**3GPP TSG SA WG4 Meeting #129-e *S4-241450***

**Online August 19 2024- August 23 2024 in revision of S4aR240043**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *CR-Form-v12.3* | | | | | | | | |
| **Pseudo CHANGE REQUEST** | | | | | | | | |
|  | | | | | | | | |
|  | **26.822** | **CR** | **-** | **rev** | **-** | **Current version:** | **0.1.1** |  |
|  | | | | | | | | |
| *For* [***HE******LP***](http://www.3gpp.org/3G_Specs/CRs.htm#_blank)*on using this form: comprehensive instructions can be found at* [*http://www.3gpp.org/Change-Requests*](http://www.3gpp.org/Change-Requests)*.* | | | | | | | | |
|  | | | | | | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Proposed change affects:*** | UICC apps |  | ME |  | Radio Access Network |  | Core Network |  |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | |
| ***Title:*** | [FS\_5G\_RTP\_PH2] Dynamic Traffic Characteristics of RTP Senders. | | | | | | | | | |
|  |  | | | | | | | | | |
| ***Source to WG:*** | Huawei, HiSilicon | | | | | | | | | |
| ***Source to TSG:*** | SA WG4 | | | | | | | | | |
|  |  | | | | | | | | | |
| ***Work item code:*** | FS\_5G\_RTP\_Ph2 | | | | |  | ***Date:*** | | | 7-8-2024 |
|  |  | | | |  | |  | | |  |
| ***Category:*** | **B** |  | | | | | ***Release:*** | | | 19 |
|  | *Use one of the following categories:* ***F*** *(correction)* ***A*** *(mirror corresponding to a change in an earlier release)* ***B*** *(addition of feature),* ***C*** *(functional modification of feature)* ***D*** *(editorial modification)*  Detailed explanations of the above categories can be found in 3GPP [TR 21.900](http://www.3gpp.org/ftp/Specs/html-info/21900.htm). | | | | | | | | *Use one of the following releases: Rel-8 (Release 8) Rel-9 (Release 9) Rel-10 (Release 10) Rel-11 (Release 11) … Rel-17 (Release 17) Rel-18 (Release 18) Rel-19 (Release 19)  Rel-20 (Release 20)* | |
|  |  | | | | | | | | | |
| ***Reason for change:*** | | FS\_5G\_RTP KI #12 deals with enhancements for data burst marking for 5G RTP.  One topic under study in this key issue:  - Identify and document the way RTP senders can generate data bursts e.g.  - WebRTC paced sending implementation, including the different configurations of WebRTC paced sending.  - Other common RTP implementations or libraries that are commonly used  This contribution provides additional information about the WebRTC paced sending implementation. In addition, it provides information about traffic patterns generated by popular RTP sender implementations in a conversational communications setting. For both a webRTC sender and a open source gstreamer based RTP sender implementation traffic is captured and analyzed for dynamic traffic characteristics.  In addition 2 popular servers for RTP/RTSP are tested. | | | | | | | | |
|  | |  | | | | | | | | |
| ***Summary of change:*** | | A solution to KI#12 for identifying dynamic traffic characteristics in RTP Sender   * WebRTC sender in Google Hangout scenario * Gstreamer implementation using RTP sender pipeline and receiver using a webcam capture and rendering at the receiver * Testing of 2 popular application servers for real time communication wowza and MTX media server | | | | | | | | |
|  | |  | | | | | | | | |
| ***Consequences if not approved:*** | | No information on dynamic traffic characteristics from popular RTP sender implementations is included in the report, KI objectives not completed. | | | | | | | | |
|  | |  | | | | | | | | |
| ***Clauses affected:*** | | 2, 6.X (new clause) | | | | | | | | |
|  | |  | | | | | | | | |
|  | | **Y** | **N** |  | | | |  | | |
| ***Other specs*** | |  |  | Other core specifications | | | | TS/TR ... CR ... | | |
| ***affected:*** | |  |  | Test specifications | | | | TS/TR ... CR ... | | |
| ***(show related CRs)*** | |  |  | O&M Specifications | | | | TS/TR ... CR ... | | |
|  | |  | | | | | | | | |
| ***Other comments:*** | |  | | | | | | | | |
|  | |  | | | | | | | | |
| ***This CR's revision history:*** | | * Revision 1 * Add results for server side implementation wowza, simple RTSP server * Add information about the network in between * Restructured the results webrtc, peer 2 peer and server based. * Add gstreamer results in including audio PCM packets * Paced sender, check results when the bandwidth of network gets throttled/limited for webRTC to test the paced sender | | | | | | | | |

|  |
| --- |
| CHANGE 1 |

# 2 References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

- References are either specific (identified by date of publication, edition number, version number, etc.) or non‑specific.

- For a specific reference, subsequent revisions do not apply.

- For a non-specific reference, the latest version applies. In the case of a reference to a 3GPP document (including a GSM document), a non-specific reference implicitly refers to the latest version of that document *in the same Release as the present document*.

[1] 3GPP TR 21.905: "Vocabulary for 3GPP Specifications".

[2] 3GPP TS 26.522: "5G Real-time Media Transport Protocol Configurations".

[3] 3GPP TS 23.501: "System architecture for the 5G System (5GS)".

[4] IETF RFC 8872: "Guidelines for Using the Multiplexing Features of RTP to Support Multiple Media Streams".

[5] IETF RFC 5761: "Multiplexing RTP Data and Control Packets on a Single Port".

[6] 3GPP TR 23.700-70: "Study on architecture enhancement for Extended Reality and Media service (XRM); Phase 2".

[7] IETF RFC 8285 (2017): "A General Mechanism for RTP Header Extensions", D. Singer, H. Desineni, R. Even.

[8] IETF RFC 3711: "The Secure Real-time Transport Protocol (SRTP)".

[9] IETF RFC 9335: "Completely Encrypting RTP Header Extensions and Contributing Sources".

[10] IETF RFC 6904 (2013): "Encryption of Header Extensions in the Secure Real-time Transport Protocol (SRTP)", J. Lennox.

[11] IETF RFC 8402 (2018): "Segment Routing Architecture".

[12] IETF RFC 791 (1981): "Internet Protocol".

[13] IETF RFC 5109: "RTP Payload Format for Generic Forward Error Correction (ULP FEC): Uneven Level Protection, different redundancies for different packets with different importance".

[14] IETF RFC 8627: "RTP Payload Format for Flexible Forward Error Correction (Flex FEC): flexible FEC".

[15] IETF RFC 6681: "Raptor Forward Error Correction (FEC) Schemes for FECFRAME: FEC scheme based on the Raptor".

[16] IETF RFC 6865: "Simple Reed-Solomon Forward Error Correction (FEC) Scheme for FECFRAME: FEC scheme based on Reed-Solomon".

[17] IETF RFC 5053: "Raptor Forward Error Correction Scheme for Object Delivery".

[18] IETF RFC 6330: "RaptorQ Forward Error Correction Scheme for Object Delivery".

[19] IETF RFC 6363: “Forward Error Correction (FEC) Framework”.

[20] IETF RFC 8854: “WebRTC Forward Error Correction Requirements”.

[21] 3GPP TR 38.340: "Study on User Equipment (UE) power saving in NR".

[22] IETF RFC 8298: "Self-Clocked Rate Adaptation for Multimedia".

[23] Enhancing Video Network Resiliency With LTR and RS Code | At Scale Conferences, available online: https://atscaleconference.com/enhancing-video-network-resiliency-with-ltr-and-rs-code/

[24] P. Aggarwal et al., [2304.03732] Enabling immersive experiences in challenging network conditions (arxiv.org)

[25] Nvidia GeForce Now, Video FEC for WebRTC presentation 17 Nov. 2022, available online: https://www.youtube.com/watch?v=igm7QkqxHqk&ab\_channel=KrankyGeek

[26] Holmer S., et al., Handling Packet Loss in WebRTC, 2013 IEEE International Conference on Image Processing, available online: <https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/41611.pdf>.

[27] A Google Congestion Control Algorithm for Real-Time Communication, draft-ietf-rmcat-gcc-02, 2016.

[28] WebRTC source code: https://source.chromium.org/chromium/chromium/src/+/main:third\_party/webrtc, retrieved May 1, 2024.

[29] IETF RFC 8698: "Network-Assisted Dynamic Adaptation: A Unified Congestion Control Scheme for Real-Time Media", 2020.

[30] Self-Clocked Rate Adaptation for Multimedia, draft-johansson-ccwg-rfc8298bis-screamv2-00, 2024.

[31] IETF RFC 4588: "RTP Retransmission Payload Format".

[32] 3GPP TS 26.114: "IP Multimedia Subsystem (IMS); Multimedia telephony; Media handling and interaction".

[YY] [GStreamer: open source multimedia framework](https://gstreamer.freedesktop.org/), Online:

https://gstreamer.freedesktop.org/.

[YY1] Wowza Streaming Engine:

https://www.wowza.com/

[YY2] Media MTX: Ready-to-use SRT / WebRTC / RTSP / RTMP / LL-HLS media server and media proxy that allows to read, publish, proxy, record and playback video and audio streams:

http://github.com/blueviron/mediamtx

[YY3] Adobe RTMP Specification (veriskope.com):

https://rtmp.veriskope.com/docs/spec/

|  |
| --- |
| CHANGE 2 (ALL NEW TEXT) |

## 6.X Solution X: Dynamic Traffic Characteristics of RTP Sender implementations

### 6.X.1 General

This is a solution to KI #12 Enhancements of Data Burst Marking.

A data burst indicates a set of multiple PDUs generated and sent in a short period of time as defined in clause 3.1 of TS 23.501 [3]. Data burst is a common transmission characteristic in communication networks.

The source of traffic in this study is Real Time Transport Protocol as used for conversational or Extended Reality applications.

The data burst transmission could be beneficial for the 5GS network, e.g., power saving and efficient radio resource management. In Release 18, the End of Data Burst indication has been introduced to enable the UE power saving in the 5GS, i.e., the NG-RAN node can configure to move a UE into CDRX for power saving after transmitting the end PDU of the data burst. In Release 19, the data burst size has been concluded to enable the RAN radio resource management as described in clause 8.5 of TR 23.700-70 [6].

This contribution aims to observe dynamic traffic characteristics from popular RTP senders and possible implications for low latency and real-time communication for usage and support in 5G RTP.

In this study we explore:

- The open source WebRTC implementation of RTP Sender in a browser implementation

- The cross platform GStreamer [YY] implementation of RTP senders for peer-to-peer (P2P) transmission

- The sending and receiving measured of two popular real-time streaming servers.

### 6.X.2 WebRTC and Paced Sender implementation

#### 6.X.2.1 WebRTC and Paced Sender implementation

WebRTC (Web Real-Time Communication) [28] is an open-source project providing web browsers and mobile applications with real-time communication (RTC) via application programming interfaces (APIs). It allows audio and video communication and streaming to work inside web pages by direct peer-to-peer communication, eliminating the need to install plugins.

It can be implemented in web browsers and is a likely source of real-time conversational RTP traffic in practice.

Based on the documentation available [28] from November 2021 in the WebRTC source code this clause describes the way packets may be generated and transmitted in a WebRTC environment.

WebRTC does not send out data packets generated by an encoder directly, instead a pacing module is used that limits and smoothens the packet train that is sent out to the network.

The idea is that, say for example, a 30 fps stream of 6 MBps is sent to the network, in the ideal case this would result in equally sized frames of around 25 kiloBytes in around 21 equally sized packets. While in practice, over a 1 second sliding window the average bit-rate of 6Mbps might be measured, on a shorter timescale there may be periodic bursts that overshoot this average bit-rate with a much larger instantaneous bit-rate. One of the causes could be a sudden movement in a frame.

To avoid potential impact of these cases on the network and application, the WebRTC solution introduces the paced sender. The paced sender introduces a buffer in which the media packets are queued before being sent out. This avoids, for example, that large video frames are split into a large burst of packets that will disrupt other streams such as audio streams that are critical for intelligibility. WebRTC uses multiplexed RTP transport with different streams sent on the same port and ip tuple, thus potentially such an overshoot in video packets could stop other packets from being sent out in time as the socket may be blocked when transmitting the video packets.

NOTE: The standard approach in RTP is to use different ports for different media streams, but in practice this could have some drawbacks as this increases resource usage and may complicate NAT traversal. In this case, the prioritization would depend on the operating system implementation.WebRTC allows multiplexing audio and video on the same port by establishing a relationship between the media lines included in the SDP using the attribute a=group:BUNDLE.

For sending out the media, a leaky bucket algorithm is used for pacing them on to the network. In this implementation packets are inserted into a buffer first before being transmitted. The buffer contains separate first in first out (FIFO) queues for each media type/track, such that audio can for example be prioritized over video. Equal priority streams can be sent in round-robin fashion, avoiding these streams from blocking each another.

In a leaky bucket the rate of packets that is sent out is limited, potentially leading to a buffering of the input in case the input is larger than the output rate. A leaky bucket is one way to limit the burst sizes in traffic sent to the network.

In WebRTC this output rate is called the *pacing\_rate.*

The typical way a packet is generated and paced on to the network in WebRTC is as follows:

1. Media frame is packetized into RTP packets
2. The packets are sent for scheduling (transmission)
3. The pacer is called to enqueue the packet
4. The packets stay in the queue until the pacer finds a scheduled moment to transmit them out based on the leaky bucket algorithm
5. At the time that the packet is scheduled, the packet is forwarded to the RTP module for final timestamping
6. The packet is sent on the low level interface (i.e. the UDP socket), and it is now out of scope of the WebRTC implementation

Asynchronous to this, the WebRTC implementation tries to estimate the send bandwidth in order to set the *pacing\_rate,* i.e. the rate at which packets are sent to the network and secondly the *padding\_rate*, the bit rate (if any) used to send padding in case no packets are available to be sent out*.* Figure y1 illustrates the WebRTC transmission flow.

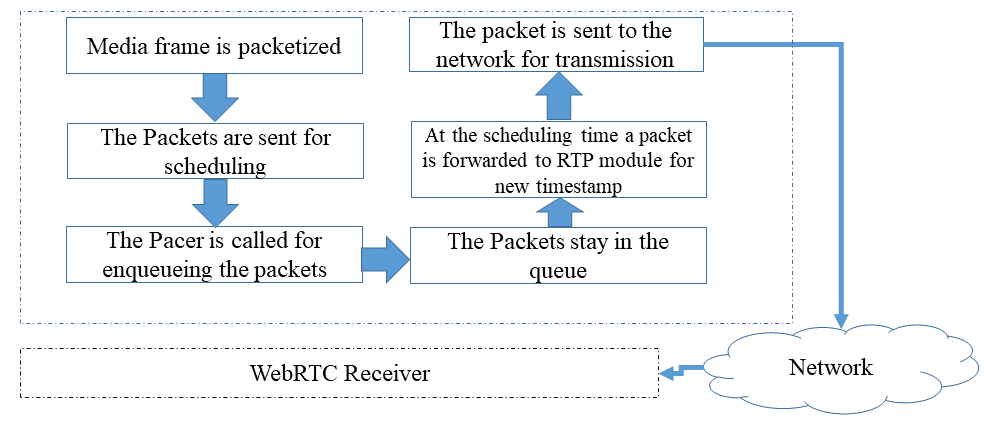


Figure y1 Packet processing in WebRTC paced sender of an RTP packet

The pacer of WebRTC prioritizes based on different criteria:

a) Packet Type, with most to least prioritized:

1. Audio
2. Retransmission
3. Video and FEC
4. Padding

b) Based on the enqueueing order.

When the queues are empty the implementation will aim to generate padding frames and send them out at the *padding\_rate*. In some cases the *pacer\_rate* may be overridden and ignored, when there is a significant encoding overshoot for example. In addition WebRTC implementation enables setting the maximum time a packet can stay in a queue using a function (*setQueueTimeLimit(TimeDelta limit))*.

Bandwidth estimation in the WebRTC implementation is done by bandwidth probing, i.e. a cluster of packets is requested to be transmitted over the network to gauge if this will lead to increased delay and or loss. The implementation provides a function to do this measurement, enabling one to use this information to update the *pacing\_rate*.

The WebRTC implementation has defined additional API function to monitor the states and statistics of the pacer.

#### 6.X.2.2 WebRTC and Paced Sender evaluation

To collect traffic characteristics, the following setup is deployed:

- In this case a call is started on a Microsoft edge browser and Wireshark is used to collect the traffic trace. The machine initiating the call is a Lenovo laptop with 16GB of RAM running windows 11 operating system. The network is a wifi connection on 2.4 Ghz band setup by a Samsung Galaxy A54 5G connected to a 5G network using wifi hotspot feature.

Figure y2 illustrates the traffic pattern from a webRTC sender that joined a google hangout call on Machine A. This stream has a lower bit-rate as compared to the previous experiment. The traffic is clearly separated in burst that corresponds to the frame transmission.

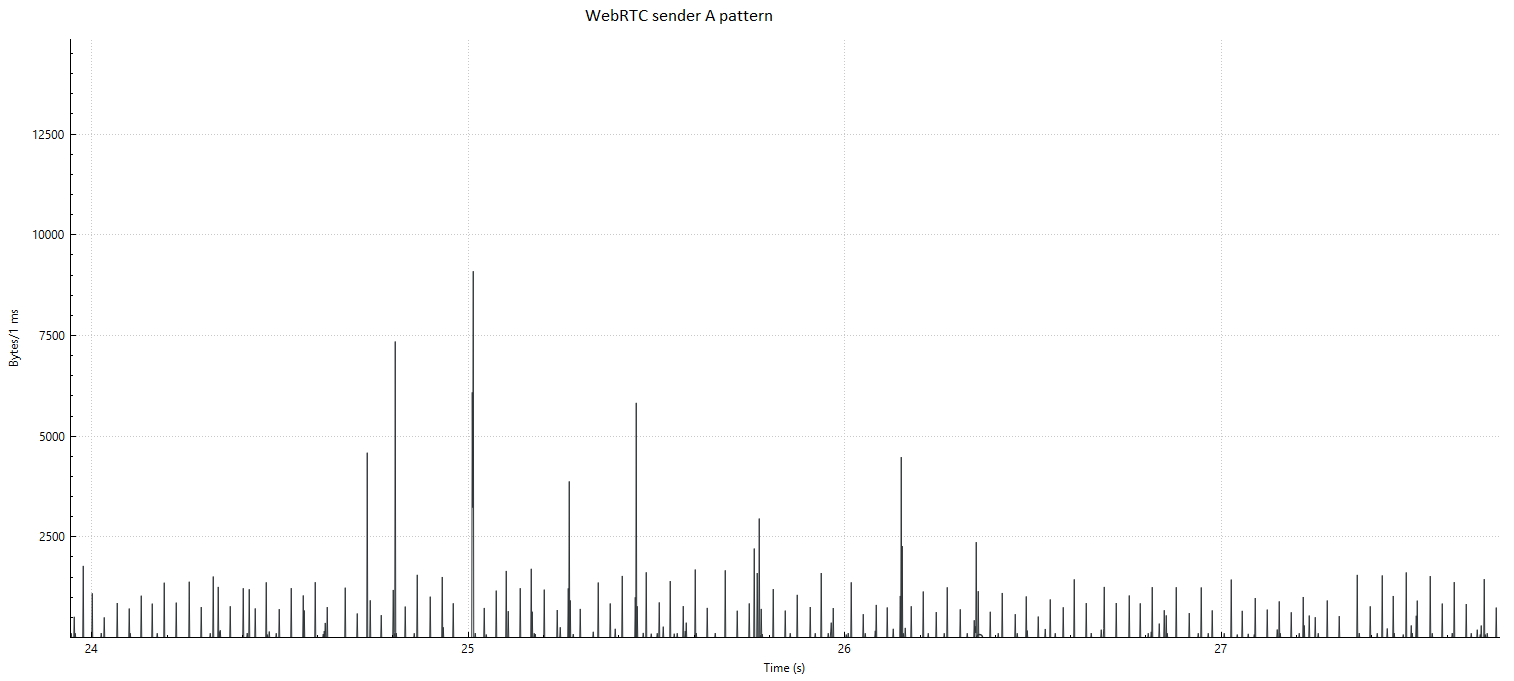


Figure y2a Sending pattern from WebRTC in a Google hangout call with video only

In a second experiment, we use a machine running Ubuntu 24.4 with 8GB or RAM and Intel core i5 processor. The Ubuntu machine/linux kernel enables throttling of the bandwidth and latency. The communication is via a wifi and a 5G hotspot.

Linux traffic control was used using a token bucket filter (tbf) to slow down the data rate to 200kbit and only enable 16kbit bursts. The results are shown in Figure y2b. We see the magnitude of the peaks is reduced as the hangout video quality drops after the traffic control filter is enabled. However the general traffic characteristics look the same, there is no smoothening effect observed that may result from the paced sender. So it seems in this case the general rate control of hangout for the video encoding reacted quick enough to avoid critical network operation.

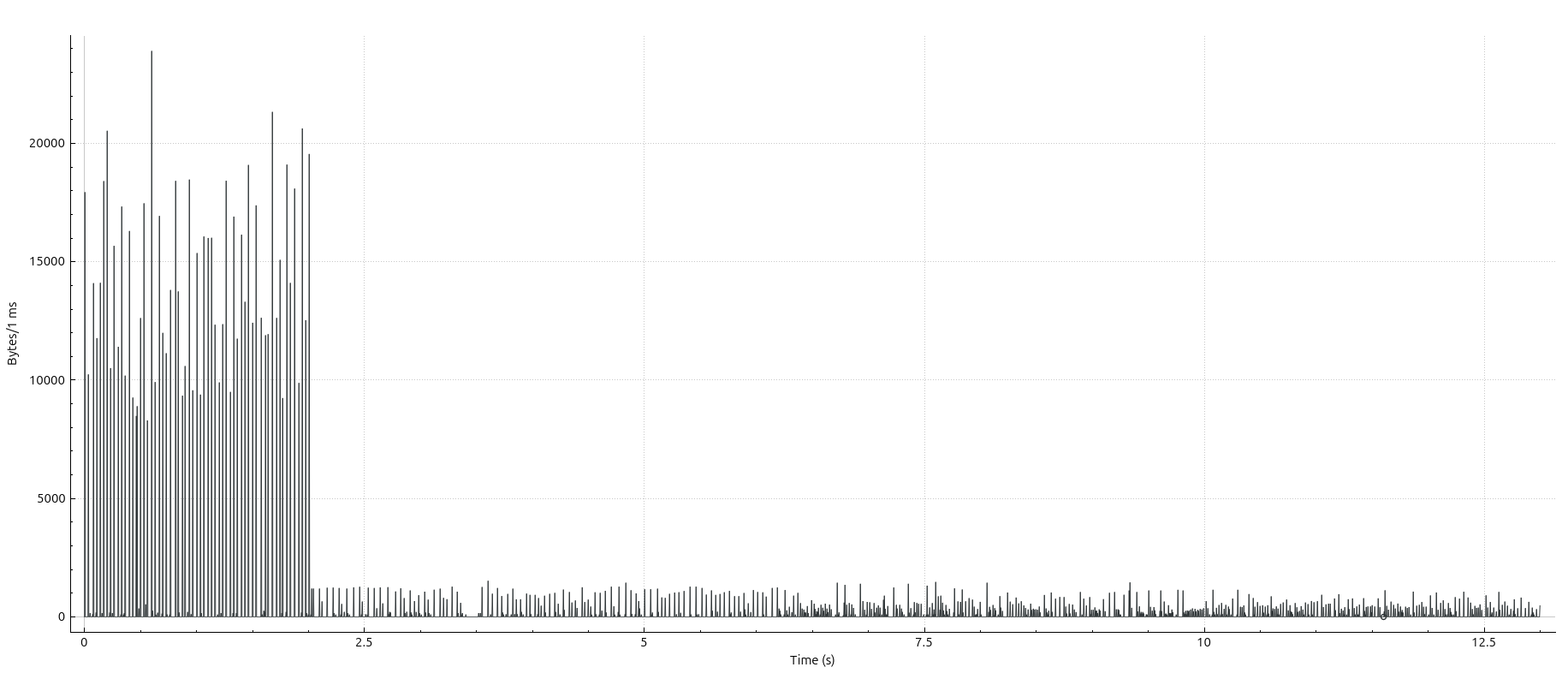


Figure y2b Resulting traffic characteristics at sender when the network bandwidth is throttled to 200kbit audio + video

### 6.X.3 GStreamer Multimedia Framework RTP implementation for P2P

#### 6.X.3.1 GStreamer Multimedia Framework RTP implementation details

GStreamer [YY] is a library for constructing graphs of media-handling components. It supports a range of applications from simple playback to streaming up to complex mixing and editing workflows.

Gstreamer works on all major operating systems as as Solaris, Unix etc.

It enables graph based pipeline construction and has a broad coverage of different multimedia technologies for codecs, protocols, file format encapsulations etc.

The GStreamer 1.x API has been stable since 2012 and the library is quite likely to be found as a sub-component in many practical media systems implementations, especially open source implementations.

In addition, the design of light weight data passing implies high performance and low latency, very applicable to real-time communications.

Due to the stability and long track record and wide deployment of this library, some sample pipelines utilizing the RTP features are developed in this study for collecting details on the traffic characteristics of this framework in simple RTP workflows.

The setup uses a camera followed by an H.264 encoder tuned for ultra low latency and optimal speed settings that is fed to an RTP pay-loader linked to a network sink. The receiver receives the RTP packets, decodes the streams and renders it to screen enabling a 1 way conversational application.

#### 6.X.3.2 Gstreamer RTP sender evaluation Video

In this section the RTP video sender and receiver are evaluated.

To collect traffic characteristics, the following setup is used:

- Two Machines A and B (Lenovo and HP, both AMD Ryzen 5 processor, 16GB RAM are connected over wifi network (2.4 GhZ) setup by a Galaxy A65 5G phone using hotspot function. Machine A runs the real time-frame capture, encoding packetization and UDP transmission (sender pipeline from the previous clause). Machine B receives the packets and depacketizes, decodes and renders the frame (receiver pipeline from the previous clause). Wireshark is used to collect the traffic data. In this setup a relatively high bit-rate of 16 MBps is used. GStreamer 1.24.5 is used for the experiment.

Figure 1x outlines the experimental setup.

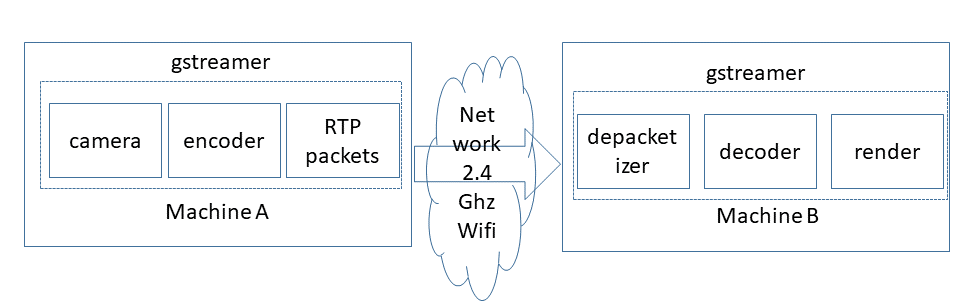


Figure 1x experimental setup flow

Figure y3 illustrates the sending data traffic pattern in bytes per millisecond on a logarithmic scale. It is clear that the traffic is bursty and that the bursts occur on the timespan of 1-3 milliseconds.

Figure y4 illustrates the receiving data traffic pattern in bytes per millisecond on a logarithmic scale. It is clear that the traffic bursty and that the burst occur in the timespan of 1-3 milliseconds, however compared to yy already a little bit of dispersion can be seen due to transmission and delays experienced in the transmission.

Table X demonstrates a snapshot from the data collected from the sender A, one interesting observation is that the inter burst time is not constant. Between some burst there is approximately 20 milliseconds of idle period, while for other bursts the inter burst time is about 40-45 milliseconds.

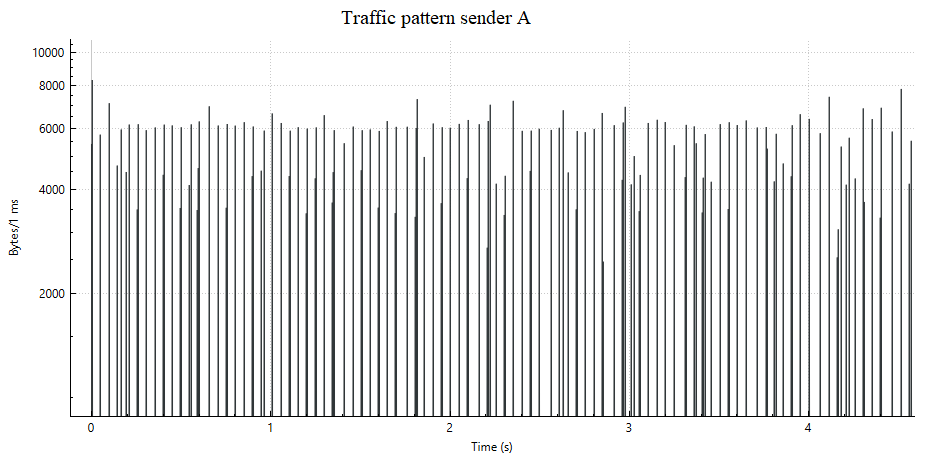


Figure y3 traffic pattern in gstreamer RTP sender on machine A

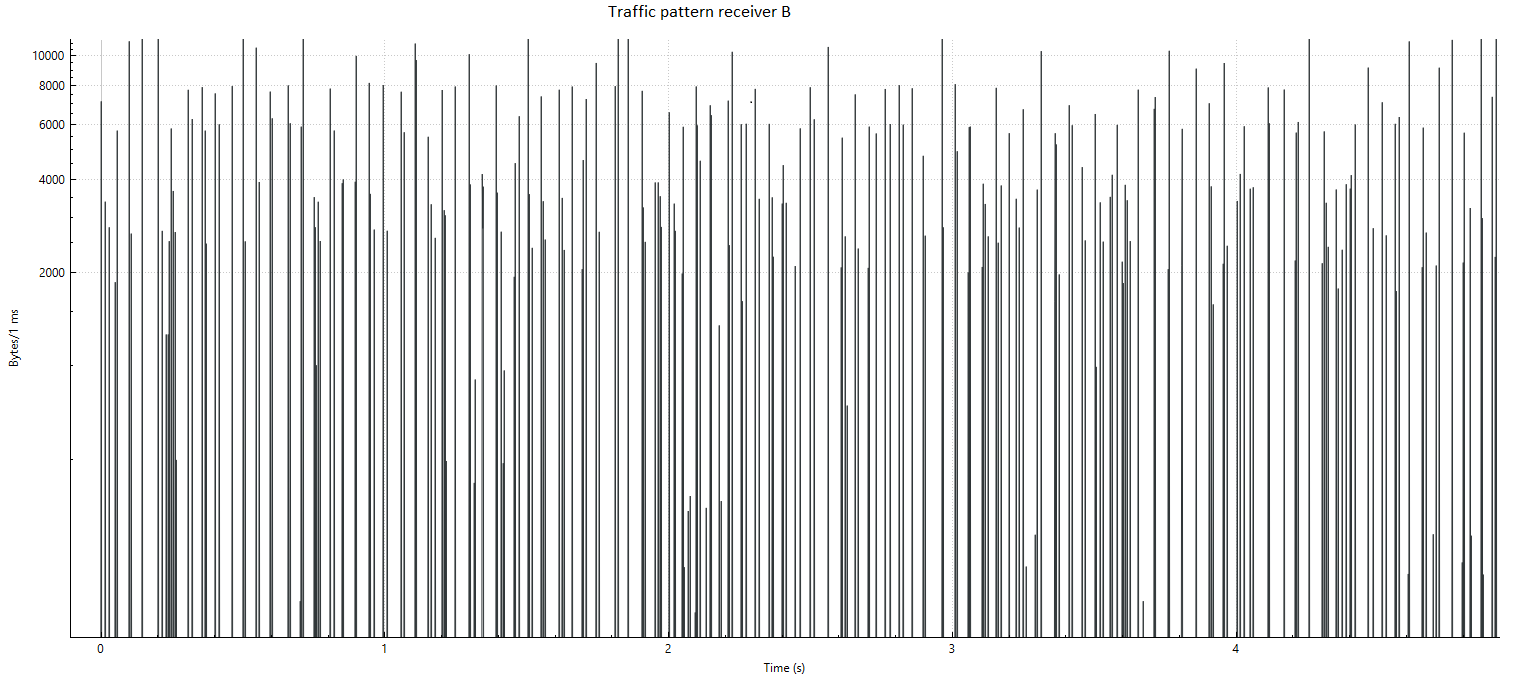


Figure y4 Traffic on machine B at the RTP receiver

Table x Snapshot of data from the sender, illustrating the inter-burst times

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| packet number | time | source ip | receiver ip |  | length |
| 108 | 0.192814 | 192.168.178.229 | 192.168.178.110 | UDP | 766 |
| 109 | 0.192830 | 192.168.178.229 | 192.168.178.110 | UDP | 767 |
| 110 | 0.192846 | 192.168.178.229 | 192.168.178.110 | UDP | 845 |
| 111 | 0.192859 | 192.168.178.229 | 192.168.178.110 | UDP | 520 |
| 112 | 0.192872 | 192.168.178.229 | 192.168.178.110 | UDP | 448 |
| 113 | 0.210483 | 192.168.178.229 | 192.168.178.110 | UDP | 56 |
| 114 | 0.210584 | 192.168.178.229 | 192.168.178.110 | UDP | 565 |
| 115 | 0.210627 | 192.168.178.229 | 192.168.178.110 | UDP | 985 |
| 116 | 0.210655 | 192.168.178.229 | 192.168.178.110 | UDP | 944 |
| 117 | 0.210687 | 192.168.178.229 | 192.168.178.110 | UDP | 1110 |
| 118 | 0.210716 | 192.168.178.229 | 192.168.178.110 | UDP | 1027 |
| 119 | 0.210742 | 192.168.178.229 | 192.168.178.110 | UDP | 758 |
| 120 | 0.210770 | 192.168.178.229 | 192.168.178.110 | UDP | 722 |
| 121 | 0.256040 | 192.168.178.229 | 192.168.178.110 | UDP | 56 |
| 122 | 0.256087 | 192.168.178.229 | 192.168.178.110 | UDP | 317 |
| 123 | 0.256104 | 192.168.178.229 | 192.168.178.110 | UDP | 416 |
| 124 | 0.256125 | 192.168.178.229 | 192.168.178.110 | UDP | 584 |
| 125 | 0.256143 | 192.168.178.229 | 192.168.178.110 | UDP | 812 |

#### 6.X.3.3 Gstreamer RTSP sender/receiver evaluation PCM audio + Video

In this section the RTP video + audio sender and receiver are evaluated.

The video is as in previous section, while the audio is transmitted as PCM audio.

To collect traffic characteristics, the following setup is used:

- Two Machines A and B (Lenovo and Asus, first with AMD Ryzen 5 processor and second with Intel i5, 16GB RAM/8GB RAM are connected over wifi network (2.4 GhZ) setup by a Galaxy A65 5G phone using hotspot function. Machine A runs the real time-frame capture, encoding packetization and UDP transmission (sender pipeline from the previous clause). Machine B receives the packets and depacketizes, decodes and renders the frame (receiver pipeline from the previous clause). Wireshark is used to collect the traffic data.

The results are shown in the figures below, the dynamic traffic characteristics bear similarity with those in the previous section, but the audio packets are much more frequent peaks with a smaller size.

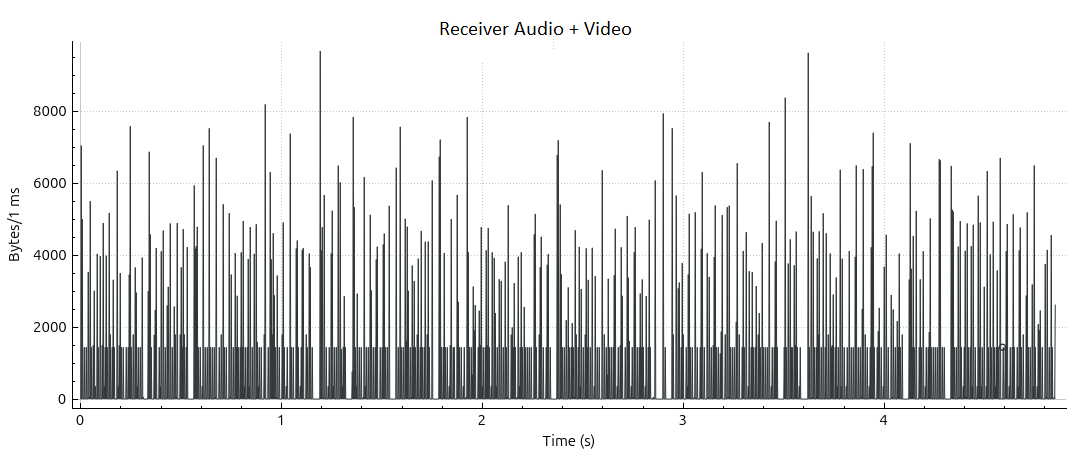


Figure 6.x.3.3.1

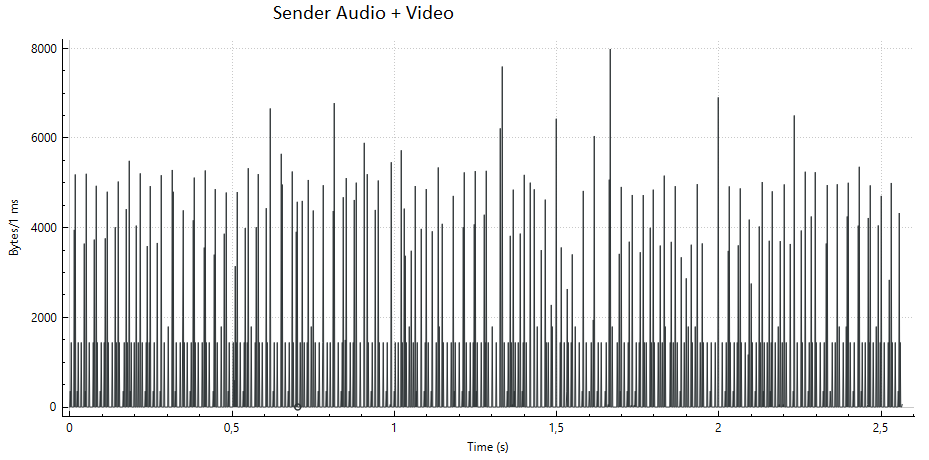


Figure 6.x.3.3.2

### 6.X.4 Server Side Senders

#### 6.x.4.1 Server sender implementation Details

In this clause we consider server implementations of RTP senders using RTSP or RTP.

Both a popular commercial server and a popular open source server implementation are evaluated.

The commercial server used for evaluation is Wowza Streaming Engine [YY1] (before wowza media server), one of the popular streaming servers that can support a wide range of transmission protocols. Wowza streaming engine is a Java application that runs as a service. The server can be used for a variety of live and on-demand streaming applications and it supports RTP/RTSP and low latency delivery.

The open source server used in this evaluation is the popular Media MTX (before called simple RTSP server) [YY2], Media MTX is a real time media server/proxy that can support different protocols, including RTSP and RTP, but also WebRTC and RTMP [YY3]. The github statistics on [YY2] show this as one of the most popular open source RTP/RTSP server implementations. It is possible, similar as in wowzo to ingest camera and other streams into the server for redistribution.

#### 6.x.4.2 Server side implementation evaluation

Wowza streaming engine version 4.8.27 was used deployed on a Lenovo laptop with 16GB or RAM with AMD Ryzen 5. Another laptop with Intel i7-6600 2.6 GhZ CPU and 16GB of RAM is used to receive the stream from the server. A camera stream is ingested over RTP to the wowza streaming engine using a gstreamer pipeline similar as the P2P sender in the previous section. In wowza streaming engine a predefined session description file is used to ingest the stream. The stream is distributed using RTP/RTSP. The machines are connected over a 2.4 GhZ wifi network setup by a Galaxy A54 5G phone using the hotspot function. The receiver laptop uses VLC media player 3.0.8 for receiving the stream using RTSP and RTP.

Media MTX version 1.8.24 was used deployed on a Lenovo laptop with 16GB or RAM with AMD Ryzen 5. Another laptop Microsoft surface with Intel i7-6600 2.6 GhZ CPU and 16GB of RAM was used to receive the stream. A camera stream is ingested over RTP to the wowza streaming engine using gstreamer, instead of RTP + SDP in this case WebRTC + WHIP was used for the ingest, which automizes the session description exchange needed for the ingest. The stream is distributed by Media MTX engine using RTP/RTSP. The machines are connected over a 2.4 GhZ wifi network setup by a Galaxy A54 5G phone using the hotspot function. The receiver laptop uses VLC media player 3.0.8 for receiving the stream over RTSP.

The flow of the experiment can be shown in Figure x2, gstreamer is used for the camera ingest to MTX Media or Wowza streaming engine, then the data is transmitted to the client by the server, and Video Lan Client (VLC) is used to receive the stream and play it back. In this scenario RTSP + RTP is used.

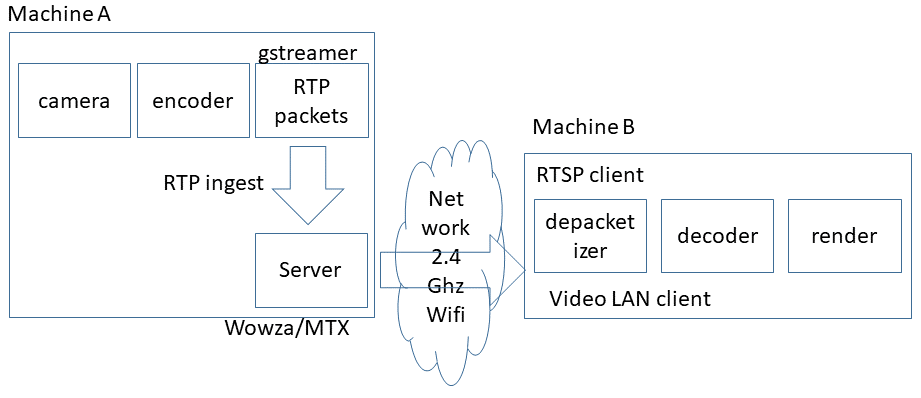


Figure 4x setup of the experiment using server

The results for the different configurations are plotted in Figures y5-y9 collecting traffic measurements. Only short intervals of time are measured, and the measurements from sender and receiver do not correspond timewise. The main point of investigation are the dynamic traffic characteristics relating to the burst size and periodicity, in each of the setups the burstiness corresponding to frame transmissions can be observed. It seems the MTX media the times between the bursts vary a bit more compared to the wowza setup.

From the experiments it seems the lower framerate mostly results from the camera capture, we have also ran tests with an artificial test source generating content at higher framerate, this results in the higher burst frequency.

In Figure y9 an articificial video source is used to ingest to MTX with a higher framerate and quality. The results show that in this case the burst frequency of traffic transmitted by MTX media is also increased.

In addition, in y10 also shows when both audio (PCM) and video (H.264) are transmitted from MTX media server, in this case we see the audio bursts are more frequent and smaller in size compared to the video burst.

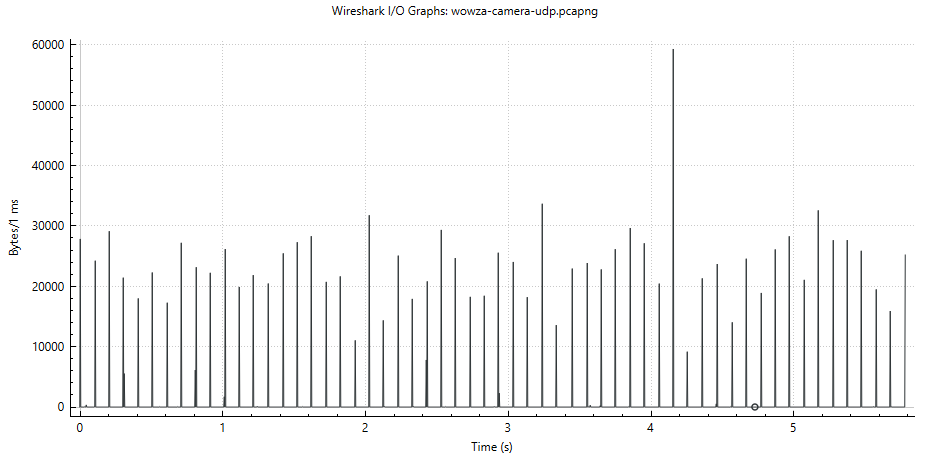


Figure y5 Wowza server side traffic sending to a VLC Receiver using RTSP/RTP

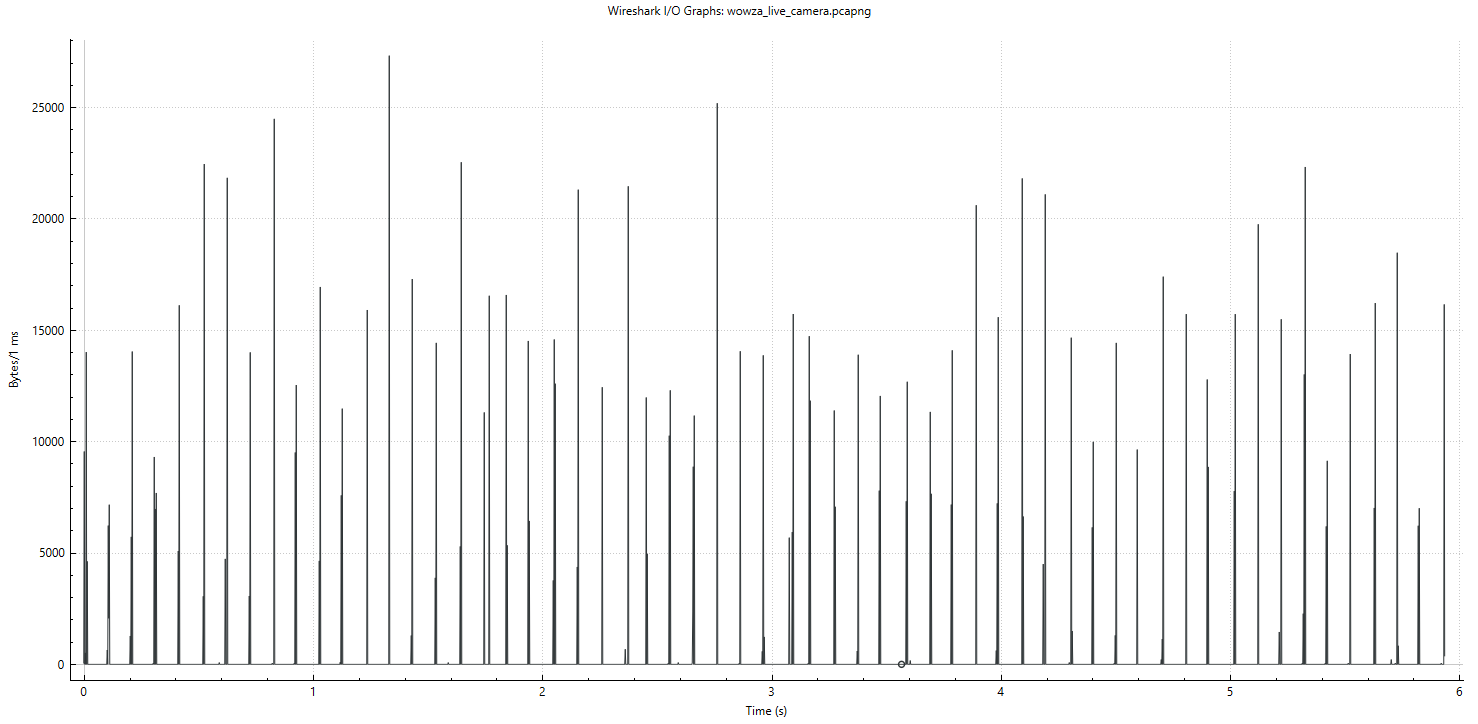


Figure y6 VLC receiver side traffic from received RTP Packets from Wowza Streaming Engine

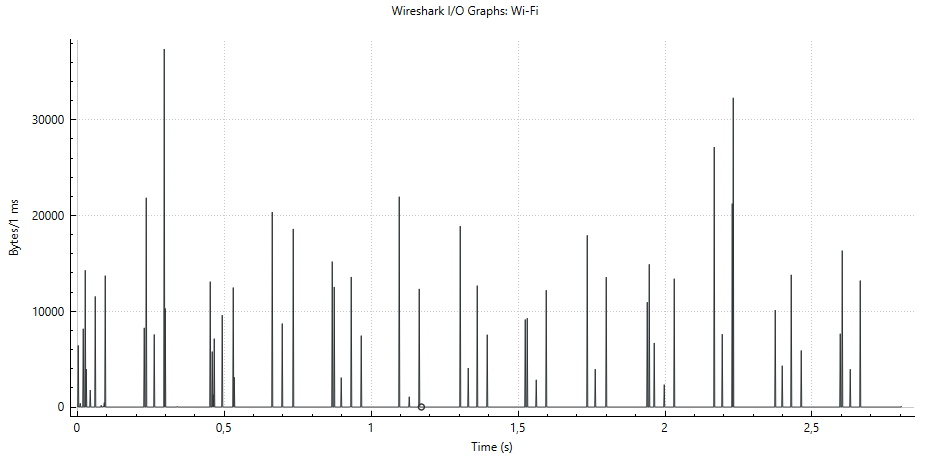


Figure y7 Camera RTP Traffic monitored from sender from MTX Media

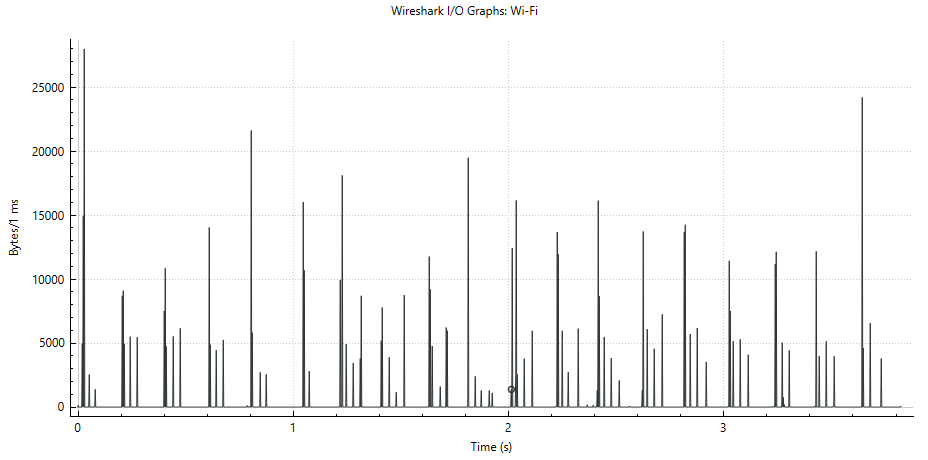


Figure y8 Camera RTP Traffic monitored at VLC receiver from MTX Media

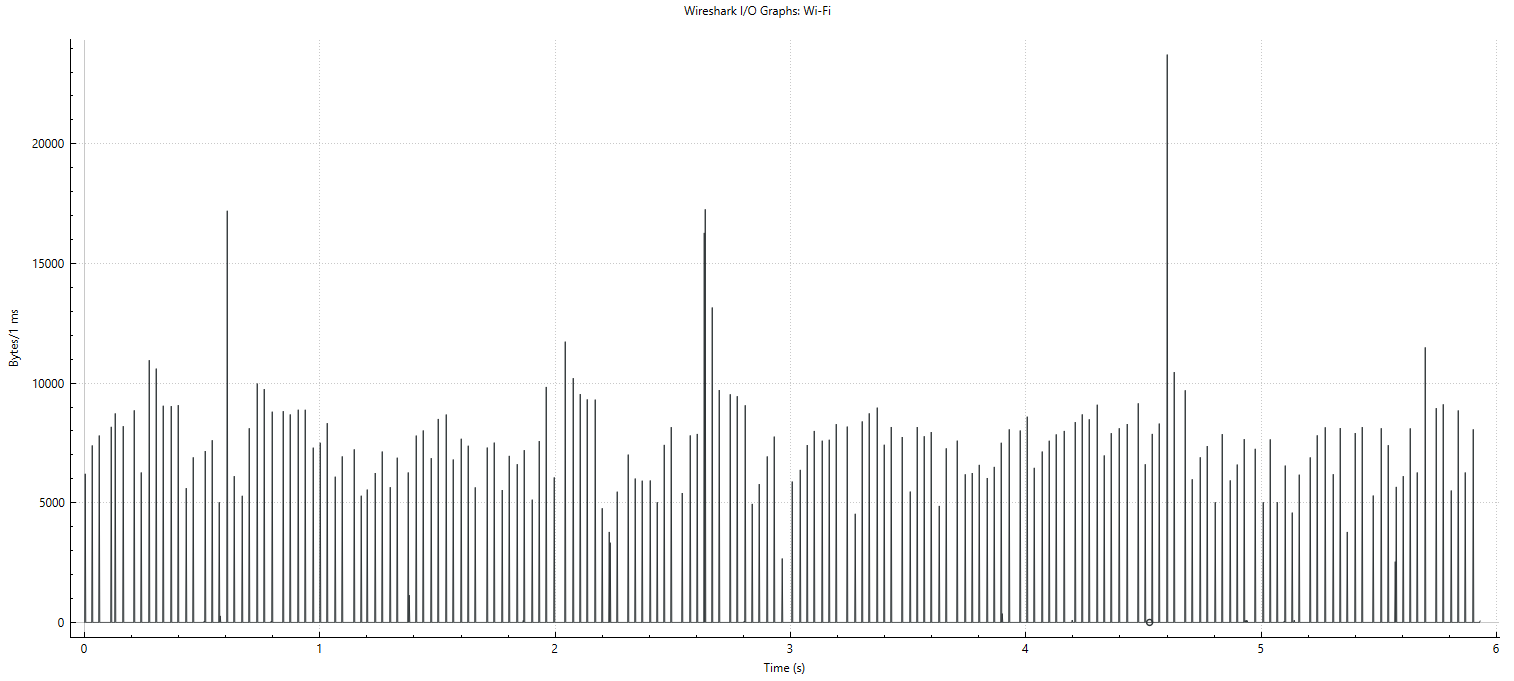


Figure y9 Syntetic test source RTP Traffic monitored from sender from MTX Media server

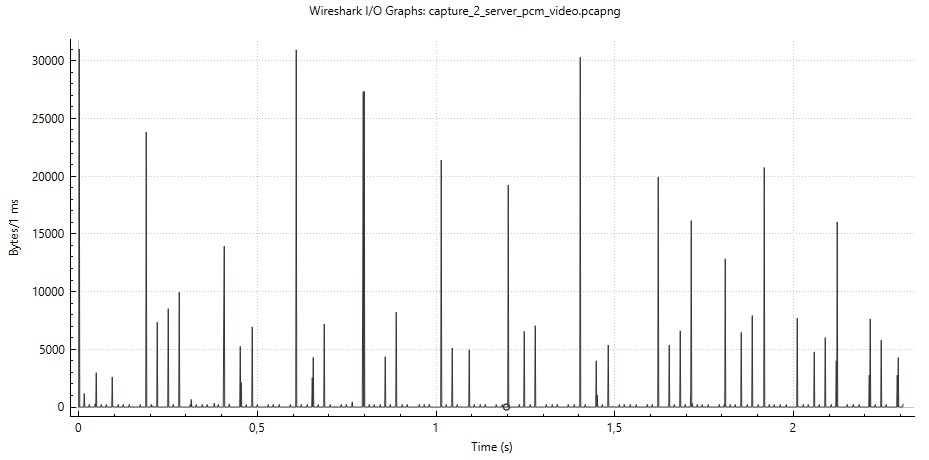


Figure y10 traffic pattern from MTX sender server audio + video

### 6.X.5 Discussion and Conclusion

**Observation 1:**

The data from the gstreamer RTP and from WebRTC browser implementation show bursts of data being transmitted in short time intervals. There is no evidence that the encoder/packager gradually produces and transmits packets, as in the experiments only short bursts were observed. It could happen on a per frame(s) basis in a short time interval. It may also be that some frames are combined in a single burst, but this cannot be observed from the current results.

Similar observations were made when an intermediate server was used for redistributing the streams.

It should therefore be possible for these application to calculate the burst size a priori before sending it out. The added latency should be limited to around a few milliseconds even in the worst case.

Bursts related to audio are more frequent than bursts related to video.

NOTE: The latency is related to sending the packets on the network, not to the encoder or RTP packager generating the packets. Detailed study of the cause of the delay is FFS. The preliminary conclusion is that a few milliseconds of delay may be introduced is only a worst case estimate.

**Observation 2:**

The inter burst time interval seems regular, but not constant. Therefore signalling of the time to next burst, if known by the application may be suitable for signalling as a dynamic traffic characteristic.

NOTE: this requires more study of different patterns and situations.

**Observation 3:**

It can be derived that when the network is not overloaded the WebRTC and Gstreamer implementations do not differ too much and the influence of the paced sender module is limited.

In addition, when the network bandwidth changes all of the sudden, the rate control may adapt so well that the effect of the paced sender is not significant.

**Observation 4 (extra on P2P versus Server based):**

When comparing the end-end latency of the peer to peer setups to the server based setups, for the peer to peer setups latencies < 1 seconds are achieved while in the server routing setup latencies of around 3 seconds are observed.

**Conclusion:**

Short periodic traffic bursts in short intervals occur in typical real time conversational applications using real-time video + audio.

Given the observed traffic behaviour and the observed application behaviour it seems achievable to include information about a burst size before it is being sent out as the durations of the burst are in the order of 1-2 milliseconds, but it may require some changes in the sender implementation to achieve this.

|  |
| --- |
| End of Changes |