logic fruit Technologies

Whitepaper

Webcam based LIVE STREAMING

A Software Perspective

By:
Amritpreet Singh
Project Manager

Contents

Abstract

Applications that directly or indirectly **stream webcam video** are commonly available today. This paper intends to give a detailed review of an application built using Live555 and x264 library set, supported by V4L2 driver APIs, that could stream live webcam video to any number of requesting clients as per request. The different modules necessary for live webcam streaming will be sequentially uncovered in this presentation.

Introduction

It has been a few decades now since video technology has come into action. In the early days video was captured and transmitted in analog form. The introduction of digital integrated circuits and computers marked a new beginning for video technology - digitalization. Currently, we have all sorts of diverse and complex technologies interlinked with the video domain. This paper intends to give a detailed explanation of one such area related to video streaming, and to be precise, Webcam streaming. Raw video from the webcam has to pass through a whole lot of phases before it is suitable for transmission. The sections following would brush through all the necessary pathways the stream has to flow through to finally be received by the client.

Video Streaming Basics

☞ On-Demand Streaming

With the emergence of Pentium and its successor chips (dies), video and audio play became a reality for consumer-grade PCs. Back then video files were stored on CD-ROMS or downloaded from remote servers. Network delivery of media was still a long reality in those days. By the early 2000s, the Internet saw a booming increase in network bandwidth. With the advent of powerful media compression algorithms and more powerful Personal Computers, streaming delivery of media has become possible. The term streaming is used to describe a method of relaying data over a computer network as a steady continuous flow of bytes, allowing playback to proceed while subsequent data is being received. Instant playback was the greatest advantage posed by live streaming in comparison with 'Download and play' which could cost hours on a slow network.

☞ Live Streaming

Streaming can mainly be of 2 types, On-demand Streaming and Live Streaming. The former deals with previously recorded and compressed media. Media files are stored on specific servers and are delivered to one or more clients on request (on demand). Today we have thousands of such servers ready to stream media files on demand. A few such servers are Youtube, Vimeo, TED, etc.

When it comes to Live Streaming, the entire process starting with the capture of video to the final delivery of processed video is done on the course. This process is highly resource exhaustive. It could require significant computational resources.

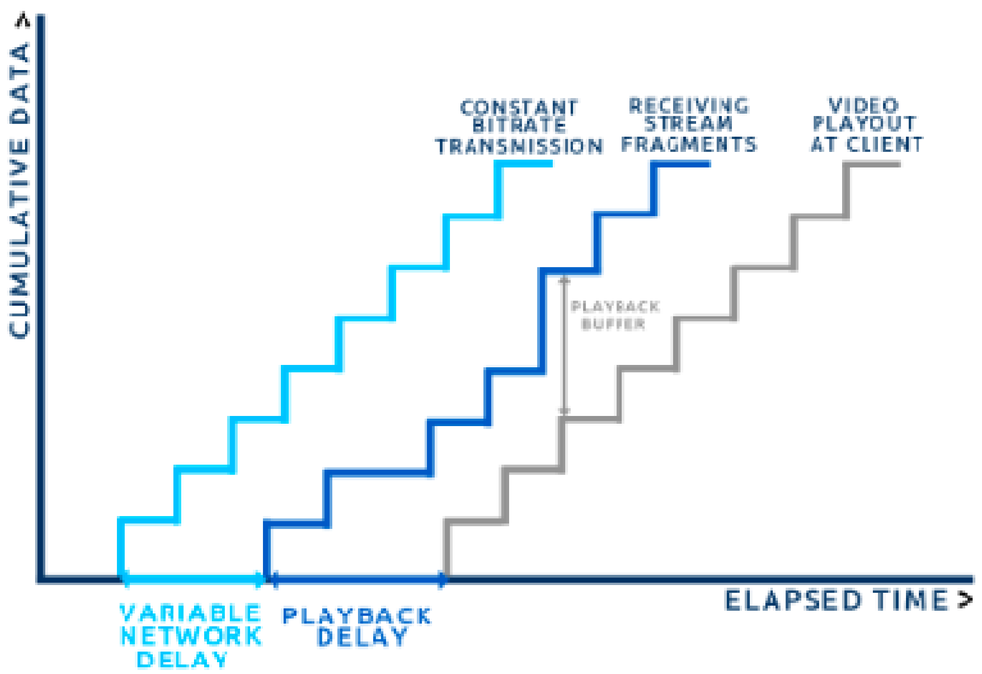

Figure 1: On demand streaming

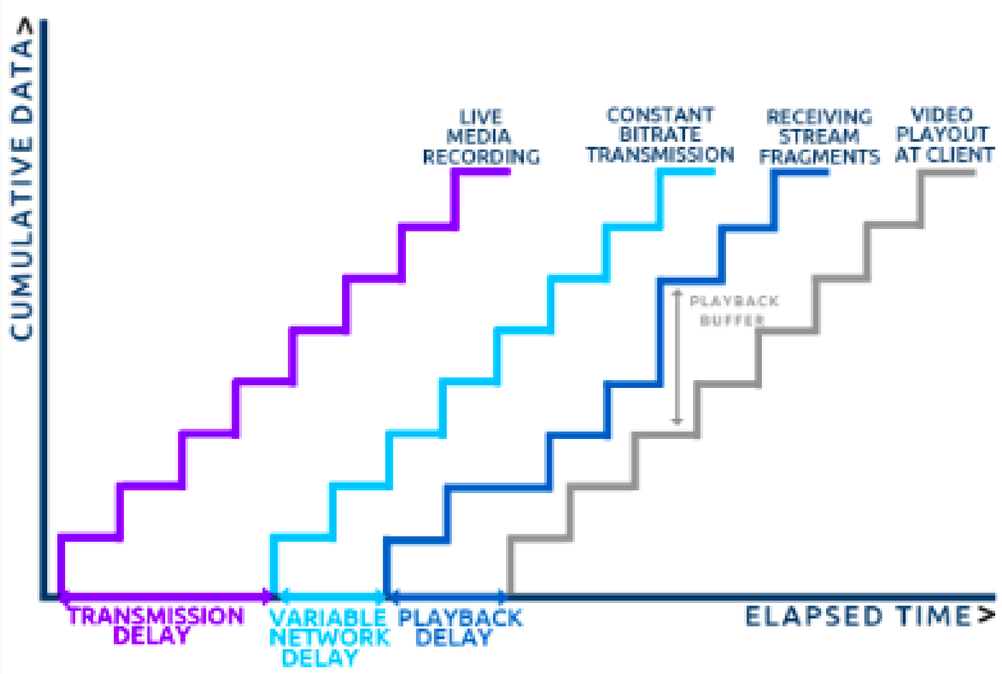

Figure 2: Live Streaming

From the diagram, it is clear that the packets transferred at a constant bitrate reach the receiver in a highly disoriented manner making it difficult for playback. The packets are streamed without jitter by streaming from a playback buffer. This introduces a playback delay. Live Streaming introduces yet another delay for transmission, mainly taking into account video capture and subsequent recording. The media needs to be compressed along the flow before reception at the client side.

Webcam Streaming

Webcams are video cameras that are often embedded into laptops. When it comes to desktops, webcams come in different models with the support driver CD-ROM. Most webcams have standardized applications that can access them for capture/recording purposes.

The webcam streaming application captures the video, compresses it, does some pixel-level manipulations, and later on sends it over the Internet, on-demand to any number of requesting clients (VLC, open RTSP by Live555). Here 'on-demand' doesn't mean storing and playback of the webcam video but rather, producing the live stream when the client requests it. The application can be better explained as an intermediary. It serves as a proxy between the back-end webcam and any number of front-end clients.

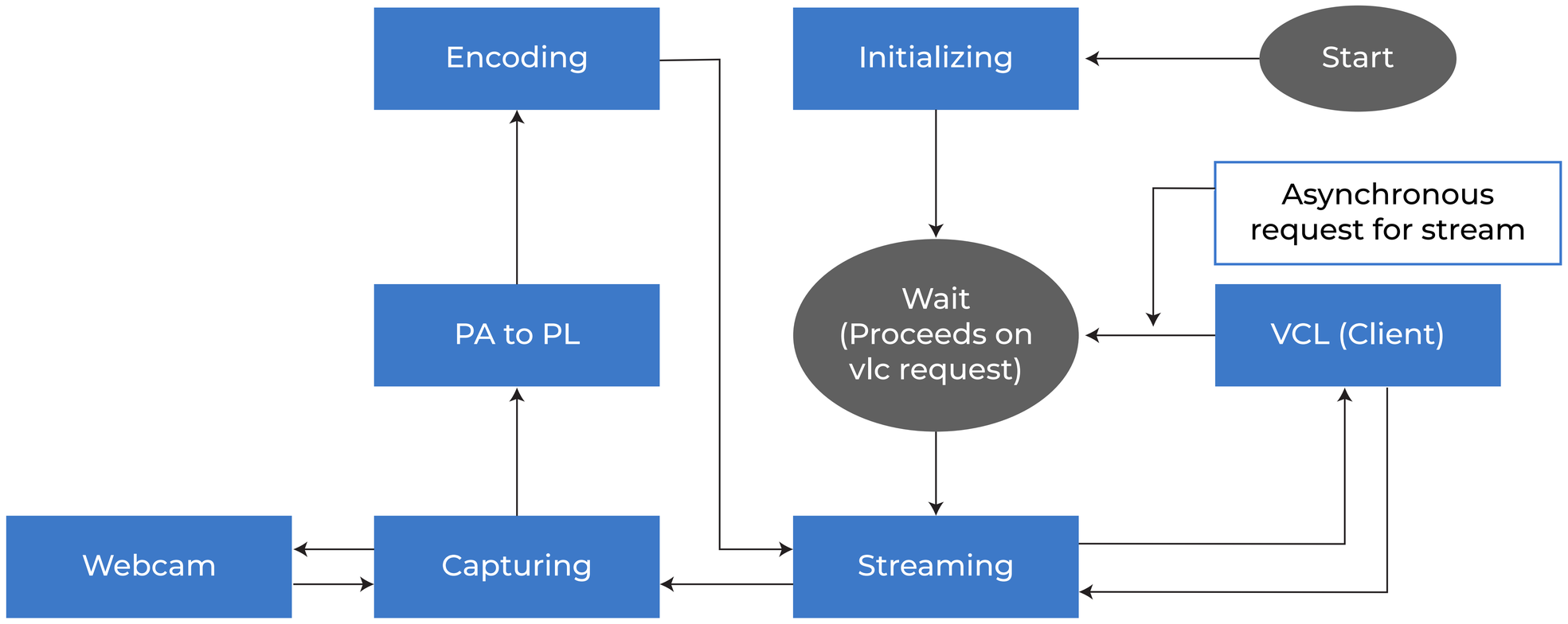

Figure 3: Block Chart

Each client receives a unicast stream from the webcam on demand. The webcam streamer isolates the webcam module (Hardware) from the requesting client (VLC for instance).

The application is a cascaded assembly of 2 major sections: the Capture/Encode section and the Streaming section. This isn't purely a serial arrangement, rather the sections interact with each other accordingly. The block diagram makes it clear that the application is asynchronous.

The initial state of the application can be considered as 'not running' or 'yet to start'. When the application is executed, the Streamer section does the initial job of setting up an RTSP server. Following initialization, the Webcam Streamer enters an event loop. Event loops are programming constructs that wait for and dispatch events or messages in a program. Applications developed using the Live555 library set are event-driven. The application goes into a wait state constantly monitoring a few parameters. When a client says, VLC requests a stream, the event loop breaks the wait and calls the corresponding event handlers. A smooth transfer of control to the Capture/Encode module happens consequently. V4L2 also known as Video4Linux2, a collection of device drivers and API support, helps in the capture of live video streams from the webcam module.

The captured video is transferred as such to a pixel manipulation unit, namely, PA to PL block. The Packed to Planar block converts the raw packed format YUYV data to planar YUV4:2:0. The raw video captured can account for several megabytes/minute depending on the resolution. Fortunately, the Encoder Block does the job of compressing the video to achieve excellent size reduction. H264 video encoding has been used in the application with astounding compression statistics. Raw video capture of 40 MB (50 frames) could be diminished to a few 100 KBs!

The Capture/Encoder module switches its control back to the Streaming Section handing over the compressed video data as network abstraction layer units (NAL units). The streaming module delivers the data to the client as RTP packets until the client closes the session.

RTP-RTCP-RTCP Protocols

The Internet was primitively designed to support data communications. Much of the data transfers were accounted for by emails and file transfers. As the number of nodes and users started increasing exponentially, the need for multimedia communication over the Internet emerged. In response to this, researchers came up with a family of protocols, comprising Real-Time Transmission Protocol (RTP), its control part Real-Time Transmission Control Protocol (RTCP), and the network remote control, Real-Time Streaming Protocol (RTSP).

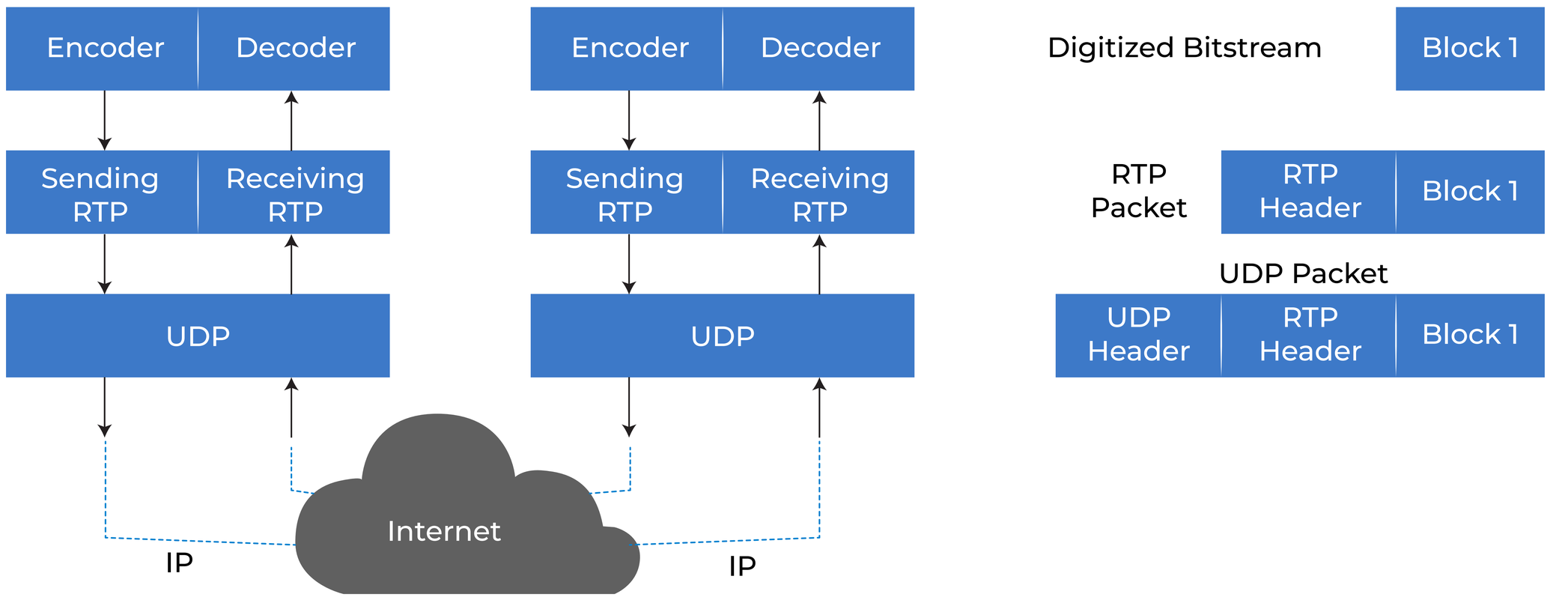

Figure 4: RTP Protocol

RTP, developed by the Internet Engineering Task Force (IETF) is an Internet standard protocol used for transmission of real-time multimedia as well as media on-demand. The RTP standard in effect details a couple of protocols: RTP and RTCP. While RTP is used for the transmission of media across the network, RTCP is used for the exchange of Quality Of Service (QOS) parameters. Though RTP runs over UDP/IP in most scenarios, special techniques can be incorporated to encapsulate them in the TCP/IP layer.

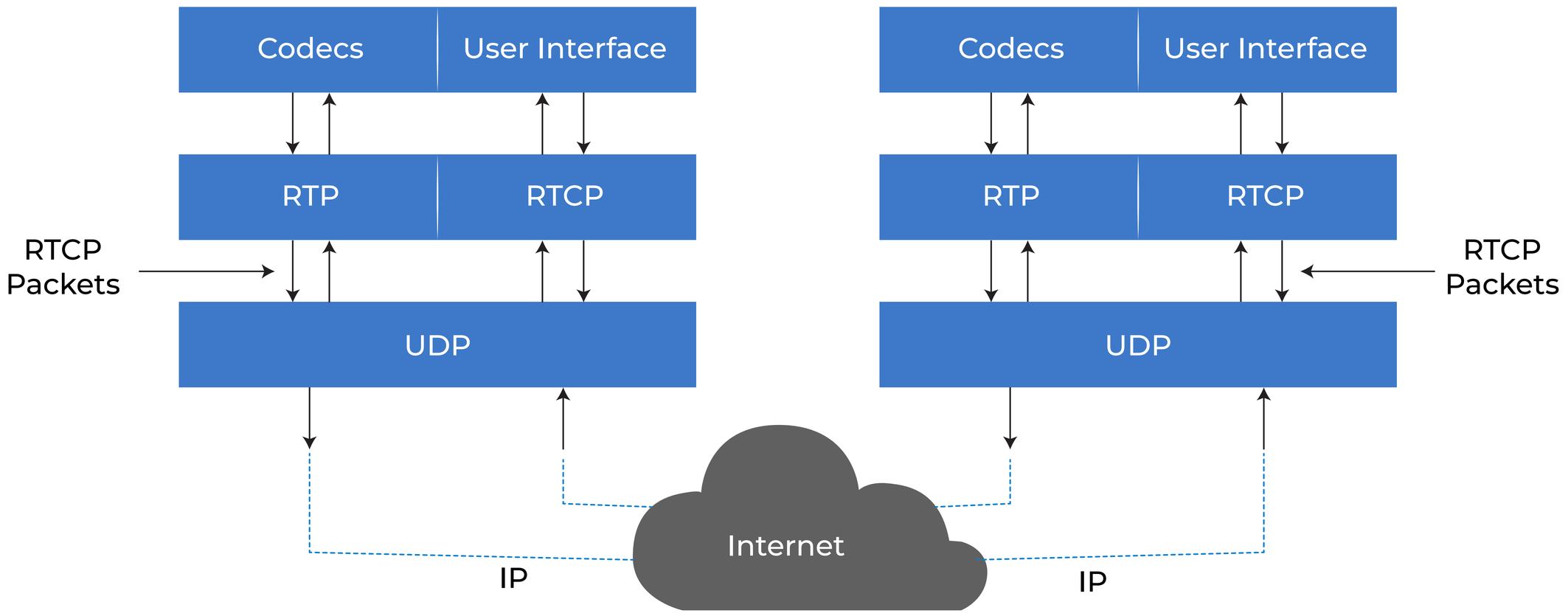

Figure 5: RTCP Protocol

RTCP, the control protocol is designed to work hand in hand with RTP Protocol. RTCP packets contain sender or receiver reports that provide quality of service statistics, such as the number of packets sent, number of packets lost, inter-arrival jitter, etc. If the application is adaptive, the RTCP feedback could save it from predicted congestion by increasing the compression ratio. Thus the feedback RTCP plays a crucial role in diagnostic purposes to localize eventual problems.

RTSP as in RFC 2326, states that it is an application layer protocol that acts as a 'network remote control'. RTSP has control over the transfer of RTP data packets over IP. Controls such as play, pause, record, etc resemble the available functionalities of the present-day iPods, MP3 Players, etc. Though RTSP stays out of packet transmission liabilities, it could enable the interleaved transfer of RTCP-RTP packets. There is no concept of an RTSP connection. Instead, a server maintains a session labeled by an identifier. RTSP sessions are independent of transport layer protocols such as TCP. During an RTSP session, the client may open and close many reliable transport connections (TCP) or it may use connectionless UDP.

Capture/Encode Section

☞ Capture Block

The Capturing block lies very close to the webcam hardware module. In fact, through the underlying webcam driver APIs, the Capturing block can latch the captured chunks of raw video onto any buffer/queue. A good understanding of the V4L2 API set could come in handy while dealing with capture modules. An essential criterion to be noted here is the platform on which the webcam streamer is running.

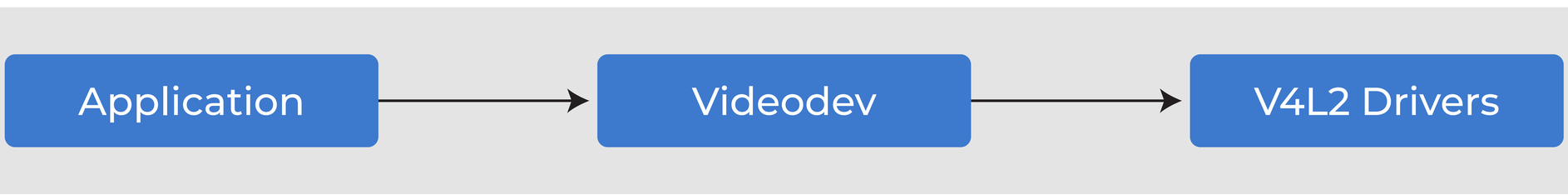

Figure 6: Interaction between user application and V4L2 driver

V4L2 driver API, Video for Linux two, as the name suggests is platform specific (Linux). V4L2 stands for V4L version 2. It is a dual-layered driver system of which the upper layer is the videodev module. The videodev module is registered as a character device with major number 81. Beneath videodev is the V4L2 drivers module. The V4L2 drivers are seen as clients by the videodev module. When a V4L2 driver initializes, it registers each device it will handle with videodev by passing a structure to videodev which contains all the V4L2 driver methods, minor numbers, and a few other details.

Regular Linux driver methods don't differ much from the V4L2 driver methods except for a couple of parameters. When an application invokes a driver call, the control first moves to the videodev method, which translates the inode structure pointers to analogous V4L2 structure pointers, and subsequently calls the V4L2 driver's methods.

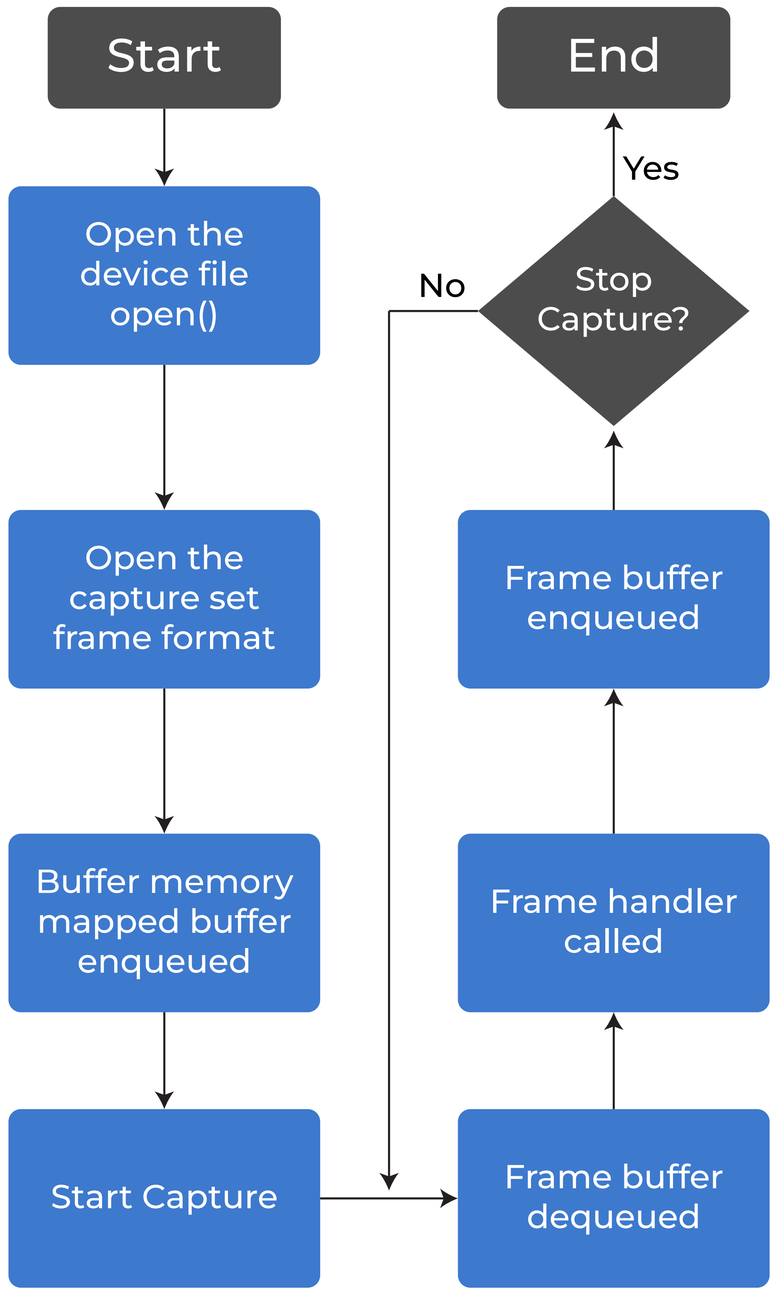

Figure 7: Capture Flow Diagram

In Linux, the default capture device is generally at /dev/video0. Any other video source could result in a different index(/dev/video(x)). The capture has to be opened and queried to make sure the device is available for capture. After the query checks, the frame format has to be implicitly added. Most capture modules allow 2 to 3 different formats by default, though some cameras adamantly cling to the YUYV format. YUYV format comes under the YUV color space in contrast to the RGB format groups present under the RGB color space. Once the format has been specified, buffers are requested and subsequently enqueued. On starting the capture, a chunk of data bytes captured by the device is moved onto the requested buffer. This buffer is dequeued and a suitable handler for moving/processing the frame is called. The process of enqueueing-dequeuing continues until the capture is implicitly stopped.

☞ Color Space

A color space is a mathematical representation of a set of colors. The three most widely used color models are RGB (for computer graphics), YUV (for video systems), and CMYK (for color printing). RGB color model is an additive color model, where red, green, and blue are added together to produce a wide spectrum of colors. A color model is a method for describing a color. Thus the RGB color model uses a combination of red (R), green (G), and blue (B) for describing an individual color. So what difference does it make with color space? Color spaces account for all the different colors a color model can span.

When storing video digitally, either RGB or YUV can be used. Each color model has different formats of their own for storing video files. A few citable examples for YUV are YUYV, YVYU, YV12, etc, and for RGB –RGB16, RGB24, RGB32, etc. To understand the difference between these color spaces, one must have a basic understanding of how the human brain perceives images. RGB stores video intuitively. For each pixel, RGB holds values for red, blue, and green. One of the most commonly used formats, RGB24 allocates 24 bits for each pixel, with 8 bits for each color. Now that is 256 (0-255) different shades of red, green, and blue. The color span for RGB24 can be found with some easy permutations, which would account for 16777216 different colors!

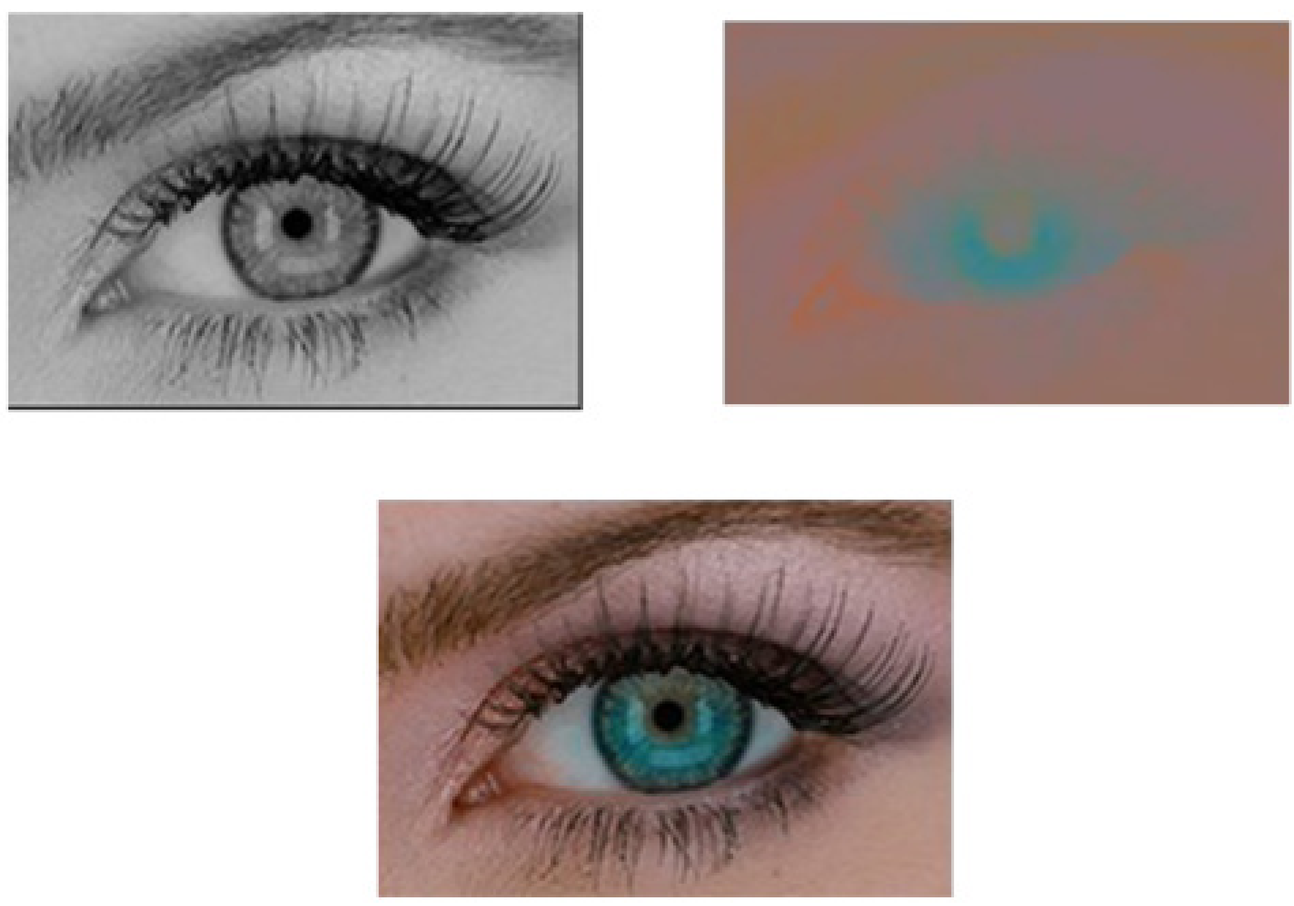

Figure 8: YUV Perception

YUV in contrast stores color the same way as the human brain works. The core component that the brain acknowledges is brightness or luma. Y represents the luma component and can be found out from RGB by averaging color channels with different weights for each channel. Luma is simply a positive value with 0 marking black and 255 marking white. U and V, also known as Cb and Cr stand for the chrominance or colour. Cr color spectrum moves from red at one peak to green at the other whereas Cb moves from blue to yellow. This is one solid reason for the emergence of the concept known as 'impossible colors' which states that it is almost impossible to perceive certain pairs of color as a single color (red+green and yellow+blue).

☞ Packed to Planar block

YUV formats fall into 2 different categories, the packed formats where Y, U, and V components are packed together and stored in a single array, and the planar formats, where each component (Y-U-V) is stored in three different arrays and are later on fused to form the final image, from three distinctive planes. The raw Video output given by Webcam Streamer is YUYV, a packed format. Now Most video encoders directly work on planar formats such as YUV4:2:0. YUV4:2:0 has been chosen as the output video format for PA to PL block. The daunting task of pixel conversion thus has to be met. To develop a suitable algorithm, an in-depth understanding of both these formats is necessary.

YUYV also known as YUY2 is arranged in the memory as shown below

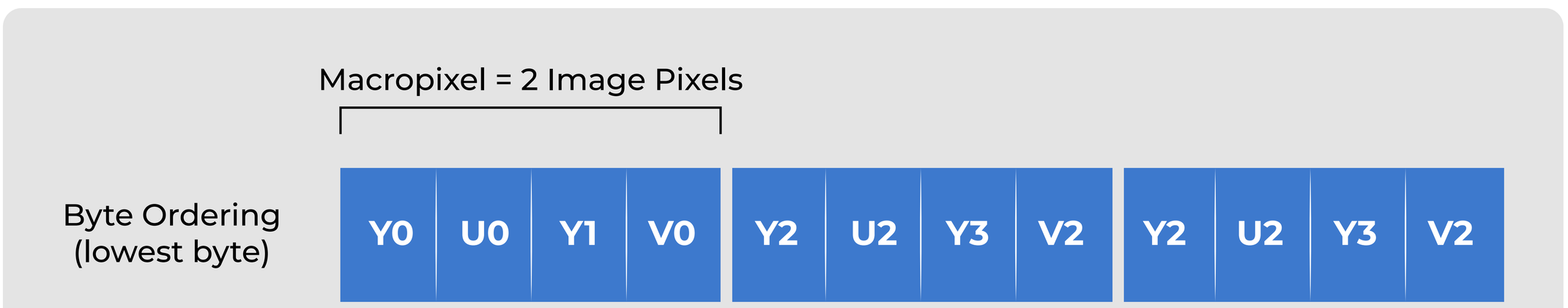

Figure 9: YUYV Representation

In this format each 4 bytes is 2 pixels. 4 bytes would thus represent 2 Y's, 1 U, and 1 V. Y would thus be assigned to both the pixels while Cb and Cr are shared. The diagram explains this well. The first four bytes when grouped have 2 Y bytes, Y0 corresponding to pixel 1 and Y1 corresponding to pixel 2. Both pixels together share the U0 and V0. The alignment is something that should be noted here. All three components; Y-U-V are arranged into a single array in an ordered manner. Packed format YUV finds similarity to YUV4:2:2 in a planar format.

Planar formats are quite different from packed formats. The suffix 4:2:0 represents the chroma subsampling. There are a wide variety of YUV planar formats such as YUV4:2:0, YUV4:1:1, and YUV4:2:2 each having distinctions in the way their chrominance has been sub-sampled. Chroma subsampling can be explained with the help of a sub-sampling scheme as shown below.

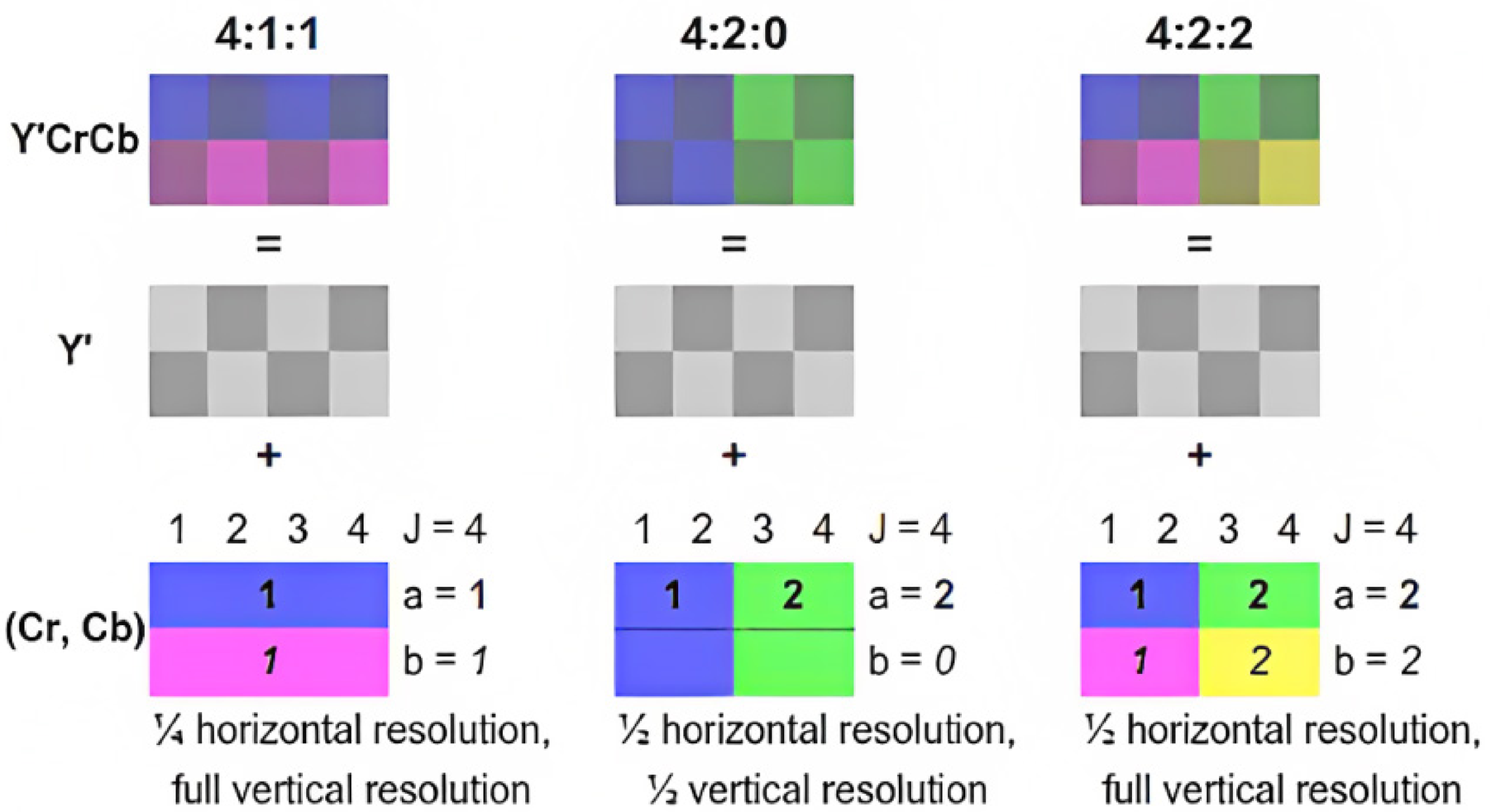

Figure 10: Sub-Sampling Scheme

Planar formats are often expressed as a ratio of three parameters (J:a:b). The scheme considers 2 rows, 4 pixels each for the illustration. The 3 parameters can be defined as J: Horizontal Sampling reference (pixels per row)a: Chrominance samples present in the first row: Number of chrominance sample changes between rows 1 and 2. This clearly explains YUV4:2:0. A slab of 8 pixels as 2 rows of 4 pixels would contain 2 different samples of chroma in the first row with no changes as it moves to the next row. This is shown in the diagram. The memory representation of such a format is employed using 3 distinctive arrays holding Y, U, and V separately. One interesting thing to note here is that each pixel in the YUV4:2:0 image would contain only 12 bytes in contrast to YUYV which contains 16 bytes. Thus there is a significant narrowing in the video data size.

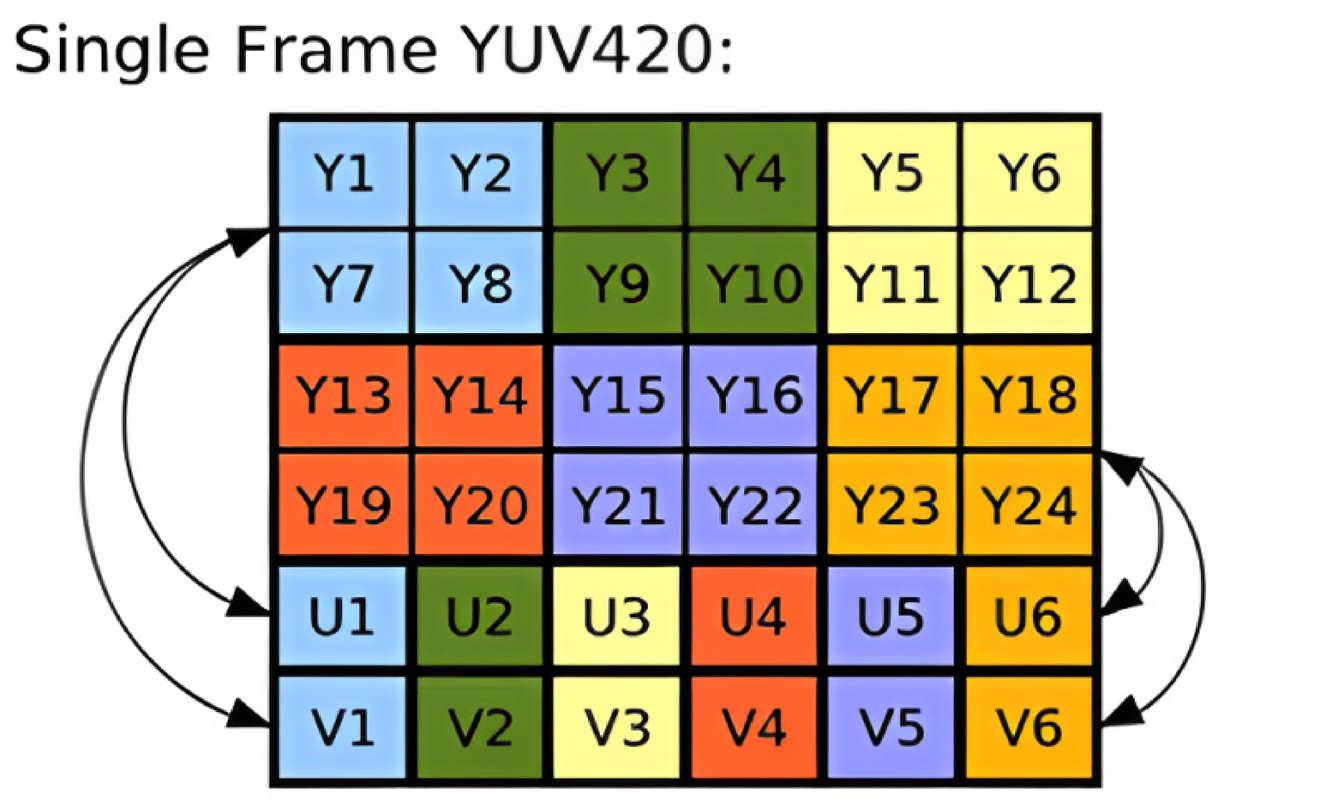

Figure 11: YUV4:2:0 Representation

The PA to PL block does the pixel manipulations on a frame basis. It sequentially takes unit frames as inputs, handles them based on the algorithm, and outputs the frame to the next block in the cascaded assembly. The algorithms have been developed taking into account the component arrangement and chroma weightage.

☞ Encoding Block

Since the early ages of digital images, there has been a demand for compression. Though it is practical to store raw digital images, storing raw video sequences is just not feasible. An hour-long raw High Definition video at 25 frames per second would draw up to 500 GB. Compressing each image on its own would initially seem to soothe the issues, but it won't. To overcome this problem, video compression algorithms have made use of temporal redundancies in video frames. Using such an approach, the value of the current frame could be predicted taking into account the previously encoded/decoded frame.

This technology of video compression is known as the motion compensation approach. MPEG-4 advanced video coding (AVC) standard is one such motion compensation standard. Some of the earlier standards are MPEG-1, MPEG-2, and MPEG-4 advanced simple profile (ASP). MPEG-4 AVC gives almost double the compression rate at the same quality, in comparison with MPEG-4 ASP. MPEG-4 AVC was jointly developed by the Moving Pictures Expert Group (MPEG) and ITU-T Video Coding Experts Group (VCEG). MPEG-4 AVC is also called H.264.

Depending on the H.264 encoding scheme, different frames such as Intra-frame (I-frame), Predictive-frame (P-frame), and Bi-Directional frame (B-frame) may be used. In H.264, different encoding types are slice dependent unlike older standards, which are frame dependent. This implies there are I-Slices, P-Slices, and B-Slices. Intra-frame or I-frame is a self-contained frame that can be independently decoded without any previous frame reference. I-frames have only one single I-Slice and are referred to as instantaneous decoder refresh (IDR) frames. I-frames are needed as starting points or re-sync points. They don't use motion prediction and offer negligible compression gain. P-frames use the previous I-frame as a reference, along with motion prediction. B-frames use one slice from the past (reference) along with one slice from the future (Prediction).

For a single frame, data can be removed by simply moving out unnecessary information. 'Difference coding' is a method used for the reduction of video data between a series of frames, used by H.264 as well as many other encoding standards.

Figure 12: Difference Coding

Difference coding algorithms carefully compare frames to find out redundant data. All the unchanged video data in the current frame in comparison with the older frame are removed or are not transmitted. Thus, a huge reduction of data is possible for video sequences having negligible motion vectors within them. The decoder does the job of uncompressing the encoded video sequence. Most video players have inbuilt decoders within them. De-Compressors do the job of adding back the redundant data to the slice/frames depending upon the slice type/frame type.

The frames on the top are compressed on an individual basis and are transmitted. Motion JPEG is one such format that uses the above-mentioned scheme. Redundant data accumulation is proportional to the number of frames, in this scenario. The frames on the bottom half, suggest an adaption of a difference coding algorithm, where the motion vectors alone are bypassed additively.

The tedious task of encoding the pixel-manipulated YUV4:2:0 video remains. x264 could be utilized in the application for the same. x264 is a free library set used for encoding video streams into MPEG-4 AVC. It provides a wide set of command line interfaces and APIs.

```
if( x264_param_default_preset( &param, "medium", NULL ) < 0 )
        exit(1);

if( x264_param_apply_profile( &param, "baseline" ) < 0 )
        exit(1);

i_frame_size =
    x264_encoder_encode( h, &nal, &i_nal, &pic, &pic_out );
```

Figure 13: x264 API Set

A few important APIs have been chosen for explanation. The x264_param_default_preset() call sets the underlying encoder to compress the video at a specific preset mentioned by the parameter string. A preset is a set of options that would provide a certain encoding speed to compression ratio. “medium” rated encoding should neither be too slow nor too quick. The speed has a relationship with the encoded output quality when it comes to live streaming, as video has to pass through several blocks in quick succession.

The x264_param_apply_profile() call does the job of assigning a profile. The H.264 encoding standard is not one single standard used in all cases, but rather a set of tools. The profiles define which tool has to be selected. Here the string “baseline” suggests an H.264 standard, comprising just the basic features for the compression purpose.

Streaming Section

☞ Initializing Block

The former half of this paper included the application level flow incorporating various blocks. One such unit is the Initialization Block. This block helps in setting up an RTSP server. To exactly understand the need for such a server, the application has to be viewed alternately. The streaming server stands as a proxy between the Webcam module and the end client (say VLC). Thus whenever VLC requests a live webcam stream, the application has to somehow detect this notification and pass on the stream to the client. All these duties are handled by an RTSP Server. Live555 library set is developed in a highly structured manner making it easier for expansion.

The server listens keenly on a predefined port all the while the application is running. Any change would trigger 'callback' functions to carry out all the event handling. 'Callback' is a piece of executable code that is passed on as an argument to some other code. Live555 event handlers use call-back mechanisms for handling events. Live555 library set supports both 'on-demand' streaming as well as live streaming. 'on demand' streaming in most cases is from stored video files. Live streaming on the other hand uses a live video source. The basic structure of an event loop is as follows.

```
while (1) {
    events = getEvents();
    for (e in events)
        processEvent(e);
}
```

Figure 14: Event Loop Model

The server initialization is succeeded by an infinite loop, constantly waiting for events. The implication from the pseudo-code construct is clear; the loop keeps rotating, initially grabbing events and then calling the event handler. Having seen the Basic event loop structure, one can ponder on the method of reception of an event. Three important system calls for dealing with events are select(), poll(), and epoll(). Live555 has incorporated select for monitoring events.

The select() API enables the program to do a few basic tasks: check whether there are any incoming I/Os to be attended. Linux manual page gives a formal description for select as: “select() examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and errorfds to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively”.

```
int select(int nfds,
fd_set *restrict readfds,
fd_set *restrict writefds,
fd_set *restrict errorfds,
struct timeval *restrict timeout);
```

Figure 15: Select API

select() API has some features worth mentioning; Primarily, it checks whether the descriptors can be 'read' from or 'written' to. 'read' lets the server know a new packet has arrived while 'write' notifies the server it's okay to reply to the corresponding descriptor. Second, the timeout parameter can be made null, meaning the select call blocks indefinitely until a monitored descriptor showcases some change.

An RTSP URL is created on behalf of the webcam module by Webcam Streamer Application which is later on used by the client for getting the stream.

☞ Streaming Block

The word streaming would be more than familiar by now. What would happen when VLC requests a stream? What causes the flow of packets across the network? To answer all these questions, a deeper analysis of the Streaming Block is required. It is advised that the reader has a copy of the Live555 library set before proceeding to the coming sections.

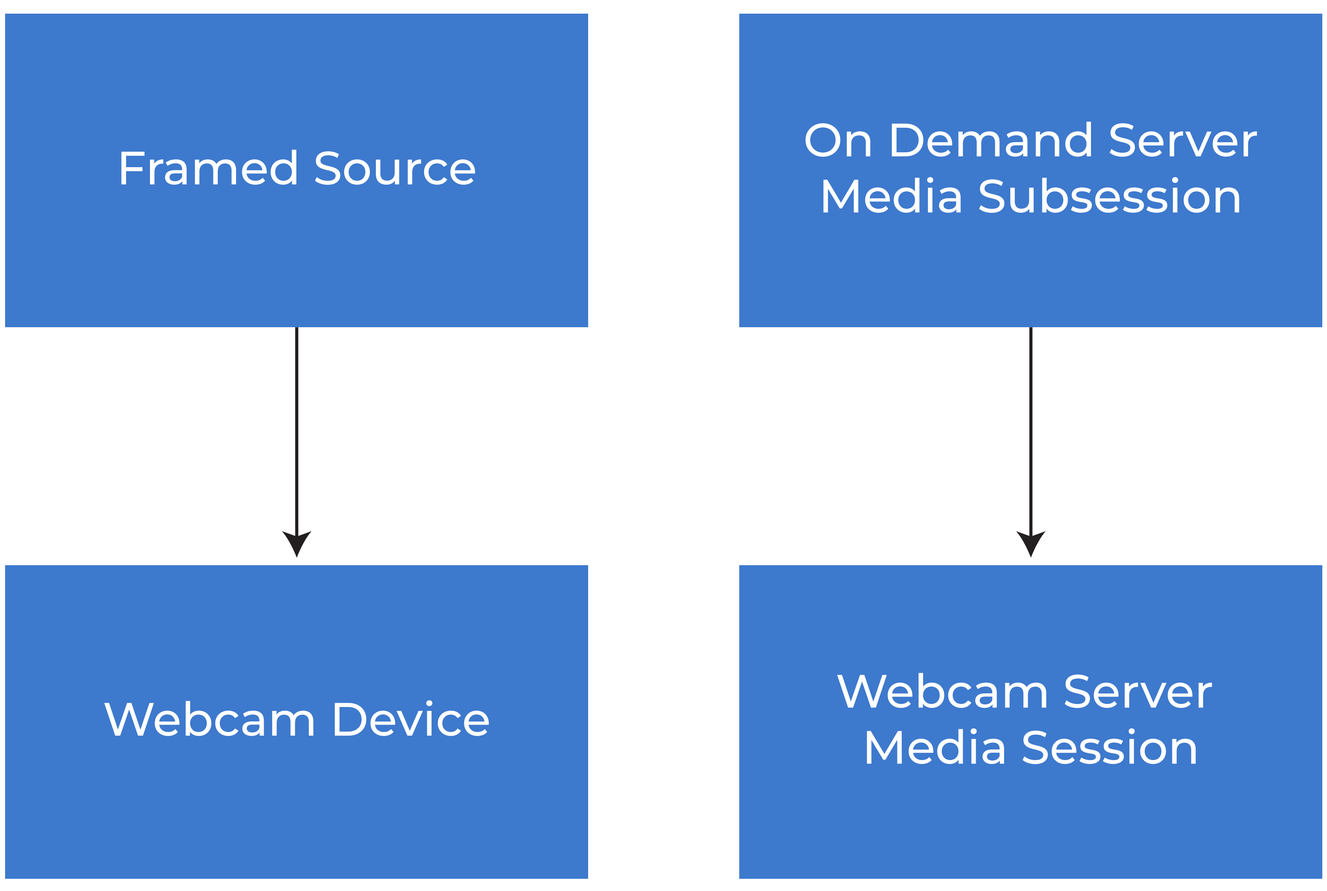

Figure 16: Live555 Sub-classing

The framed source is one among many classes from the Live555 library set which contains important APIs. getNextFrame() API takes in the pointer address of the memory location where NAL units are stored as a parameter to later push it for streaming. Thus direct bridging between the encoder output and Live555 libraries can be made feasible. FramedSource has been subclassed to exhibit a suitable implementation for the webcam module. This sub-class, WebcamDevice, holds functions that directly invoke the webcam which in turn drives other functions for pixel manipulation as well as encoding. Video taken in, on a frame basis by the encoder is encoded and stored to a suitable memory location. The address of each independent NAL unit stored is pushed onto a queue which is accessed inside the deliverFrame() function inside WebcamDevice class.

```
void WebcamDevice::deliverFrame() {
    if(!isCurrentlyAwaitingData())
    return;
    x264_nal_t * nal = nalQueue.front();
    nalQueue.pop();
    assert(nal->p_payload != NULL);
    int truncate = 0;
    ****NAL HEADER PARSING DONE HERE****
    if(nal->i_payload-truncate >(signed int)fMaxSize) {
        fFrameSize = fMaxSize;
        fNumTruncatedBytes =
        nal->i_payload-truncate - fMaxSize;
        }
    else
        fFrameSize = nal->i_payload-truncate;
    fPresentationTime = currentTime;
    memmove(fTo,nal->p_payload + truncate,fFrameSize);
    FramedSource::afterGetting(this);
```

Figure 17: NAL Unit Transfer

The NAL headers are carefully parsed and the payload is separated. This is given to the getNextFrame API, present in the parent class. GetNextFrame() calls doGetNextFrame() which in turn calls afterGetting(). afterGetting() cyclically calls getNextFrame() and this cycle continues. One important thing to note here is that doGetNextFrame() has been virtually defined allowing it to be implemented as per the user's choice if FramedSource is to be sub-classed. This is exactly being done by the Webcam-Device subclass with its implementation of doGetNext-Frame(), probing the NAL queue to see if it is empty.

The memmove() call copies the NAL unit address onto fTo, a char pointer member within FramedSource. Note that afterGetting() has been called from the WebcamDevice class to pass on this object containing the NAL unit address to the parent class FramedSource. If doGetNextFrame() results in an empty queue with no NAL units, it returns without blocking. Later when NAL units arrive in the queue, the event loop handles it as an event.

OnDemandServerMediaSubsession is yet another class that has been sub-classed. WebcamServerMediaSession, the child class, does the job of transferring control to WebcamDevice class. Though an instance of WebcamServerMediaSession is created initially, the transfer of control takes place later on, after the RTSP handshakes.

Source	Destination	Protocol	Length	Info
192.168.239.27	192.168.239.190	RTSP	196	OPTIONS
192.168.239.190	192.168.239.27	RTSP	220	Reply: F
192.168.239.27	192.168.239.190	RTSP	222	DESCRIBE
192.168.239.190	192.168.239.27	RTSP/SDP	687	Reply: F
192.168.239.27	192.168.239.190	RTSP	249	SETUP rt
192.168.239.190	192.168.239.27	RTSP	261	Reply: F
192.168.239.27	192.168.239.190	RTSP	232	PLAY rts
192.168.239.190	192.168.239.27	RTCP	1516	Sender F
192.168.239.190	192.168.239.27	RTP	1516	PT=Dynam

Figure 18: RTSP Handshakes

192.168.239.27 is the client IP(VLC), and 192.168.239.190 is the server IP. RTSP Handshake in this scenario has been initiated by OPTIONS which shows various other types of requests. After a few rounds of requests and responses, a SETUP is passed by the client to the server. Each RTSP command is handled by the event loop, as mentioned in the previous section. On handling SETUP, createNewStreamSource(), a virtual function inside OnDemandServerMediaSubSession is called. The derived class, WebcamServerMediaSession has an alternate definition of this same function which in effect routes the call to this class.

WebcamDevice class constructor is being called inside createNewStreamSource(), defined inside WebcamServerMediaSession. This constructor initializes the webcam module and starts pushing NAL pointers into the queue. The constructor in turn triggers the call to deliverFrame().

Future Works

Live streaming is a broad domain. Time constraints have always been a problem for streaming live. A research-based analysis for the reduction of latency could make the Webcam Streamer more robust. This enhancement is highly prioritized. Quick responses to clients are always an add-on. Incorporating audio capture into the streamer would help the streamer to look full-fledged. Lateral work in this area is also preferred.

logic fruit Technologies

Thank You!

Does anyone have any questions?

Gurugram (Headquarter)

806, 8th Floor
BPTP Park Centra Sector–30,
NH–8 Gurgaon – 122001
Haryana (India)

info@logic-fruit.com

+91-0124 4643950

Bengaluru (R&D House)

Sy. No 118, 3rd Floor,
Gayathri Lakefront,
Outer Ring Road, Hebbal,
Bangalore - 560 024

sales@logic-fruit.com

+91 80-69019700/01

United States (Sales Office)

Logic Fruit Technologies
INC 691 S Milpitas Blvd
Ste 217 (Room 9)Milpitas
CA 95035

info@logic-fruit.com

+1-408 338 9743