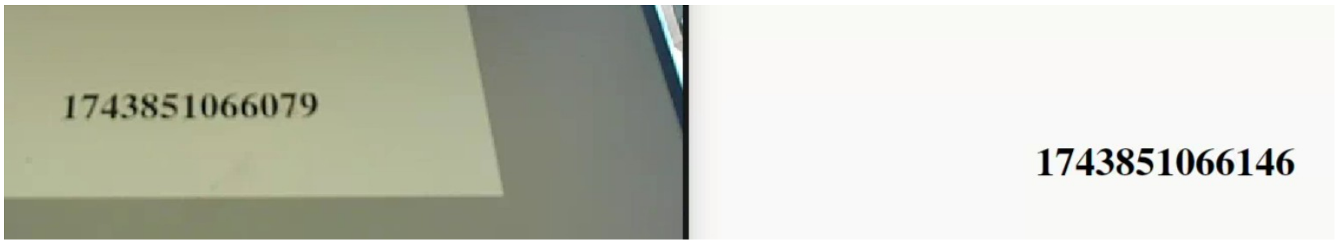
Fakidis George: Benchmarking gstreamer pipelines for 2D video end to end latency

Problem Statement

Measuring end to end latency for 2D video streaming usually involves specialized equipment and dedicated setups. We attempt to simplify the benchmarking process by creating two custom tools. In this analysis, the measurements are made using G-Streamer, an open-source multimedia-framework that provides building blocks for different scenarios of streaming multimedia.

Setup

The setup used here includes a desktop computer using Linux and a webcam(Creative Live! Cam Sync 1080p v2 Web Camera). Setting it up involves running a timer at a specified framerate and pointing the camera to the timer. That way we get 2 values, one from the actual timer and one after the whole video streaming operation completes. In practice, it looks like this:



Thus, taking their difference gives us the end to end latency for video streaming. For video streaming we used G-Streamer and experimented with different codecs and video formats. Here, we did not focus on comparing different renderers. Precompiled binaries of G-Streamer from package repositiories were used and not a custom compilation tinkering with compiler flags.

Introduction

In this effort, two tools were made in order to measure the end to end latency of a simple local video streaming system. The first one is a simple timer that can be configured based on the desired frame rate, useful to avoid tearing. The second one is a tool based on computer vision(CV), more specifically the **tesseract-ocr** model, that receives a video that contains two numbers as indicated above that are the milliseconds passed after pressing start and takes the difference between the two of them to compute the end to end latency of the video streaming.

With this setup it is considerably easier to benchmark different approaches. However the process of starting the timer, recording a video and then passing it to the CV tool could not be automated further but it already processes significantly more data than a manual method would.

The CV model had around 100ms of processing time to detect the text on a screen(given a jpeg of 1920x1080), real-time usage might be difficult.

G-Streamer Pipelines

G-Streamer pipelines are basically a way to easily connect different multimedia building blocks in order to achieve the desired result.. Using a command-line tool we can easily assemble a pipeline, which is basically a sequence of connected media elements. The general structure of every pipeline involves the sender taking the video feed from the camera (compressed or uncompressed), doing some kind of encoding, sent over the network. The receiver receives the network packets, decodes them and lastly renders them to the screen. To transfer video packets over the network, we use the Real-time Transfer Protocol(RTP), responsible for handling audio and video over IP networks. The "*videoconvert*" element in the pipelines below does not do any kind of expensive operations just converting between color spaces as rendering elements or codecs only support specific color spaces. The "*autovideosink*" element is basically choosing a renderer from the different ones available in the system. The "v4l2src" element receives video from the camera using the video for linux 2 driver(v4l2) which provides a compressed video feed and an uncompressed video feed. The "rtpXpay" and "rtpXdepay" elements are used to insert the video frame into the rtp payload and to extract the video frame from the rtp payload respectively.

Local-Compressed-JpegDec:
```
gst-launch-1.0 v4l2src ! jpegdec ! autovideosink
```

Local-Uncompressed:
```
gst-launch-1.0 v4l2src ! video/x-raw ! videoconvert ! autovideosink
```

Those two pipelines were used to establish a baseline before we attempt to send video over the network. Most webcams offer two kinds of video, compressed (MPEG) and uncompressed(YUYV color space buffer).

Rtp+jpeg-codec-4:

Sender:
```
gst-launch-1.0 v4l2src ! rtpjpegpay ! udpsink host=127.0.0.1 port=2001
```

Receiver:
```
gst-launch-1.0 udpsrc port=2001 ! application/x-rtp,media=video,encoding-name=JPEG,payload=26 ! rtpjpegdepay ! jpegdec !  videoconvert ! autovideosink
```

This pipeline was used to establish a baseline before using more sophisticated codecs. Sending uncompressed packages through the network was impossible due to the packet size limit of RTP.

Rtp+raw+vp8-codec-3:

Sender:
```
gst-launch-1.0 v4l2src ! video/x-raw ! videoconvert ! vp8enc ! rtpvp8pay ! udpsink host=127.0.0.1 port=2001
```

Receiver:
```
gst-launch-1.0 udpsrc port=2001 ! application/x-rtp,media=video,encoding-name=VP8,payload=96 ! rtpvp8depay ! vp8dec ! videoconvert ! autovideosink
```

Rtp+raw+multithreaded+vp8-codec-6:

Sender: gst-launch-1.0 v4l2src ! video/x-raw ! videoconvert ! vp8enc threads=4 ! rtpvp8pay ! udpsink host=127.0.0.1 port=2001

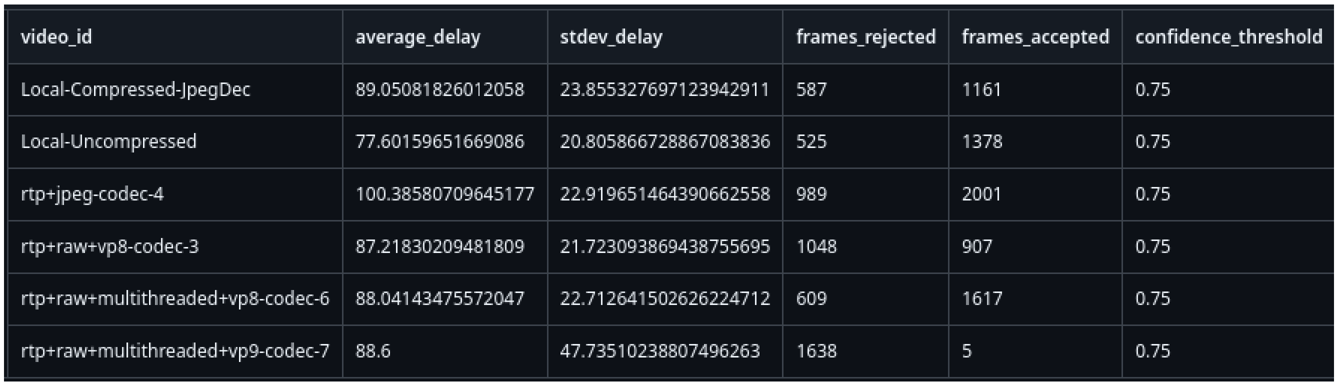
Receiver: gst-launch-1.0 udpsrc port=2001 ! application/x-rtp,media=video,encoding-name=VP8,payload=96 ! rtpvp8depay ! vp8dec threads=4 ! videoconvert ! autovideosink

Rtp+raw+multithreaded+vp9-codec-7:

Sender: gst-launch-1.0 v4l2src ! video/x-raw ! videoconvert ! vp9enc threads=4 ! rtpvp9pay ! udpsink host=127.0.0.1 port=2001

Receiver: gst-launch-1.0 udpsrc port=2001 ! application/x-rtp,media=video,encoding-name=VP9,payload=96 ! rtpvp9depay ! vp9dec threads=4 ! videoconvert ! autovideosink

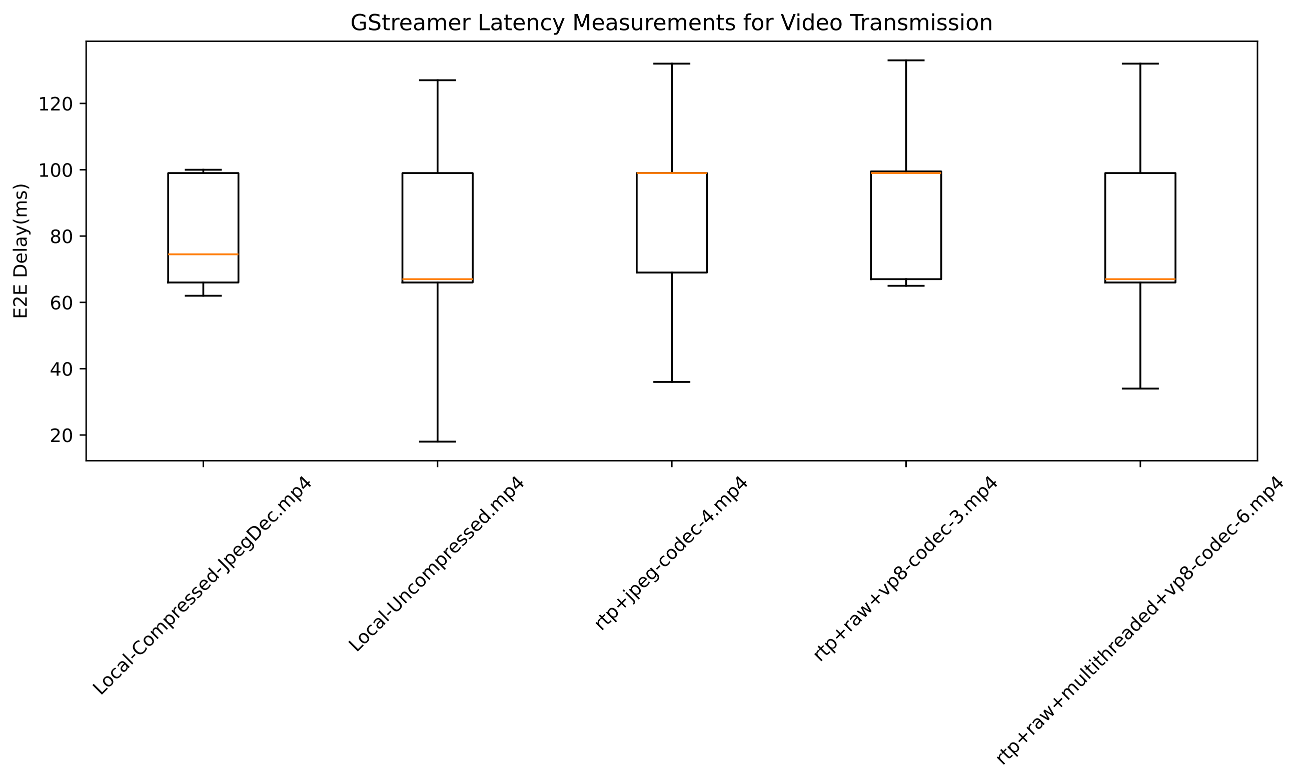
Results



video_id	average_delay	stdev_delay	frames_rejected	frames_accepted	confidence_threshold
Local-Compressed-JpegDec	89.05081826012058	23.855327697123942911	587	1161	0.75
Local-Uncompressed	77.60159651669086	20.805866728867083836	525	1378	0.75
rtp+jpeg-codec-4	100.38580709645177	22.919651464390662558	989	2001	0.75
rtp+raw+vp8-codec-3	87.21830209481809	21.723093869438755695	1048	907	0.75
rtp+raw+multithreaded+vp8-codec-6	88.04143475572047	22.712641502626224712	609	1617	0.75
rtp+raw+multithreaded+vp9-codec-7	88.6	47.73510238807496263	1638	5	0.75

Discussion

Confidence threshold 0.75 means that if the CV model was less than 75% sure about the result it gave, then the frame would be rejected. This number was selected after manual inspection of frames and seeing the confidence of the model. Whenever the model was less than around 75-80% confident the answer was mostly wrong as the numbers were not clear enough. The results are similar to a colleague’s approach to measurements with dedicated equipment and setup so the tooling can be used for further experiments. VP9 codec could not function properly hence the overwhelming number of rejected frames. Multithreading does not seem to offer much of a reduction in latency as expected, probably needed for bigger resolutions where the payload is larger. The VP9 measurements are not included in the plot as they were invalid as the codec seemed to not function properly.



Delays are practically how many frames behind is the stream from the actual thing being showed(the timer in this case), hence the standard deviation of delay was around 20ms, which means mostly a frame behind and sometimes exactly at the same frame.

An observation of mine is that the local uncompressed video should have lower latency as 30 fps is 33ms per frame drawn, and instead the latency is twice that. Ideally it would be 33ms + the time needed to draw a buffer on the screen, which if GPU accelerated, should be much lower than 33ms if my intuition is correct. Using a gpu pipeline it could be properly diagnosed it it is a problem of G-Streamer's timing mechanisms. The 30fps is enforced by the camera's latency anyway, so trying to rate-limit the drawing causes the doubling of the latency, as soon as a frame is received it should be drawn.

Trying to automate the whole recording video process and then CV the video buffer to have the result in real-time was disproportionately more time-consuming, but might be implemented in the future.