer *id*. The id will be sent to the receiver so sender and receiver can use this to identify the block. This argument should be carried when requesting an update or retreat of a block.
(2) Each block has a deadline requirement. The *deadline* argument represents the desired block completion time in milliseconds.
(3) Each block has its own importance to the user experience. The application can assign the block a priority to indicate the importance of the block. The lower the *priority* value, the more important the block thus the more likely the block get delivered.

The API of DTP is shown in fig. 2. The fd is the QUIC connection id. The *update* and *retreat* function is used to update or cancel the block sending. The return value of *update* and *retreat* function indicate the success of the action. Both functions will return 0 if succeeds, -1 if fails.

Both *send* and *receive* are running in non-block mode. The return value of *send* is the delivery rate of blocks with the same priority. The application can utilize this information to adjust the sending rate of each priority. For example, if the application finds the lowest priority blocks always get dropped due to the limited bandwidth, the application can stop generate those blocks to save the computation power.

The *receive* function will copy the receiving block into the buffer allocated by the caller (Specified by *void * block* and *size_tsize*). The id of the block will be put into the *id* argument. *receive* function will only copy the full received block. If there is still packets of the block that has not arrived, receive will not return those block to application. That is because the block may get canceled or dropped by the sender. A partial block will cause confusion of the receiver. The return value of *receive* is the actual size of the block. If the block size specified in the receive function is smaller than the size of the receiving block, then the block will be partial copied. Next time receive function is called, the remaining block will be copied and the id will be the same. This fragmentation will give extra burden to applications. To avoid the fragmentation, sender and receiver can negotiate a max block size when handshaking.

3.3 Deadline-aware scheduler

We extend the scheduler in QUIC to take into account many factors when picking blocks in sender buffer to send. The goal of the schedule is to deliver as much as high priority data before the deadline and drop obsolete or low-priority blocks. To achieve this, the scheduler utilizes both bandwidth and RTT measurement provided by the congestion control module and the metadata of blocks provided by the application to estimate the block completion time. The scheduler will run each time ACK is received or the application push the data.

Currently, the scheduler in QUIC will only consider the priority of the stream. However, this simple algorithm cannot get optimal result in some cases. Suppose the bandwidth reduces and the scheduler choose not to send the low priority stream. Then the bandwidth is restored. In this time, the data block with lower priority is closer to the deadline then the high priority stream. If in this round the scheduler still chooses to send the high priority stream, then the low priority stream may miss the deadline next round and get dropped. In some cases, the scheduler can choose to send a low priority stream because it's more urgent. But it should do so without causing the high priority stream missing the deadline.

Another factor which needs to be taken into account is the block remaining size to transfer. Dropping a block when it is about to finish transmission will waste all previous resource because partial delivery of a block is meaningless.

$$Real\ Priority = priority \times |deadline - (\frac{remaining\ size}{bandwidth} + \frac{RTT}{2})| \quad (2)$$

We use a function $f(deadline, priority, remaining\ size, RTT, bandwidth)$ to combine all those factors into a *Real Priority* value (smaller value means higher priority). Then the scheduler just picks the block with the smallest value. One example function is shown in eq. (2). The function calculates the block remaining transmission time using $\frac{remaining\ size}{bandwidth} + \frac{RTT}{2}$ then compare it to the deadline. The more close to the deadline, the smaller the real priority value. In this way, the scheduler can take into account the deadline approaching and priority into account. Blocks which are severely overdue will get big real priority value and can be dropped accordingly.

The scheduler algorithm works as follows. First, the scheduler updates the *Real Priority* value based on the function specified by the application. If the *Real Priority* is bigger than a threshold α which is also set by the application. Then it will drop the block and send the *RESET STREAM* frame to notify the other end. After that the block with the lowest *Real Priority* value is returned.

4 RELATED WORK

Traditional real-time application optimization There is a lot of work regarding how to improve traditional real-time application performance. The typical example is video conferencing. Many solutions couples the application level rate adaptation along with network transport.

WebRTC, Google Hangouts, Apple Facetime both take a reactive approach to adjust the bitrate of the video. When congestion happens, it will send the already encoded frame anyway, then reduce the encoding bitrate for future frames. This reactive process is slow and can not mitigate congestion.

Salsify[2] propose to codesign the encoder and transport layer together to fine-tune the encoded frame size according to network capacity. However, this approach requires to replace the whole stack and does not support hardware encoder and decoder.

Real-time transport protocol There are also many transport protocols to support these real-time applications. RTP and RTCP were designed at the mid-1990s when the network condition is suboptimal. To make use of such a network, they design many complex mechanisms such as redundancy, time sync *etc.*. However, due to its complexity and many optimizations just for media transportation, it is not easy to deploy and adapt to newer applications and networking environment.

Perkins *et al.* [12] discuss the mapping between RTP and QUIC and extend QUIC to support real-time media transport. Palmer *et al.* [11] propose extension for QUIC to support video streaming. Compared with their proposals, our proposal is more general and focus on deadline requirements for emerging applications instead of optimizing for specific existing application.

5 CONCLUSION

Motivated by emerging real-time applications, we design a new transport protocol DTP to meet the deadline requirement of data delivery. We further design a new scheduler to prioritize some blocks when network bandwidth is limited. Our protocol is extended based on QUIC and fully compatible with QUIC wire format. Using our protocol can free the application developer from worrying about the network conditions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback on the paper. This work is supported by National Key R&D Program of China (no. 2018YFB1800303) and NSFC Project (No. 61872211).

REFERENCES

[1] Yong Cui, Tianxiang Li, Cong Liu, Xingwei Wang, and Mirja Kühlewind. 2017. Innovating transport with QUIC: Design approaches and research challenges. *IEEE Internet Computing* 21, 2 (2017), 72–76.

[2] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. 2018. Salsify: Low-Latency Network Video through Tighter Integration between a Video Codec and a Transport Protocol. In *NSDI 18*.

[3] 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.

[4] Jana Iyengar and Martin Thomson. 2019. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-19.

IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-19.txt

[5] Teemu Kämäräinen, Matti Siekkinen, Jukka Eerikäinen, and Antti Ylä-Jääski. 2018. CloudVR: Cloud Accelerated Interactive Mobile Virtual Reality. In *2018 ACM Multimedia Conference on Multimedia Conference*. ACM, 1181–1189.

[6] Zeqi Lai, Y Charlie Hu, Yong Cui, Linhui Sun, and Ningwei Dai. 2017. Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. ACM, 409–421.

[7] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge Assisted Real-time Object Detection for Mobile Augmented Reality. In *ACM MobiCom*.

[8] Luyang Liu, Ruiguang Zhong, Wuyang Zhang, Yunxin Liu, Jiansong Zhang, Lintao Zhang, and Marco Gruteser. 2018. Cutting the cord: Designing a high-quality untethered VR system with low latency remote rendering. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 68–80.

[9] Xing Liu, Qingyang Xiao, Vijay Gopalakrishnan, Bo Han, Feng Qian, and Matteo Varvello. 2017. 360 innovations for panoramic video streaming. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 50–56.

[10] Igor Lubashev. 2018. *Partially Reliable Message Streams for QUIC*. Internet-Draft draft-lubashev-quic-partial-reliability-03. IETF Secretariat. http://www.ietf.org/internet-drafts/draft-lubashev-quic-partial-reliability-03.txt

[11] Mirko Palmer, Thorben Krüger, Balakrishnan Chandrasekaran, and Anja Feldmann. 2018. The QUIC Fix for Optimal Video Streaming. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. ACM, 43–49.

[12] Colin Perkins and Jörg Ott. 2018. Real-time Audio-Visual Media Transport over QUIC. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. ACM, 36–42.

[13] 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*. ACM, 99–114.

[14] Feng Qian, Lusheng Ji, Bo Han, and Vijay Gopalakrishnan. 2016. Optimizing 360 video delivery over cellular networks. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*. ACM, 1–6.

[15] Hang Qiu, Fawad Ahmad, Ramesh Govindan, Marco Gruteser, Fan Bai, and Gorkem Kar. 2017. Augmented vehicular reality: Enabling extended vision for future vehicles. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*. ACM, 67–72.

[16] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. 2003. *RTP: A Transport Protocol for Real-Time Applications*. STD 64. RFC Editor. http://www.rfc-editor.org/rfc/rfc3550.txt