Development of Augmented-Reality (AR) Video Streaming Over WebRTC and Analysis of Processing Overheads

Submitted in partial fulfillment of the requirements of
CS 8903

By
Kalit Inani
ID No. 904026355

Advised By:
Dr. Umakishore Ramachandran

Under the supervision of:
Anirudh Sarma

GEORGIA INSTITUTE OF TECHNOLOGY, ATLANTA
December 2024

Abstract

The next generation of real-time streaming applications like video-conferencing apps will augment video streams that are generated and computed from the feed. Understanding computation overheads present in enabling such advanced features can inform scheduling strategies within the edge computing environments. This research project aims to explore the possibilities of distributing the components of WebRTC, the backbone technology for real-time video conferencing, across edge devices. We specifically focus on the analysis of the computational overheads associated with deploying augmented reality and FFmpeg filters in video streams. All contributions done as part of the project can be found here: https://github.com/Kalit31/WebRTC_research.

1. Introduction

A. WebRTC

WebRTC, which stands for Web Real-Time Communication, provides an API as well as a protocol for peer-to-peer communication and data transmission. It finds a peer-to-peer path to exchange audio/video in an efficient and low-latency manner. The protocol is composed of 4 major sequential steps. I will explain each of them briefly below:

1. Signalling -

   To initiate a WebRTC communication, the users must know whom they are going to communicate with and what data they will be sharing. This step helps to initiate the communication between two users. During the setup phase, an entity known as a signalling server acts as a proxy to route messages from one user to another. All messages use Session Description Protocol (SDP). Each user encodes their addresses, ICE candidates, security options and metadata about the audio/video tracks they wish to exchange. An important point to consider is that the connection between the user and the signalling server is not a WebRTC connection yet. Rather, the messages are sent using usual REST endpoints or websockets (we will use the latter in our setup).

2. Connecting and NAT Traversal -

   Once the SDP messages are exchanged between the users, they can attempt a direct connection using a tool known as Interactive Connectivity Establishment (ICE). It is a protocol which assists in the establishing a direct communication between two machines without the need of a central server. Internally, they use a concept called ‘NAT Traversal’ and STUN/TURN servers.

   - NAT, which stands for Network Address Translation, is a technique used to map private IP addresses to public IP addresses and vice versa. NAT allows multiple devices on a local network to share a single public IP address when accessing the internet. It acts as an intermediary between the internal network and the outside world.

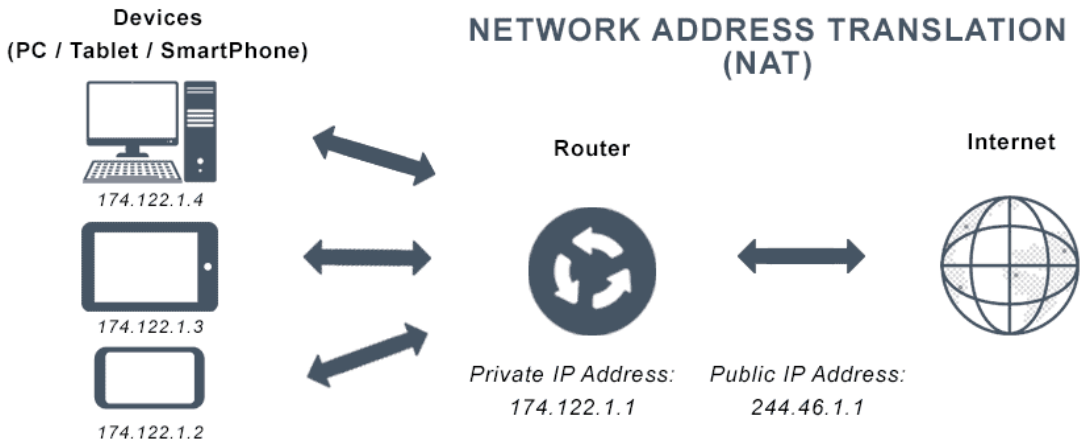


Fig: An illustration of NAT

- STUN, which stands for Session Traversal Utilities for NAT, is a protocol used to enable devices behind a NAT firewall or router to discover their public IP address and port mapping, and to facilitate the traversal of NATs in real-time communication applications like video conferencing. In simple terms, the user agent requests a STUN server to inform about its public IP address and port through NAT.

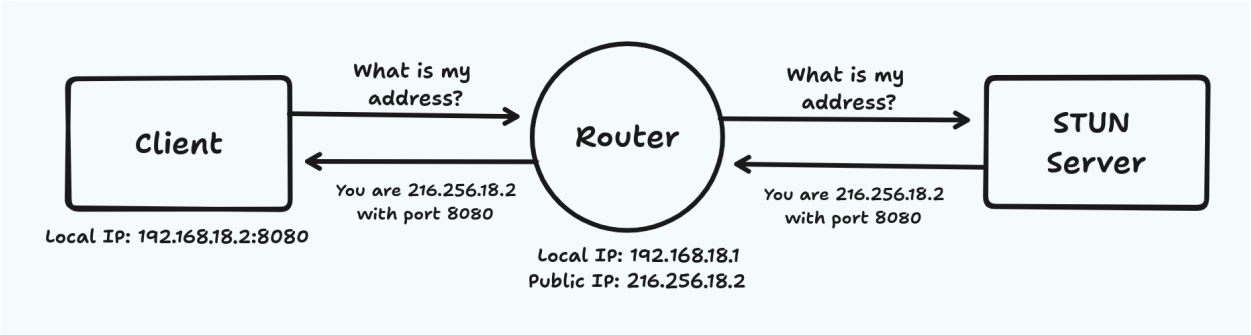


Fig: An illustration of STUN server

- TURN, which stands for Traversal using Relays around NAT, is a protocol used to facilitate communication between devices that are behind strict NATs, like Symmetric NAT, where STUN-based methods are not relevant. TURN provides a solution by relaying the traffic through a TURN server, which acts as an intermediary between the two clients.

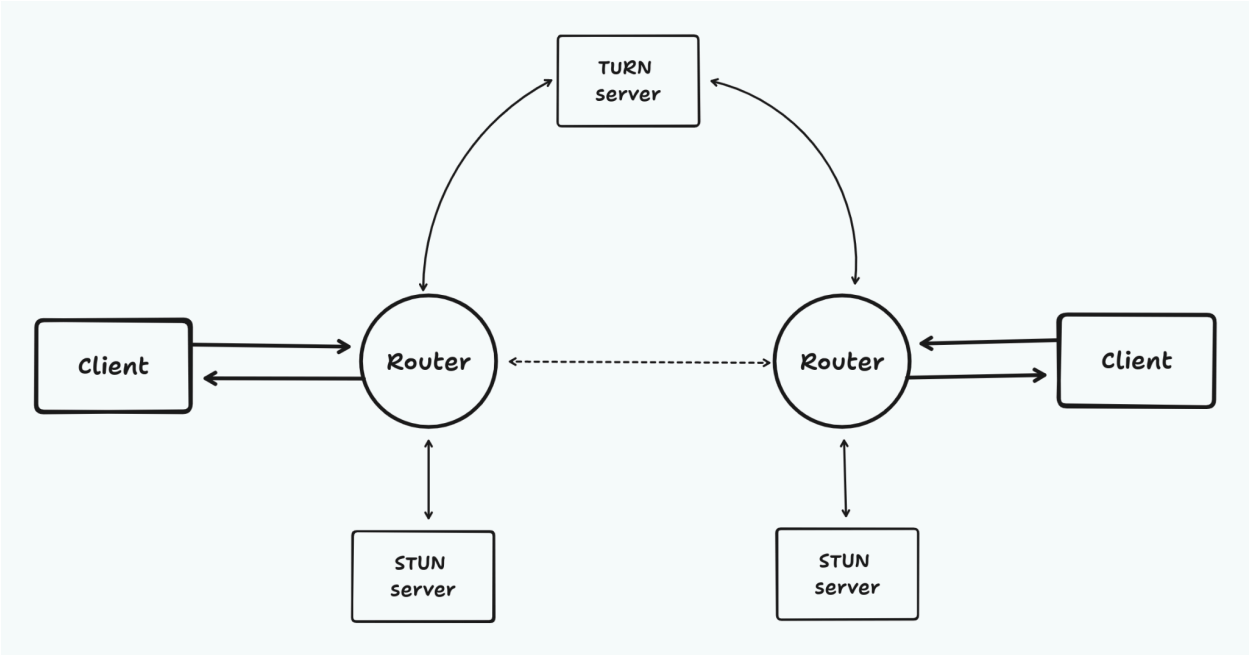


Fig: Comparison between TURN and STUN servers

- In more detail, the ICE protocol enables devices to find the best possible network path to establish a direct communication channel. It collects all available candidates for how the devices can potentially connect. Among these candidates, it selects the candidates which form the most efficient path. Next, these candidates are relayed over to the remote peers using SDP through the signalling server.

3. Securing the transport layer -

   The next step is to establish an encrypted transport channel for sharing data. WebRTC uses two protocols called Datagram Transport Layer Security (DTLS) - TLS over UDP, and Secure Real-Time Transport Protocol (SRTP) - encrypts RTP packets. The DTLS encrypted connection is used for DataChannel messages whereas audio/video transmission is secured using SRTP.

4. Communicating with peers -

   WebRTC uses two protocols for communication namely Real-Time Transport Protocol (RTP) and Stream Control Transmission Protocol

(SCTP). RTP is used to exchange media encrypted with SRTP whereas SCTP is used for exchanging messages with DTLS.

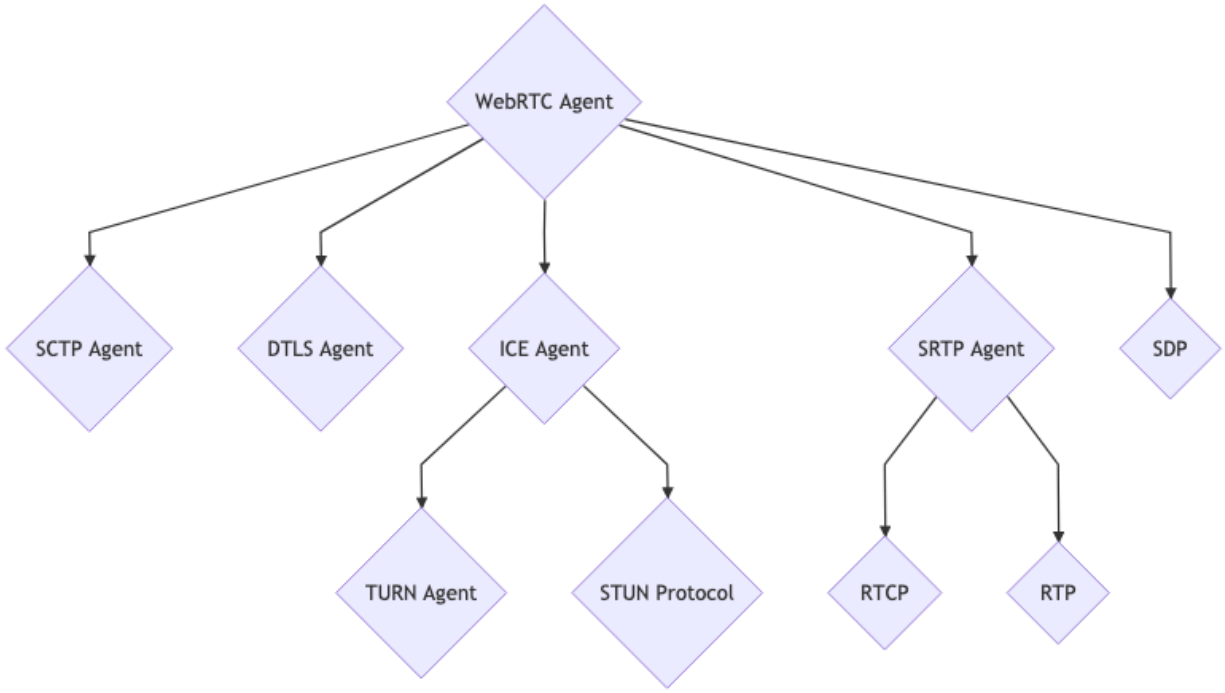


Fig: A set of protocol stack involved in WebRTC

As part of the current project, we use the WebRTC protocol to develop a video-conference application.

B. FFmpeg

FFmpeg is an open-source and popular command line tool for processing video, audio and other multimedia files. It provides a wide range of tools for decoding, encoding, transcoding, multiplexing, demultiplexing, streaming, filtering and manipulating any type of multimedia files. It supports 100s of codecs including popular ones like H.264, VP9, AAC, and MP3.

A typical usage of ffmpeg looks like this: a multimedia file is given as an input to ffmpeg. It then demultiplexes the audio and video tracks into separate data packets. These packets are then decoded into uncompressed frames. Here,

additional processing and filtering can be performed on the frames like adjusting the brightness, scale, bitrate, etc. Next, the frames are encoded again and multiplexed into an output file.

FFmpeg provides several low-level libraries, including *libavcodec, libav, libswscale*, for developing multimedia processing software.

As part of the current project, we employ FFmpeg to stream source videos to users. This stream is later shared by each user over the WebRTC connection. Moreover, we compute the processing time for adding FFmpeg-based filters on top of each frame in the video stream.

C. Augmented Reality Filters

With the rise in technological advancement in the realm of Augmented Reality (AR), AR-based filters are expected to become a commonplace. Today, there exist several social media applications like Instagram, Facebook, TikTok, Snapchat, and Whatsapp which employ these types of AR-filters. Augmented-Reality filters are digital overlays on top of video streams that add interactive elements to real-world environments. Some examples include, adding a hat on a person's head, displaying crackers when a person is clapping, animating a person's face, etc.

The next-generation of real-time streaming applications will augment video streams with augmented reality assets which are computed from the feed. Thus, understanding the overheads they incur will serve to be crucial to develop scheduling strategies in edge computing environments. The overheads are not only limited to the latency, but also the energy and battery consumption.

As part of the current project, we employ Google's Mediapipe library to overlay AR filters (more details can be found later). The library worked well on CPU, however it posed several challenges in compatibility with GPU and Nvidia Jetson TX1 (discussed later).

2. Methodology and Discussions

Let us start by designing the WebRTC video-conference application. We already discussed the basic overview of the protocols and APIs it offers. Now, let us see how they work in practice.

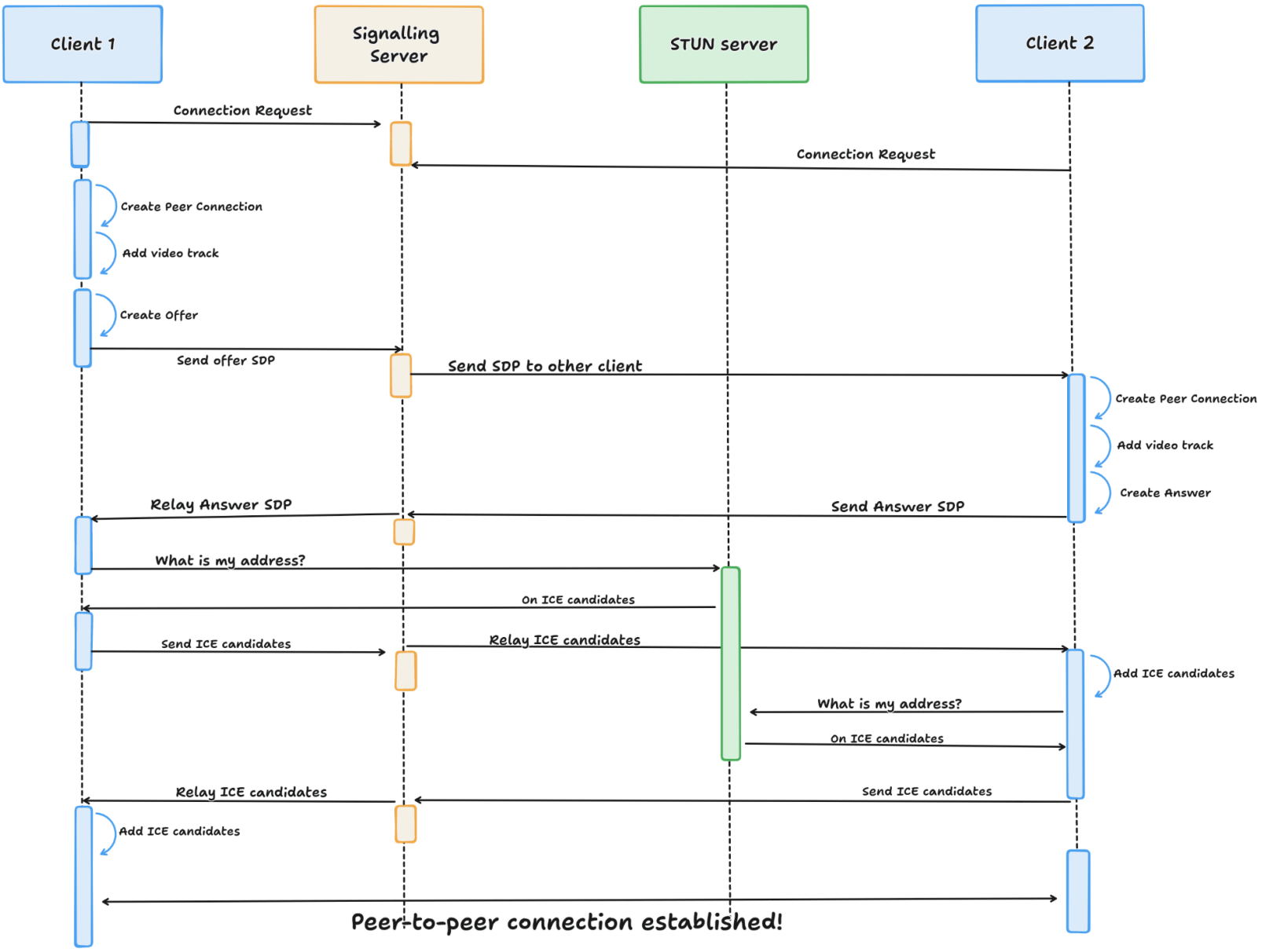


Fig: WebRTC peer-to-peer connection setup

The above figure provides a comprehensive illustration of the internal workings in a WebRTC peer-to-peer connection. Here is the brief description of each of the component:

- Signalling server: It acts as the proxy server between the two clients before the WebRTC p2p connection is established. It relays SDP messages to-and-fro clients.
- Client: It initiates a peer-connection, adds a H.264 video RTP track, generates an offer SDP with the above added track information and sends it to the signalling server. It also pings the STUN server asking for ICE

candidates (entities to help locate a client on the Internet) and shares this as well with the signalling server. The signalling server relays these messages with the other client.

- STUN server: As discussed in the above section, its task is to fetch ICE candidates for the requesting client.
- The offer which the client generates locally is set as the local description, and the answer which it receives for its offer is set as the remote description. The case is opposite for the other client. The second client sets the offer which was sent by the first one as the remote description. Also, it sets the answer which is to be sent to the other client as its local description.

We implement the above setup using Pion's WebRTC API implementation in Go.

As our real-time video sharing setup is ready, let us shift our gears to understand the processing and filtering in video streams using FFmpeg and Google's mediapipe.

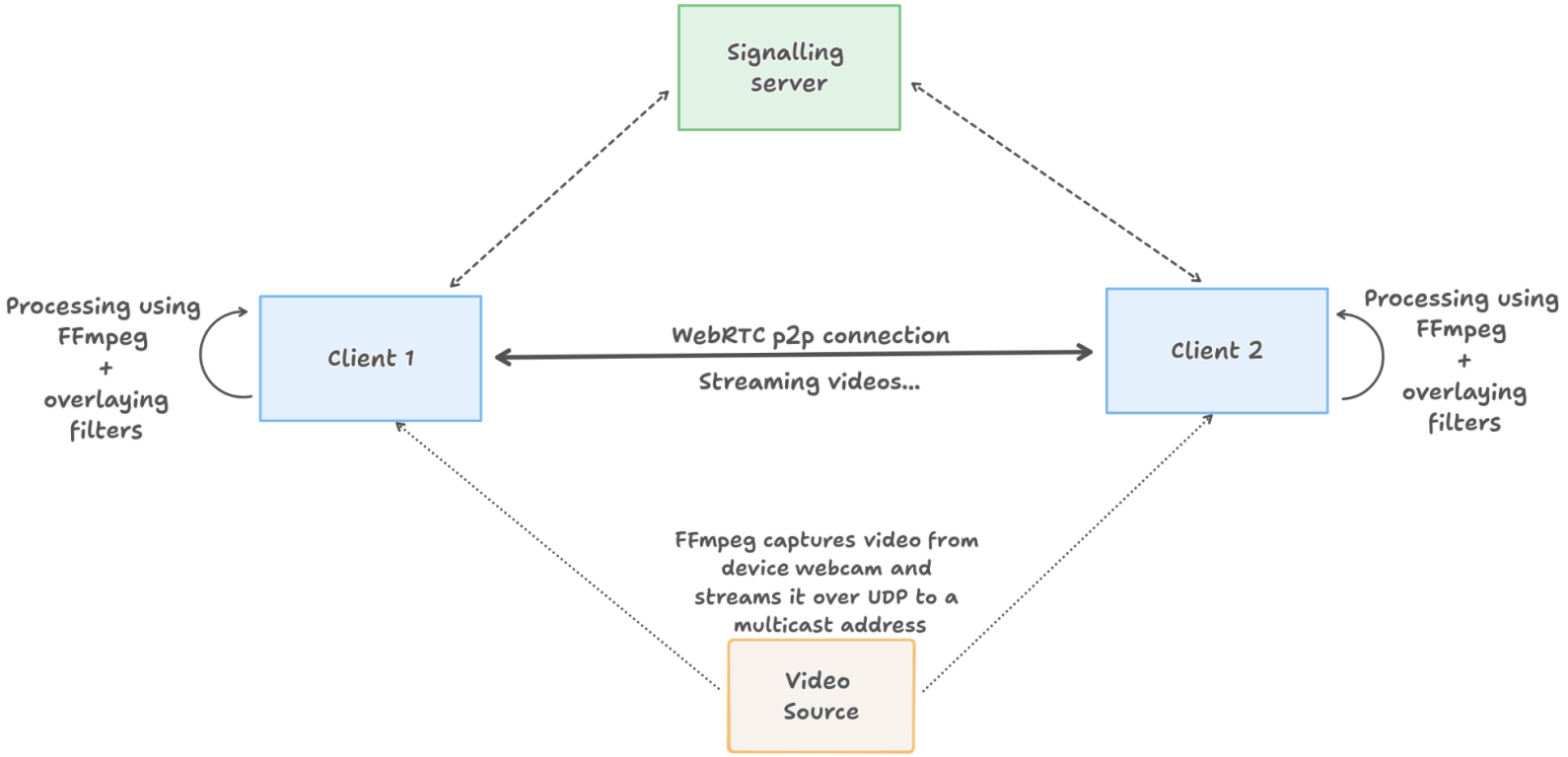


Fig: Initial setup of our system

The initial experiments are run on my local machine with the following stats:

- Intel i7 processor with 12 cores

- 16 GB RAM
- NVIDIA GeForce GTX 1050 Ti/PCIe/SSE2 Graphics Card

Since both the clients are running on the same machine, they cannot use the webcam video streams simultaneously. To mitigate this, we use FFmpeg to capture the video stream from the device camera, encode the stream as a MPEG transport stream and finally stream it over UDP to a multicast address. This way, both the clients can fetch the video stream from the multicast address, perform processing on it and exchange videos with their peers. Below is the FFmpeg command for the same:

```
ffmpeg -f v4l2 -i /dev/video0 -f mpegts udp://224.0.0.251:5353
```

- *-f v4l2*: It tells ffmpeg to use video4linux2 format for capturing video from the device
- *-i /dev/video0*: It tells ffmpeg to capture video from the first video device (webcam)
- *-f mpegts*: It tells ffmpeg to encode the video stream in MPEG-TS format.
- *udp://224.0.0.251:5353*: This specifies the multicast address which serves as the destination of the output stream.

We explore processing and evaluate computation time for two types of filters. Below are the flowchart of the processing workflow for each of them. The video processing code is written by making use of the Golang implementation of FFmpeg provided here: https://github.com/asticode/go-astiav.

A. FFmpeg-based filters

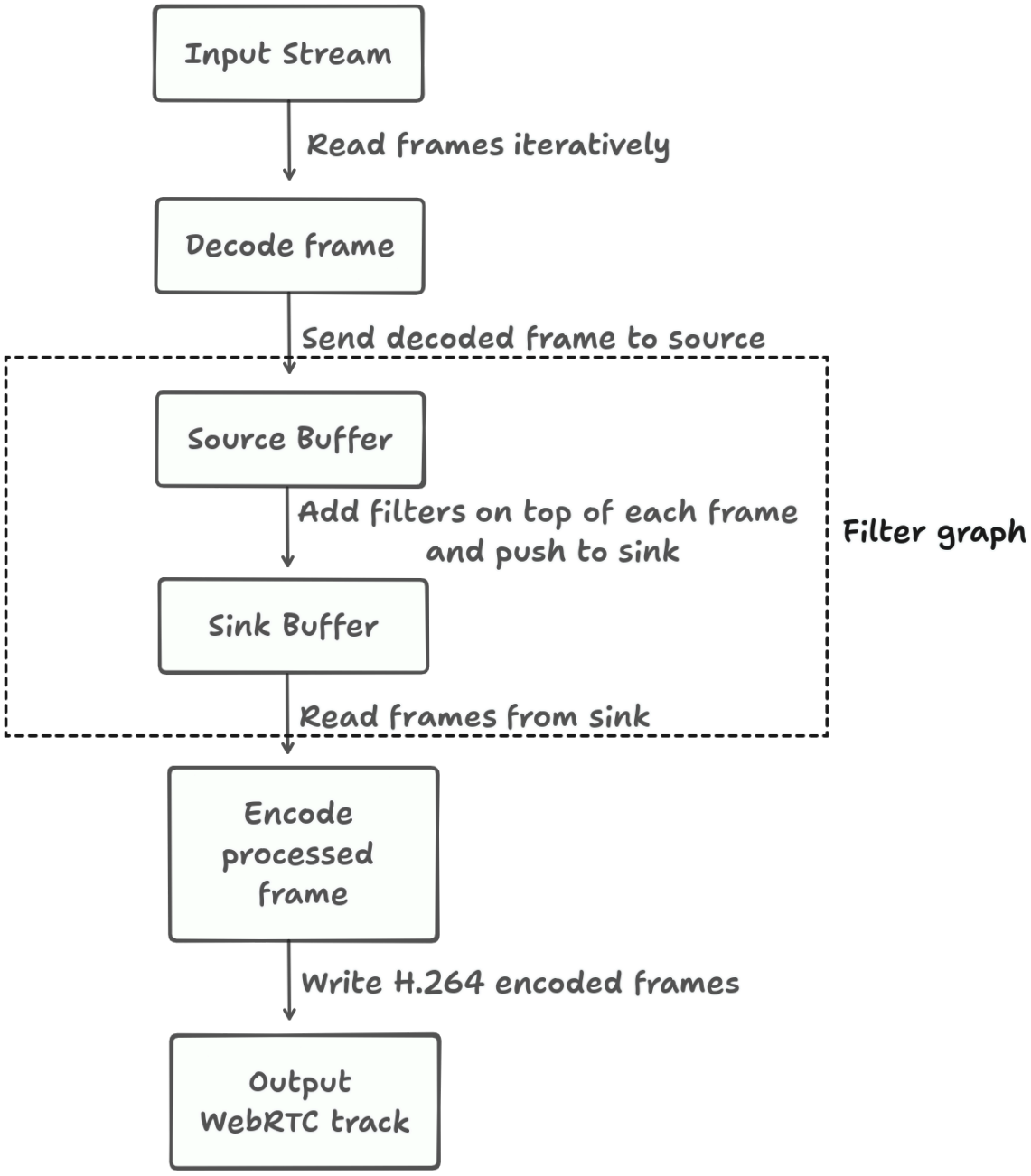


Fig: Flowchart of our workflow for adding FFmpeg-based filters

B. Augmented Reality (AR) filters

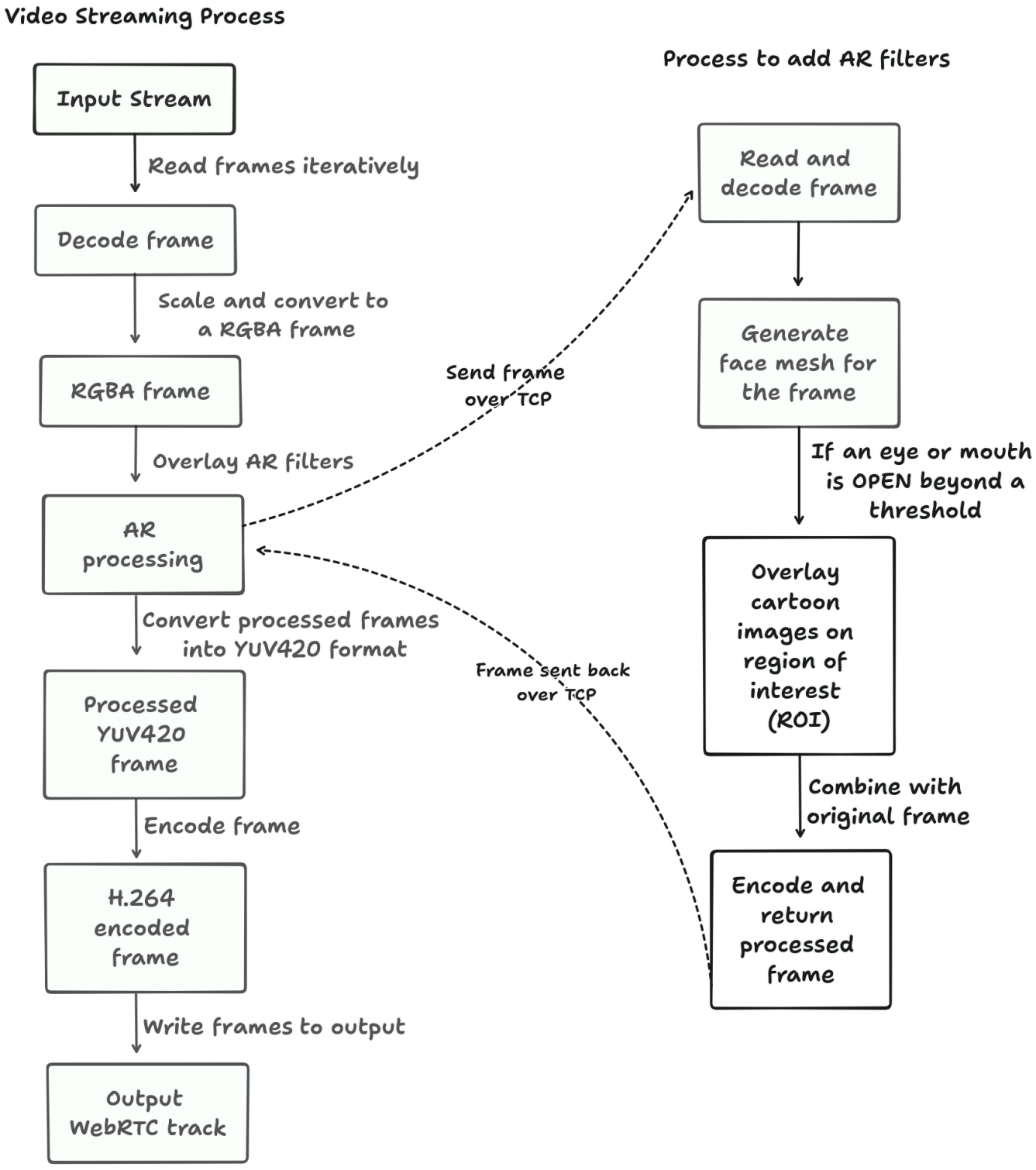


Fig: Flowchart of our workflow for adding Augmented Reality (AR) filters

In both the above approaches, we perform a real-time computation of the processing time for each frame (more details in the experiments section).

We also explored volumetric video datasets. As part of the exploration, we stumbled across a comprehensive and recent dataset for full scene volumetric videos - https://cuhksz-inml.github.io/full_scene_volumetric_video_dataset/factsfigures.html. The dataset was massive, containing 500 GBs of volumetric videos, and captured using Azure Kinect RGBD cameras. However, due to limited documentation, huge dataset size as well as limited working knowledge in concepts like body poses, point clouds, etc. we decided to avoid putting in further efforts in it.

We also tried exploring Jetson benchmark suites which work on a video dataset for a variety of tasks like object detection, segmentation, etc.. However, all popular benchmark suites (https://github.com/dusty-nv/jetson-inference) required a more recent Jetpack version, and the machine present in the lab was running out-dated software. We explored ways to upgrade the software stack, but it posed some challenges (discussed later).

3. Experiments

Let us first see how the WebRTC connection is established between two clients with the help of a signalling server.

A. Signalling Server:

```
Starting server on :8080
Received message from client 0: join
Received message from client 1: join
Writing to client 0
Received message from client 0: offer
Writing to client 1
Received message from client 0: iceCandidate
Writing to client 1
Received message from client 0: iceCandidate
Writing to client 1
Received message from client 0: iceCandidate
Writing to client 1
Received message from client 1: answer
Writing to client 0
Received message from client 1: iceCandidate
```

```
Writing to client 0
```

Fig: Logs on the signalling server

B. Client 1:

```
Connected to the server
Message from server: join
New ICE candidate: {candidate:2035536029 1 udp 2130706431 10.0.0.180
47020 typ host 0xc0002b8090 0xc00029c12a <nil>}
New ICE candidate: {candidate:346383794 1 udp 2130706431
2601:c2:b81:8d0::6568 34551 typ host 0xc0002b8130 0xc00029c17a <nil>}
Message from server: answer
Setting remote description with answer.
Message from server: iceCandidate
Received ICE Candidate: candidate:2035536029 1 udp 2130706431
10.0.0.180 48191 typ host
Peer connection not created yet. Returning...
Message from server: iceCandidate
Message from server: iceCandidate
Received ICE Candidate: candidate:1557957486 1 udp 2130706431
2601:c2:b81:8d0:db96:7395:1aac:3c80 44235 typ host

Successfully established a WebRTC connection between clients
Writing to tracks
Connection State has changed: checking
New ICE candidate: {candidate:1046940177 1 udp 1694498815
73.237.244.77 36566 typ srflx raddr 0.0.0.0 rport 36566 0xc0005b6fa0
0xc00058aeaa <nil>}
Message from server: iceCandidate
Received ICE Candidate: candidate:1046940177 1 udp 1694498815
73.237.244.77 40679 typ srflx raddr 0.0.0.0 rport 40679
ICE Candidate added successfully.

Connection State has changed: connected
Successfully connected!
```

C. Client 2

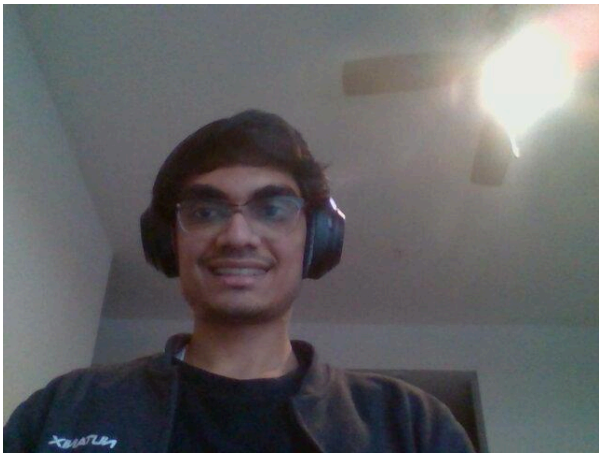
```
Connected to the server
Message from server: offer
Message from server: iceCandidate
Received ICE Candidate: candidate:2035536029 1 udp 2130706431
10.0.0.180 47020 typ host
Peer connection not created yet. Returning...
Received offer
Connection State has changed: checking
Successfully established a WebRTC connection between clients
Writing to tracks
New ICE candidate: {candidate:2035536029 1 udp 2130706431 10.0.0.180
48191 typ host 0xc000038110 0xc00001419a <nil>}
ICE Candidate added successfully.
Connection State has changed: connected
Successfully connected!
```

Once a successful WebRTC connection is established, the clients can begin to stream videos over the added tracks.

Next, let us analyze how well FFmpeg and AR-based filters perform.

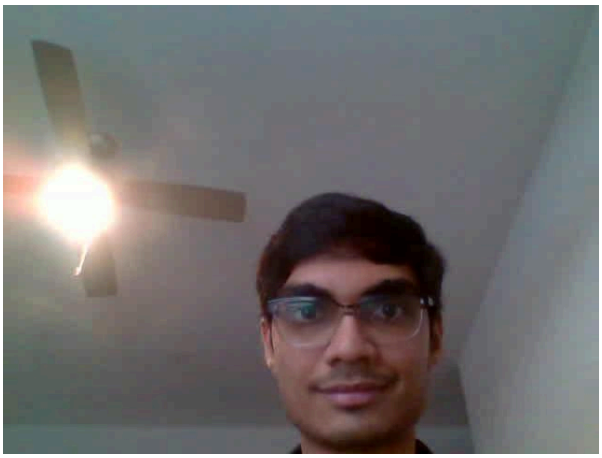
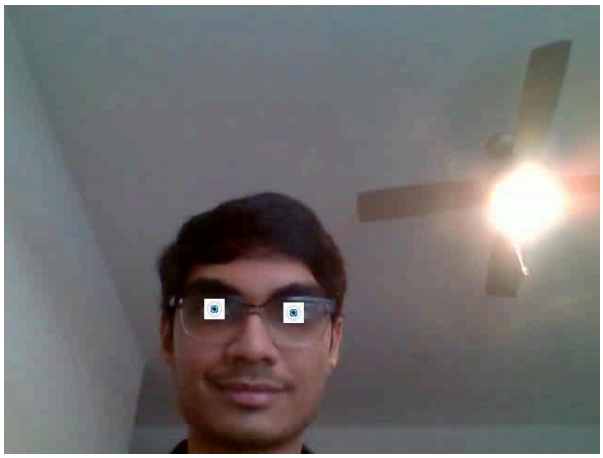
A. FFmpeg-based filters:

The FFmpeg based filters were given the task to increase the brightness by 50% and vertically flip the image frame.

Original (Before)	Processed (After)
	

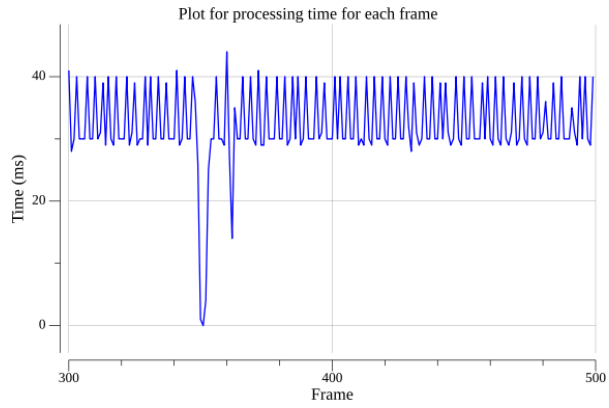
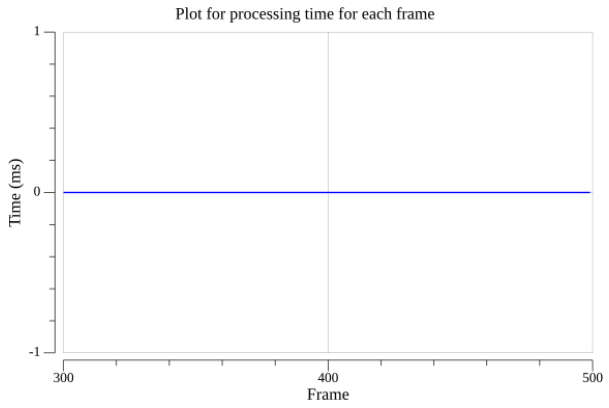
B. Augmented Reality (AR) filters:

We experimented with some basic AR filters: if the size of the eye (eye was open) and mouth increased by a threshold, a cartoon image was overlaid on top of each part. The overlaid cartoon image size would be determined by the extent to which each part is open. For eg: a larger filter image would be added on an enlarged eye compared to a normally open eye.

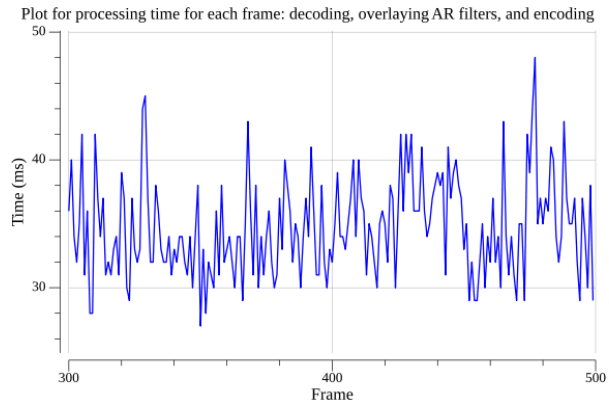
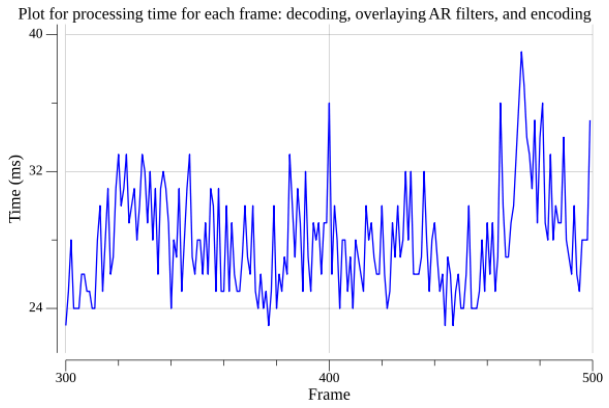
Original (Before)	Processed (After)
	
	

Next, let us look at the computation complexity graphs for each of them.

A. FFmpeg-based filters:

Complexity (decoding, adding filters, encoding) per frame	Complexity of just adding filters per frame
	

B. AR-based filters:

Complexity (decoding, adding filters, encoding) per frame	Complexity of just adding filters per frame
	

The above plots show that the processing time for AR-based filters is higher than the FFmpeg-based filters.

Next, to analyse the task from a different perspective - if hardware acceleration helps in any way? We, thus, decided to run the experiments on a machine with hardware acceleration support. We tried to run the experiments on a Nvidia Jetson TX1 available in the lab. However, after putting in a great amount of effort, we were unsuccessful in getting results. The issues we faced here are discussed in the challenges section.

Jetson device statistics:

```
Jetson: 4 CPU cores ARMv8 Processor rev 1
Device 0: "NVIDIA Tegra X1"
  CUDA Driver Version / Runtime Version          10.0 / 10.0
  CUDA Capability Major/Minor version number:    5.3
  Total amount of global memory:                 3962 MBytes
(4154626048 bytes)
  ( 2) Multiprocessors, (128) CUDA Cores/MP:     256 CUDA Cores
  GPU Max Clock rate:                            998 MHz (1.00 GHz)
  Memory Clock rate:                             1600 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 262144 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536),
2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
```

```
  Concurrent copy and kernel execution:          Yes with 1 copy
engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            Yes
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with
device simultaneously) >
```

4. Challenges

- The initial few challenges were in setting up the stream processing pipeline using ffmpeg and ensuring proper format of the frames for processing
    - https://github.com/asticode/go-astiav/issues/77
    - https://github.com/asticode/go-astiav/issues/90
- One of the major challenges in this project was to work with Google's Mediapipe library. We were able to successfully run experiments on my local machine. But, Mediapipe's limited (or close to no) support for Jetson posed a major hurdle.
    - https://github.com/google-ai-edge/mediapipe/issues/5736
        - https://github.com/google-ai-edge/mediapipe/issues/4017
        - https://github.com/google-ai-edge/mediapipe/issues/5344
        - https://github.com/google-ai-edge/mediapipe/issues/3353
        - https://github.com/google-ai-edge/mediapipe/issues/1651
        - https://github.com/google-ai-edge/mediapipe/issues/1344
        - https://github.com/google-ai-edge/mediapipe/issues/5736
    - Apparently, the legacy solution for Mediapipe which is compatible with Jetson TX1 requires Jetpack 4.6 and higher. However, upgrading the jetpack version also posed hiccups.
- Another critical challenge was to upgrade the outdated Jetpack version on the Jetson machine. However, today, there is very little support and documentation of upgrading Jetpack on Jetson TX1 machines.

5. Future Work and Conclusion

The contributions made as part of this project include:

- Developing an end-to-end video conferencing platform with WebRTC
- Designing and building the processing pipeline for video frames.
- Estimating the computation complexity for processing different filters
- Exploration of volumetric video datasets
- Exploring if hardware acceleration show some advantage (though, unsuccessful due to resource limitations)

In the future, this work can be extended by making use of the video platform and performing a fine-grained analysis of AR processing on edge devices. The experiments could be run on more sophisticated hardware stack in the future. The problem could be explored from other perspectives like energy consumption and battery drain analysis.

6. References

a. Cloud Rendering-based Volumetric Video Streaming System for Mixed Reality Services
b. Edge AR X5: An Edge-Assisted Multi-User Collaborative Framework for Mobile Web Augmented Reality in 5G and Beyond
c. Video Calling with Augmented Reality Using WebRTC API
d. FSVVD: A Dataset of Full Scene Volumetric Video
e. https://cuhksz-inml.github.io/full_scene_volumetric_video_dataset/factsfigures.html
f. WebRTC For The Curious: https://webrtcforthecurious.com/
g. Pion WebRTC: A Go implementation of the WebRTC API https://github.com/pion/webrtc
h. Github repository containing the source code for FFmpeg in C: https://github.com/FFmpeg/FFmpeg
i. A Golang-based implementation of ffmpeg and libav C bindings: https://github.com/asticode/go-astiav
j. Examples of WebRTC applications: https://github.com/pion/example-webrtc-applications
k. Google Mediapipe: Cross-platform, customizable ML solutions for live and streaming media - https://github.com/google-ai-edge/mediapipe
l. WebRTC Illustrations: https://github.com/qmcloud/WebRTC_IM
m. Guide to deploying deep-learning inference networks and deep vision primitives with TensorRT and NVIDIA Jetson: https://github.com/dusty-nv/jetson-inference