Open-Source RTP Library for High-Speed 4K HEVC Video Streaming

Aaro Altonen, Joni Räsänen, Jaakko Laitinen, Marko Viitanen, and Jarno Vanne
Ultra Video Group, Tampere University, Tampere, Finland
{aaro.altonen, joni.rasanen, jaakko.laitinen, marko.viitanen, jarno.vanne}@tuni.fi

***Abstract*— Efficient transport technologies for High Efficiency Video Coding (HEVC) are key enablers for economic 4K video transmission in current telecommunication networks. This paper introduces a novel open-source Real-time Transport Protocol (RTP) library called uvgRTP for high-speed 4K HEVC video streaming. Our library supports the latest RFC 3550 specification for RTP and an associated RFC 7798 RTP payload format for HEVC. It is written in C++ under a permissive 2-clause BSD license and it can be run on both Linux and Windows operating systems with a user-friendly interface. Our experiments on an Intel Core i7-4770 CPU show that uvgRTP is able to stream HEVC video at 5.0 Gb/s over a local 10 Gb/s network. It attains 4.4 times as high peak goodput and 92.1% lower latency than the state-of-the-art FFmpeg multimedia framework. It also outperforms LIVE555 with over double the goodput and 82.3% lower latency. These results indicate that uvgRTP is currently the fastest open-source RTP library for 4K HEVC video streaming.**

Keywords*— *Real-time Transport Protocol (RTP), High Efficiency Video Coding (HEVC), 4K video, video streaming, open-source library

I. INTRODUCTION

Proliferation of advanced multimedia devices, omnipresent connectivity, and popular video applications foster the growth of global IP video traffic, which is estimated to account for 82% of all IP traffic by 2022 [1]. Correspondingly, 4K video traffic is reported to grow more than 90% annually which means there is a snowballing demand for efficient 4K video streaming implementations, built on the latest video coding and transport technologies.

Mainstream international video coding standards, such as *Advanced Video Coding* (*AVC/H.264*) [2] and *High Efficiency Video Coding* (*HEVC/H.265*) [3], were developed to compress digital video for economic transmission. High-resolution video coding was particularly addressed by the state-of-the-art HEVC that is reported to compress 4K video with 40% higher coding efficiency than AVC for the same objective quality [4] and with up to 64% better efficiency for the same subjective quality [5].

The widespread *Real-time Transport Protocol* (*RTP*) was designed for real-time transfer of streaming media on the Internet. The latest RTP specification, RFC 3550 [6], is able to meet stringent timing requirements of real-time 4K HEVC video transmission, but it does not provide any *quality-of-service* (*QoS*) guarantees. Therefore, RFC 3550 also defines the *RTP Control Protocol* (*RTCP*) for RTP session control and QoS monitoring. Furthermore, an RTP payload format for HEVC is separately specified in RFC 7798 [7].

Over the past decades, several open-source RTP libraries have been announced. However, they either lack built-in support for RFC 7798 [8]-[15], are not compatible with RFC 3550 [16], or come with a complete multimedia framework that is not ideal for lightweight applications [17], [18].

In this paper, we present a novel RFC 3550 and RFC 7798 compliant RTP library called uvgRTP for real-time 4K HEVC streaming. It is available online on GitHub at

https://github.com/ultravideo/uvgRTP

The library was developed by our Ultra Video Group at Tampere University. It is written in C++ and licensed under a permissive 2-clause BSD. This cross-platform software can be run on Windows and Linux operating systems.

The uvgRTP library contains two main components: 1) *uvgRTP sender*; and 2) *uvgRTP receiver* that implement the sending and receiving ends, respectively. The uvgRTP sender is sped up with two techniques called optimized HEVC *Start Code Lookup* (*SCL*) and *System Call Clustering* (*SCC*) of which SCC is also applied in the uvgRTP receiver.

The remainder of this paper is organized as follows. Section II describes the basic operating principle of HEVC streaming over the RTP protocol. Section III investigates the feasibility and readiness of the existing RTP libraries for 4K HEVC video streaming. Sections IV-VI describe the proposed uvgRTP architecture, uvgRTP sender, and uvgRTP receiver, respectively. Section VII evaluates the performance of uvgRTP and benchmarks it together with LIVE555 [16] and FFmpeg [17]. Finally, Section VIII concludes the paper.

II. HEVC STREAMING OVER RTP

RFC 3550 specifies RTP header data of the RTP packet such as version and sequence numbers, a timestamp, and a payload format but not how media-specific packetization divides payload data into smaller chunks. In the case of HEVC, RFC 3550 is used in conjunction with RFC 7798 that specifies the RTP payload format for HEVC and the packetization/de-packetization rules for it.

HEVC bitstream is divided into *Network Abstraction Layer* (*NAL*) units and a single HEVC frame may contain multiple NAL units. The boundaries of NAL units are identified by start codes in HEVC bitstream. The length of the start code is three (0x000001) or four bytes (0x00000001).

RFC 7798 defines the format for a *Fragmentation Unit* (*FU*). FUs are used to fragment a single NAL unit into multiple RTP packets, which are better suited for reliable transportation. FUs, belonging to the same HEVC frame, have identical timestamps that are used to compose each frame from the respective FUs in the receiver end. Each FU also carries a unique sequence number and a FU type, i.e., the start, middle, and end FUs of each NAL unit are indicated separately in the FU header.

Ethernet provides the backbone for packet-switched networks and occupies the physical and data link layers of the

This work was supported in part by the European ECSEL project PRYSTINE (under the grant agreement 783190) and the Academy of Finland (decision no. 301820).

978-1-7281-9320-5/20/$31.00 ©2020 IEEE

TABLE I. FEATURES OF EXISTING OPEN-SOURCE RTP LIBRARIES

Library	RFC 3550	RFC 7798	License	Core Language	First Activity	Last Activity
ccRTP [8]	Yes	No	GPLv2	C++	2001	2015
JRTPLIB [9]	Yes	No	MIT	C++	1999	2020
libre [10]	Yes	No	BSD	C	2010	Active
PJSIP [11]	Yes	No	GPLv2	C	2003	Active
GoRTP [12]	Yes	No	GPLv3	Go	2011	2019
oRTP [13]	Yes	No	GPLv3	C	2001	Active
libjitsi [14]	Yes	No	Apachev2	Java	2003	Active
Restcomm Media-Core [15]	Yes	No	AGPLv3	Java	2011	2018
LIVE555 [16]	No	Partially	LGPLv3	C++	1996	Active
FFmpeg [17]	Yes	Yes	LGPLv2.1	C	2000	Active
GStreamer [18]	Yes	Yes	LGPLv2.1	C	2001	Active
uvgRTP	Yes	Yes	BSD	C++	2019	Active

Open Systems Interconnection (*OSI*) model. The Ethernet frames sent over the network consist of IP, *User Datagram Protocol* (*UDP*), RTP, HEVC, and FU frame headers as well as the HEVC payload.

III. OPEN-SOURCE RTP LIBRARIES

Table I lists existing notable open-source RTP libraries and characterizes them in terms of RFC 3550 and RFC 7789 compatibility, license, implementation languages, and development activity.

The libraries, available in [8]-[15], support RFC 3550 but not RFC 7798. Free Software Foundation's GNU project provides the ccRTP [8] library under a GPLv2 license. However, the project appears to have been inactive since 2015. JRTPLIB [9] is an RTP library developed at Hasselt University and released under a MIT license. However, it was recently announced that JRTPLIB is no longer under active development. Libre [10] is a generic implementation provided under the BSD license for multimedia communication. It supports 47 RFC specifications, but not all modules are in a stable state and may therefore be subject to changes. PJSIP [11] is another multimedia communication library that provides implementations for numerous protocols, including RTP. It is actively maintained by Teluu Ltd. and is provided under the GPLv2 license. GoRTP [12] provides an RTP stack for the Go programming language, under the GPLv3 license, but receives only occasional updates. The final three libraries, oRTP [13], libjitsi [14], and Restcomm Media-Core [15], belong to Linphone [19], Jitsi [20], and Restcomm server [21] respectively, but they are also provided as stand-alone libraries. They all contain additional functionality besides RFC 3550, which makes them less desirable for applications only requiring RTP support.

LIVE555 [16] is a popular multimedia streaming library. It is under active development and distributed under a LGPL license. LIVE555 implements fragmentation and reconstruction of HEVC bitstream, but it only partially supports RFC 7798 because it is not compatible with RFC 3550 but instead implements the former specification RFC 1889. Moreover, it does not perform SCL so it only accepts discrete NAL units as input and the HEVC bitstream has to be parsed by the application. Nevertheless, it is used, e.g., in the well-known multimedia player VLC [22].

Leaving the implementation of RFC 7798 and RTP fragmentation to the user has three shortcomings: 1) The goodput is decreased, because the user has to implement the RFC 7798 on top of a general send interface without any possibility for format specific optimizations; 2) The implementation effort is higher; and 3) Conforming to the specification is left to the user at the possible cost of decreased interoperability with other RTP implementations.

Among the existing libraries, FFmpeg [17] and GStreamer [18] are the only ones that support both the newest RFC 3550 specification and the RFC 7798 payload format for HEVC. Additionally, they both offer automatic fragmentation of RTP packets without any effort from the user.

FFmpeg is a well-known and actively developed multimedia framework that contains a multitude of libraries for various multimedia applications and includes support for RTP streaming. It is licensed under LGPLv2.1 with optional parts under GPLv2. However, using FFmpeg for RTP streaming requires linking the whole framework, which may not be desirable for lightweight streaming applications.

Similarly to FFmpeg, GStreamer is a multimedia framework with various plug-ins for a large range of applications. GStreamer is mainly written in C, but it provides bindings for other languages such as Python, Rust, and C++.

IV. UVGRTP ARCHITECTURE

Fig. 1 shows the high-level software architecture of our uvgRTP library and its basic operating principle when applied in two-way point-to-point communication between different multimedia applications. In this scheme, RTP is unidirectional and RTCP bidirectional.

A single application may open multiple concurrent uvgRTP sessions and each session can contain one or more media streamer instances. These instances can be divided into the following three functional units:

1) The uvgRTP sender fragments the input HEVC video stream, encapsulates it into RTP packets, and sends the packets over the network.
2) The uvgRTP receiver receives the RTP packets from the network, constructs a full HEVC stream from the packets, and updates the statistics of the RTP session.
3) The uvgRTP RTCP instance periodically sends error correction and synchronization data about the monitored RTP stream. Applications use this information to adjust media streaming parameters and synchronize different media streams.

The current version of the uvgRTP library has built-in support for HEVC video and the Opus audio format [23]. In addition, our generic *application programming interface* (*API*) makes it possible to implement any other media payload format on the application side as in [8]-[15]. In practice, a transmitted media format must be selected when the uvgRTP sender and receiver are created.

V. UVGRTP SENDER

The five main stages of the uvgRTP sending end are shown at the top of Fig. 2. The first stage is called *Start Code Lookup* (*SCL*). It organizes the given HEVC frame into NAL units. Secondly, the NAL units are fragmented into FUs. In the third stage, multiple FUs are passed on to the operating system with a single system call using *System Call Clustering* (*SCC*) [24]. Finally, FUs are written to a Socket Write Buffer and sent through the *Transmit* (*TX*) Queue. Next, these stages are described in more detail by paying special attention to the proposed optimization techniques.

A. Start Code Lookup (SCL)

This stage divides an input HEVC frame into NAL units that are identified by their start codes in a bitstream. The start code search has been optimized with two techniques.

First, our algorithm goes through the input frame in 8-byte chunks by using an 8-byte version of haszero [25] bitmask that determines whether a chunk contains a zero byte. If the haszero indicates that an 8-byte chunk contains a zero, the algorithm goes through the chunk one byte at a time in order to find a start code. If a start code is not found, the algorithm proceeds to next 8-byte chunk. Secondly, in order to find frame boundaries efficiently, the last byte of each input frame is set to zero and boundaries are checked only when a zero byte is found with haszero.

B. Fragmentation

The fragmentation stage splits the NAL units into FUs, which are then put into Ethernet frames to be sent over the network. The uvgRTP library supports the smallest Ethernet frame size of 1500 bytes, which is compatible with current mainstream Ethernet implementations [26]. It is also consistent with traditional network configurations that may not support larger jumbo frames.

The payloads are pushed to vectored I/O buffers. This way, the FU header and payload data can be passed on using pointers without reserving any additional memory or temporarily constructing complete FUs (headers and payload) by copying data. Reducing the number of copy operations has a significant impact on performance especially with resource-intensive video applications.

C. System Call Clustering (SCC)

A single high-resolution frame can be divided into numerous FUs, so issuing a single system call per FU would take a lot of processor time. In uvgRTP, this overhead is reduced to a fraction by implementing SCC with sendmmsg(2) – a system call introduced in Linux 3.0. It allows uvgRTP to send all RTP packets of the same HEVC frame together and only one system call is needed per frame.

D. Socket Write Buffer and TX Queue

The sendmmsg(2) system call writes the FUs into the Socket Write Buffer on the operating system side. By default, the size of the buffer is 212 KB, but it has to be increased

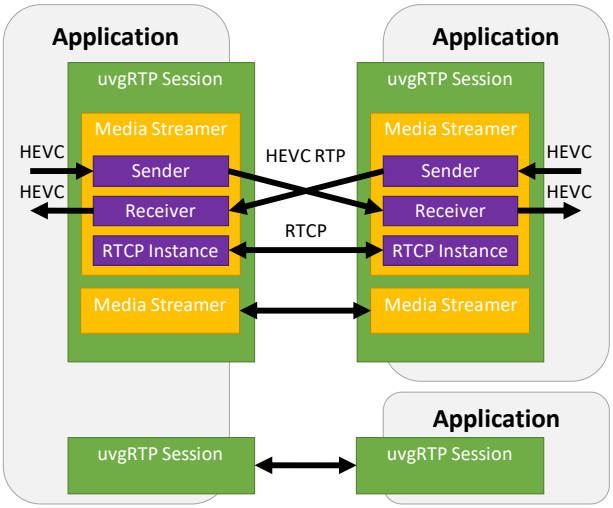

Fig. 1. High-level architecture of the proposed uvgRTP library.

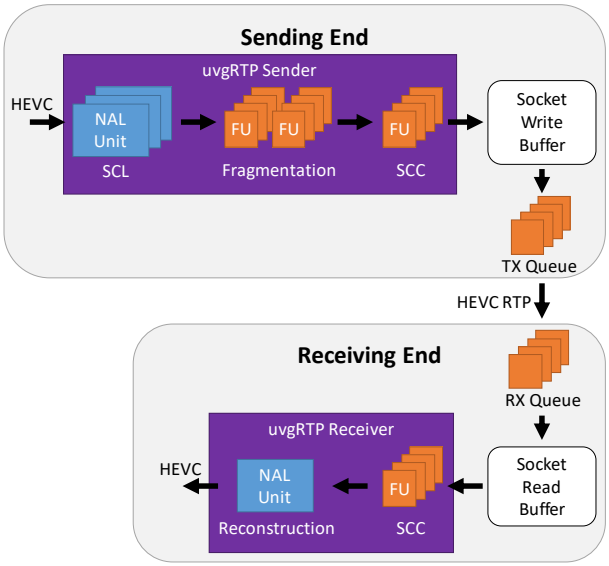

Fig. 2. Implemented data flow in the uvgRTP sender and receiver.

accordingly with the 'net.core.wmem_default' and 'net.core.wmem_max' variables if an application is able to send packets faster than the operating system can react.

The operating system sends the packets over the network via the TX Queue, which can become full if too many packets are queued. The length of the TX Queue can be adjusted by changing the 'txqueuelen' parameter of the network interface.

VI. UVGRTP RECEIVER

The four main stages of the uvgRTP receiving end are depicted at the bottom of Fig. 2. First, the operating system stores the RTP packet, received from the network, in the *Receive* (*RX*) Queue and moves it into a Socket Read Buffer. The SCC stage reads FUs from the buffer and passes them to the final stage where the full NAL unit is reconstructed from the FUs and passed on to the application.

A. RX Queue and Socket Read Buffer

First, an RTP packet received from the network is stored into the RX Queue. The queue length can be adjusted with the 'net.core.netdev_max_backlog' variable. Next, the operating system moves the packet to the Socket Read Buffer that can become full if packets are received faster that the operating

system is able to process them. The default size of the Socket Read Buffer is 212 KB and it can be resized with the 'net.core.rmem_default' and 'net.core.rmem_max' variables.

B. System Call Clustering (SCC)

The uvgRTP receiver pre-allocates space for 1024 FUs before calling recvmmsg(2). It listens to the Socket Read Buffer and uses SCC to read up to 1024 RTP packets from the buffer with only a single system call. The SCC makes it possible to increase the amount of transferred data from 1.44 KB to 1.48 MB per system call, so it has significant impact on processor utilization and goodput.

C. Reconstruction

The uvgRTP receiver keeps track of the received FUs by comparing the received FU count with the offset between the RTP sequence numbers of the first and last FUs that have the same timestamp. When all FUs of a NAL unit have been received, the receiver merges all FUs into a complete NAL unit and passes it to the application. If a single FU has been dropped, all other FUs of that NAL unit are also discarded, because the complete unit can no longer be reconstructed. The uvgRTP receiver supports both hook and polling-based output of NAL units.

VII. Performance Analysis

In our experiments, uvgRTP version 1.0 was benchmarked against FFmpeg version 4.2 and LIVE555 version 2018.12.14, which represent the state-of-the-art in RTP streaming and media processing. GStreamer was excluded from our evaluations because there was no straightforward way to integrate its closely-knit media processing filters into our benchmark setup. The other considered libraries were also omitted because of their incomplete functionalities for our measurements.

A. Experimental Setup

Fig. 3 illustrates our profiling platform. It was made up of two desktop computers (*A* and *B*) that were equipped with Intel Core i7-4770 and AMD Threadripper 2990WX *Central Processing Units* (*CPUs*), running Linux kernel versions 4.15.0 and 5.0.0, respectively. The computers were connected over a 10 Gbps *local area network* (*LAN*) through two Cisco SG350XG network switches.

All tests were carried out with a single 4K30p (3840×2160 pixels) HEVC test video sequence (598 frames) that was encoded to a bitrate of 164.95 Mb/s using the intra frame [27] period of 64. The encoding time was excluded from the measurements by encoding the sequence beforehand. To mitigate extra latency caused by file I/O, the sequence was memory-mapped to the address space of the RTP sender. The same sequence was used in all our experiments because streaming performance is independent of video content and only the bitrate is relevant.

The percentage of CPU utilization was calculated as the ratio of CPU time (user and system) to wall-clock time. CPU time was measured with the Unix time(1) command and wall-clock time with a testing script. No other user processes were running on the computers during the benchmarking. All tests were repeated 100 times and the results were averaged.

The benchmarked applications were not able to utilize entire bandwidth of our 10 Gbps test network with packet sizes that conform to the Ethernet frame size of 1500 bytes. Therefore, an upper limit for throughput was measured by

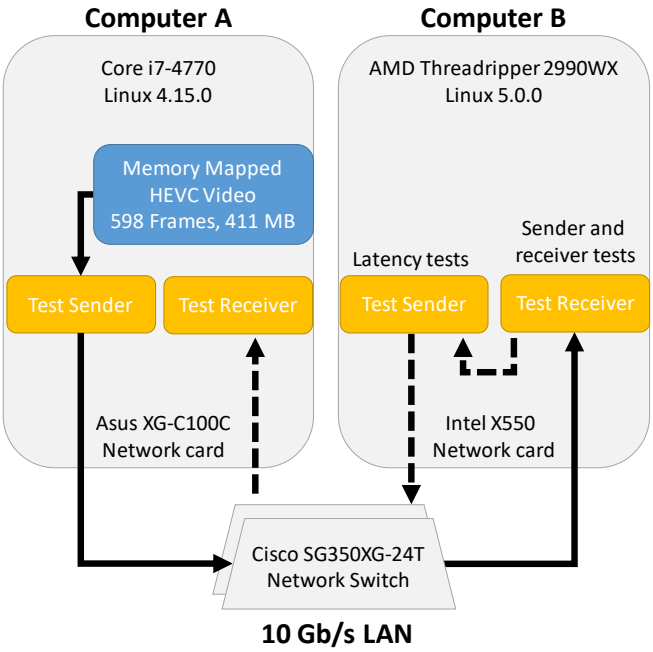

Fig. 3. Experimental setup for goodput and latency measurements.

TABLE II. Linux Variables for Both Computers

Variable	Value
Socket Write Buffer Default Size	40 MB
Socket Write Buffer Maximum Size	40 MB
TX Queue Length	10000
Socket Read Buffer Default Size	40 MB
Socket Read Buffer Maximum Size	40 MB
RX Queue Length	10000

TABLE III. FFmpeg Options Used for Measurements

Instance	Variable	Value
Both	avioflags	direct
Sender	fflags	flush_packets
Sender	max_interleave_delta	1000 µs
Receiver	fflags	nobuffer
Receiver	probesize	32 bytes
Receiver	fpsprobesize	2

sending 500 MB of data in 1.5 KB UDP packets from the Computer *A* to *B* using a script available online[1]. According to our measurements, the maximum throughput was 5.64 Gb/s that equals streaming the 4K30p video at 1033 *frames per second* (*fps*). Thus, a transfer speed of 1000 fps was set as an upper limit in our experiments. This would roughly correspond to transmitting 16K (15360×8640) video at 60 fps.

Table II tabulates the parameters for the Linux Network Stack. They were adjusted to take full advantage of the 10 Gbps network bandwidth [28] in our tests. In addition, to prepare the libraries under test for the large amount of data, the socket buffers were increased to 40 MB in all of them.

FFmpeg and LIVE555 were also consistently configured for a fair comparison. Latency and frame buffering of FFmpeg was minimized with 'AVFormatContext' options listed in Table III. The flags were set in an effort to reduce the buffering and analysis tasks of FFmpeg and thereby minimize their impact on latency and goodput.

[1] https://github.com/ultravideo/rtp-benchmarks

Correspondingly, 'OutPacketBuffer::maxSize' and 'fReceiverBuffer' were set to 40 MB in LIVE555. Because it uses a pull-based input method, we needed to control the input frame rate from our test system by setting 'fDurationInMSecs' to zero. It forces LIVE555 to query the frames immediately and limit the speed at which it receives frames by sleeping the specified frame interval. Moreover, LIVE555 lacked SCL so we used the SCL implementation from FFmpeg to divide the bitstream into NAL units.

All in all, two separate test cases were conducted. In the first test case, the goodput and frame loss were evaluated as a function of data transfer rate. The second test case measured round-trip latency. Altogether, five configurations were benchmarked: 1) an unoptimized uvgRTP; 2) uvgRTP with SCL optimization; 3) uvgRTP with SCL and SCC optimizations; 4) FFmpeg; and 5) LIVE555.

B. Goodput, CPU Utilization, and Frame Loss Evaluation

The data flow of this test case is shown by the solid line in Fig 3. Computers *A* and *B* were used as the sender and the receiver, respectively. The input frame rate was gradually increased from 100 fps to 1000 fps in steps of 100 fps.

Fig. 4 (a) illustrates the receiver goodputs for the benchmarked configurations as a function of frame rate. At lower rates, the goodput of the unoptimized uvgRTP increases practically linearly until 700 fps and remains almost stable thereafter. It reaches the peak goodput of 4.2 Gb/s at 900 fps. The inclusion of the SCL optimization extends the linear growth up to 5.0 Gb/s. Furthermore, enabling the SCC optimization makes the growth almost linear over the entire test range. In this case, the maximum goodput of 5.2 Gb/s is reached at 1000 fps. Correspondingly, the sender goodput can be computed by summing up the receiver goodput and data from lost frames.

Fig. 4 (b) plots CPU utilization for a single core at the sender end. The unoptimized uvgRTP and uvgRTP with the optimized SCL reach 100% CPU utilization and thereby become CPU bound at 700 fps and 900 fps, respectively. When both SCL and SCC optimizations are on, uvgRTP achieves the highest performance at 1000 fps but CPU utilization is also getting close to 100% in this case. Hence, the CPU can also become a bottleneck when streaming 4K video at high frame rates in a 10 Gb/s network.

Fig. 4 (c) depicts the frame losses for each configuration. In this study, we considered a frame loss of under 2.5% acceptable. The unoptimized uvgRTP is able to keep the frame loss below 1% over the entire test range. The same holds when the SCL optimization is on but not with the SCC optimization that incurs increasing frame losses the closer to the network limit it gets. It crosses the 2.5% boundary at 800 fps, which corresponds to the maximum acceptable goodput of 3.8 Gb/s. This high frame loss is caused by the bursty nature of the SCC optimization while sending. Therefore, we prefer disabling the SCC optimization at frame rates close to the network limit.

In our tests, uvgRTP with the SCL optimization was shown to achieve 4.4× as high goodput as FFmpeg that is able to reach a goodput of 1.1 Gb/s at 200 fps with a frame loss of less than 2.5%. However, at 300 fps and above, the frame loss rises above 4 % because FFmpeg is not able to empty the Socket Read Buffer fast enough. Increasing the buffer size further might help mitigate the loss, but the needed buffer size would depend on the amount of incoming data and the processing speed of FFmpeg which are typically unknown. The low CPU usage of 65% may also partially explain the limited goodput.

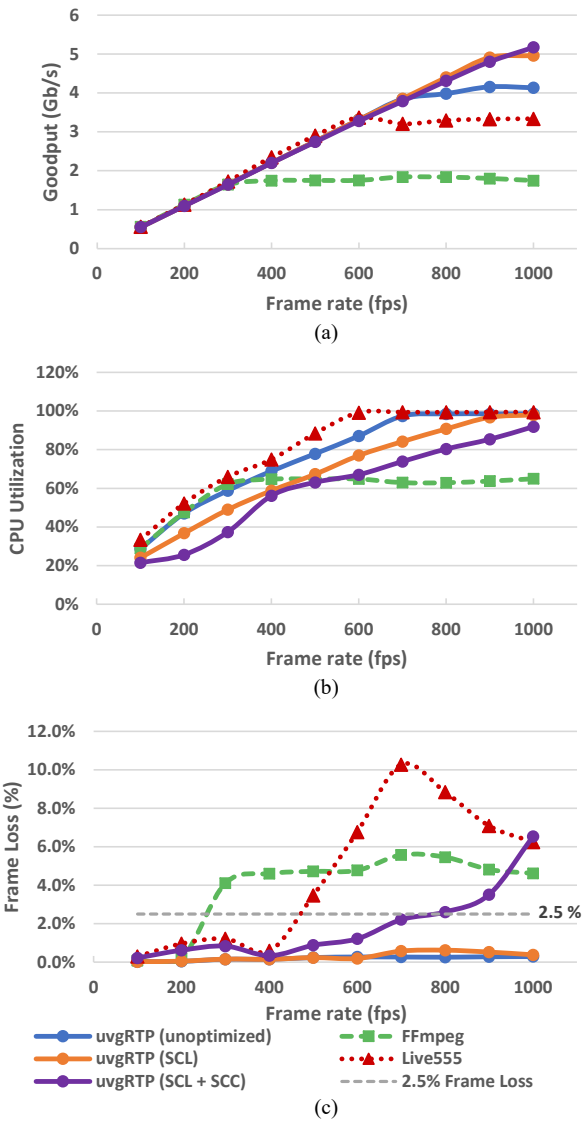

Fig. 4. Performance comparison of uvgRTP, FFmpeg, and LIVE555. (a) Goodput. (b) CPU utilization. (c) Frame loss.

LIVE555 is able to reach a goodput of 2.3 Gb/s without exceeding the 2.5% frame loss at 400 fps. The frame loss starts to increase at 500 fps and peaks at 700 fps. If the frame loss was eliminated, LIVE555 would be able to reach 3.4 Gb/s goodput. When adhering to the acceptable frame loss, uvgRTP with the optimized SCL is able to achieve 2.1× as high goodput as LIVE555.

C. Latency Evaluation

The second test case benchmarked the round-trip latencies of uvgRTP, FFmpeg, and LIVE555. They were measured by sending the HEVC video sequence from the Computer *A* to *B* and back along the path marked with solid and dashed lines in Fig. 3, respectively. The tests were performed at 30 fps because it is a widely used frame rate in real-time streaming. In addition, frame losses are smaller at low frame rates.

Table IV reports the round-trip latencies (in ms) of uvgRTP, FFmpeg, and LIVE555. The latencies were separately measured for inter and intra frames [27] of the

TABLE IV. ROUND-TRIP LATENCIES OF UVGRTP, FFMPEG, AND LIVE555

Configuration	Intra (ms)	Inter (ms)	Average (ms)	Failed test runs
uvgRTP (unoptimized)	15.0	7.9	8.0	4 %
uvgRTP (SCL)	14.2	7.0	7.1	5 %
uvgRTP (SCL + SCC)	12.9	6.3	6.4	42 %
FFmpeg	108.3	80.6	81.0	52 %
LIVE555	43.1	35.9	36.0	62 %

applied HEVC test sequence as well as their averages. The last column tabulates the shares of failed tests out of the conducted 100 test runs. A test run, with one or more lost packets, was considered failed and excluded from the reported results because a lost packet is more likely to belong to a large frame and thus reduce the average frame latency.

According to our results, the unoptimized uvgRTP achieves the average round-trip latency of 8.0 ms. The SCL optimization reduces it by 0.9 ms and the SCC optimization further by 0.7 ms to 6.4 ms. Our library is able to reduce the latency to a fraction over the competing approaches, i.e., it has 92.1% and 82.3% lower latency than FFmpeg and LIVE555, respectively. The uvgRTP with the SCL and SCC optimizations is therefore considered the leading RTP implementation, especially for low-latency applications.

VIII. CONCLUSIONS

This paper introduced a novel open-source RTP library called uvgRTP with built-in support for HEVC. According to our experiments, the current implementation of uvgRTP is capable of transferring HEVC video at 5.0 Gb/s over a 10 Gb/s LAN with an average round-trip latency of 6.4 ms. Our solution outperforms the FFmpeg framework with over quadruple the goodput and 92.1% lower latency. It also more than doubles the goodput over that of LIVE555 multimedia streaming library with 82.3% lower latency. These substantial performance improvements make uvgRTP a potential candidate for a plethora of media applications dealing with RTP streaming of 4K HEVC video.

Furthermore, most of the introduced optimizations are not tailored solely to HEVC but they can be deployed with various payload formats. For example, all the optimizations are anticipated to be compatible with the next-generation *Versatile Video Coding* (*VVC/H.266*) standard [29]. The uvgRTP architecture also allows easy addition of QoS improvements such as RTP packet retransmission (RFC 4588) [30] or Flexible Forward Error Correction (RFC 8627) [31].

In the future, our library will be expanded to work with VVC. Additionally, providing support for fast Secure RTP and RTP header compression is in our roadmap.

REFERENCES

[1] Cisco, Cisco Visual Networking Index: Forecast and Trends, 2017-2022, Dec. 2018.

[2] *Advanced Video Coding for Generic Audiovisual Services,* document ITU-T Rec. H.264 and ISO/IEC 14496-10 (AVC), ITU-T Rec. H.264 and ISO/IEC, 2009.

[3] *High Efficiency Video Coding,* document ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC), ITU-T Rec. H.264 and ISO/IEC, 2013.

[4] J. Vanne, M. Viitanen, T. D. Hämäläinen, and A. Hallapuro, "Comparative rate-distortion-complexity analysis of HEVC and AVC video codecs," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, Dec. 2012, pp. 1885-1898.

[5] T. K. Tan, M. Mrak, V. Baroncini, and N. Ramzan, "Report on HEVC compression performance verification testing," *document JCTVC-Q1011*, Valencia, Spain, Mar.-Apr. 2014.

[6] IETF RFC 3550 RTP: A Transport Protocol for Real-Time Applications, IETF, July 2003. [Online]. Available: https://tools.ietf.org/html/rfc3550

[7] IETF RFC 7798 RTP Payload Format for High Efficiency Video Coding (HEVC), IETF, Mar. 2016. [Online]. Available: https://tools.ietf.org/html/rfc7798

[8] ccRTP. [Online]. Available: https://www.gnu.org/software/ccrtp/

[9] JRTPLIB. [Online]. Available: https://github.com/j0r1/JRTPLIB

[10] libre. [Online]. Available: http://www.creytiv.com/re.html

[11] PJSIP. [Online]. Available: https://www.pjsip.org/

[12] GoRTP. [Online]. Available: https://github.com/wernerd/GoRTP

[13] oRTP. [Online]. Available: https://www.linphone.org/technical-corner/ortp

[14] libjitsi. [Online]. Available: https://github.com/jitsi/libjitsi

[15] Restcomm Media-Core [Online]. Available: https://github.com/RestComm/media-core

[16] LIVE555 [Online]. Available: http://www.live555.com/

[17] FFmpeg [Online]. Available: https://www.ffmpeg.org/

[18] GStreamer [Online]. Available: https://gstreamer.freedesktop.org

[19] Linphone [Online]. Available: https://www.linphone.org

[20] Jitsi [Online]. Available: https://jitsi.org/

[21] Restcomm [Online]. Available: https://www.restcomm.com/

[22] VLC [Online]. Available: https://www.videolan.org/vlc/

[23] Opus [Online]. Available: https://tools.ietf.org/html/rfc6716

[24] M. Rajagopalan, S. K. Debray, M. A. Hiltunen and R. D. Schlichting, "System call clustering: a profile-directed optimization technique," Technical Report, The University of Arizona, June 2002.

[25] zeroInWord [Online]. Available: https://graphics.stanford.edu/~seander/bithacks.html#ZeroInWord.

[26] 802.3-2018 - IEEE Standard for Ethernet, Aug. 2018. [Online]. Available: https://standards.ieee.org/standard/802_3-2018.html

[27] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, Dec. 2012, pp. 1649-1668.

[28] M. J. Christensen and T. Richter, "Achieving reliable UDP transmission at 10 Gb/s using BSD socket data acquisition systems," arXiv:1706.00333 [physics.ins-det], June 2017. [Online]. Available: https://arxiv.org/abs/1706.00333

[29] ITU, "New 'Versatile Video Coding' standard to enable next-generation video compression," July 2020. [Online]. Available: https://www.itu.int/en/mediacentre/Pages/pr13-2020-New-Versatile-Video-coding-standard-video-compression.aspx

[30] IETF RFC 4588 RTP Retransmission Payload Format, IETF, July 2006. [Online]. Available: https://tools.ietf.org/html/rfc4588

[31] IETF RFC 8627 RTP Payload Format for Flexible Forward Error Correction (FEC), IETF, July 2019. [Online]. Available: https://tools.ietf.org/html/rfc8627