VVDN TECHNOLOGIES

WEBCAM BASED LIVE STREAMING
A CASE STUDY
SOFTWARE PERSPECTIVE

Concept to Production

ROHIT PHILIP MATHEW
VVDN TECHNOLOGIES LTD
EMBEDDED SOFTWARE ENGINEER
ROHIT.MATHEW@VVDNTECH.IN

www.vvdntech.com

Abstract

Applications that directly or indirectly stream webcam video are commonly available today. This paper intends to give a detailed review on an application built using Live555 and x264 library set, supported by V4L2 driver APIs, that could stream out live webcam video to any number of requesting clients as per request. The different modules necessary for live webcam streaming would be uncovered in this presentation in a sequential manner. The analysis put forth incorporates the issues circling the streamer application as well.

Introduction

It has been a few decades now since video technology has come into action. In the earlier 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 on one such area related with 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.

I. Video Basics

A. On Demand Streaming

With the emergence of Pentium and its successor chips (dies), video and audio playing were becoming a reality for the consumer grade PC's. Back then video files were stored in CD-ROMS or were downloaded from remote servers. Network delivery of media was still a longed reality in those days.By the early 2000's , Internet saw a booming increase in network bandwidth. With the advent of powerful media compression algorithms and more powerful Personal Computer's, streaming delivery of media had 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 on comparison with 'Download and play' which could cost hours on a slow network.

B. 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 at 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 of such servers are You tube, 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 resource often using specific

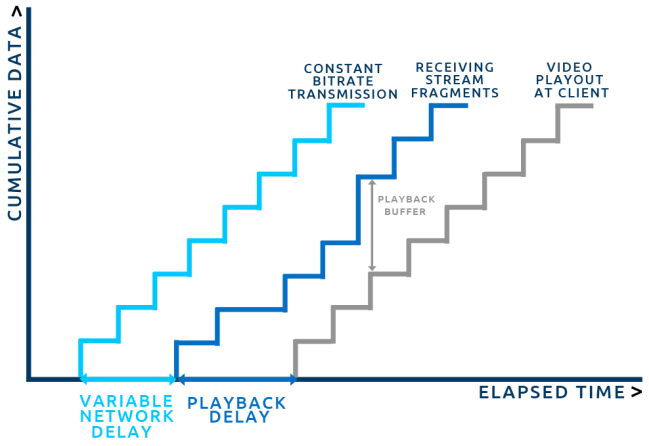

Fig 1: On Demand Streaming

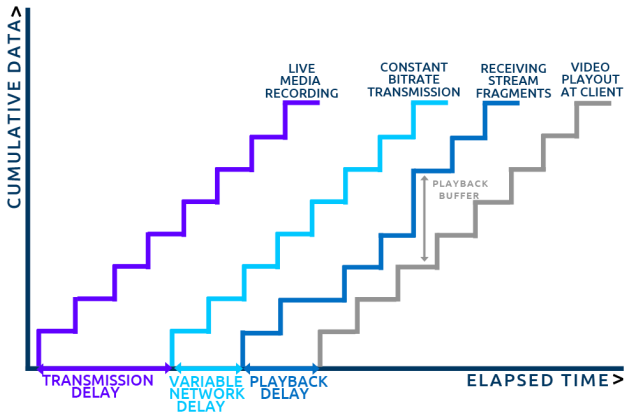

Fig 2: Live Streaming

From the diagram it is clear that the packets transferred at a constant bitrate reaches 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 the playback delay. Live Streaming introduces yet another delay for transmission, mainly taking into account the video capture and subsequent recording. The media needs to be compressed along the flow before reception at the client side.

II. Webcam Streaming

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

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 a stored playback of the webcam video but rather, producing the live stream when the client requests for 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.

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

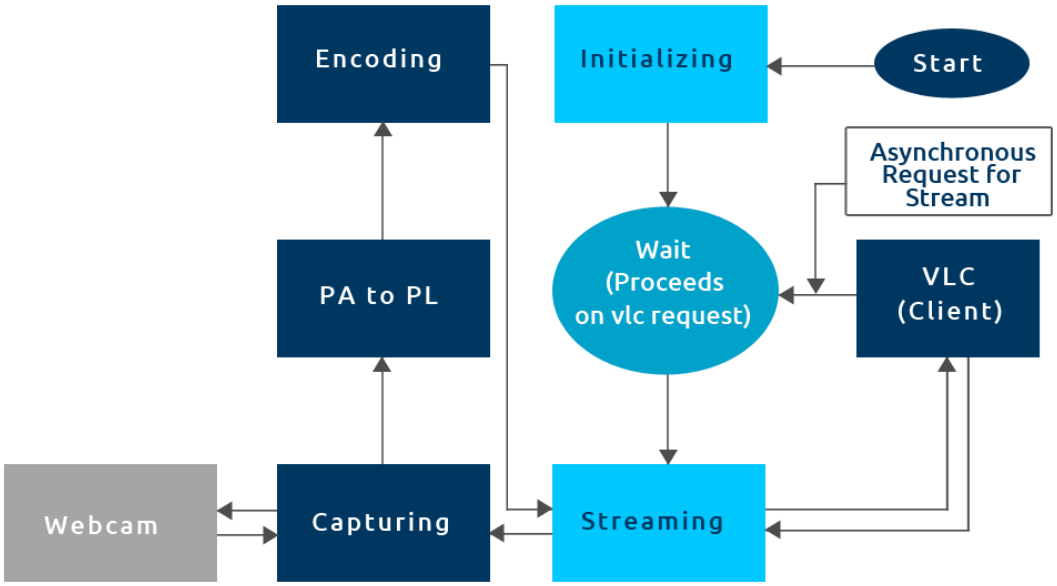

Fig 3: Block Chart

The application is a cascaded assembly of 2 major sections: Capture/Encode section and 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. Looking into the diagram, the yellow blocks encompasses the Capture/Encode section and the blue ones, Streaming section. The red ones are external to the application and the purple ones shows the state of the application.

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 the initialization, the Webcam Streamer enters an event loop. Event loops are programming constructs that waits for and dispatches events or messages in a program. Applications developed using Live555 library set are event driven in nature. The application goes into a wait state constantly monitoring a few parameters. When a client say, VLC requests for 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 stream 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. This will be explained in depth in the coming sections. The raw video captured can account to 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 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.

The application is a cascaded assembly of 2 major sections: Capture/Encode section and 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. Looking into the diagram, the yellow blocks encompasses the Capture/Encode section and the blue ones, Streaming section. The red ones are external to the application and the purple ones shows the state of the application.

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 the initialization, the Webcam Streamer enters an event loop. Event loops are programming constructs that waits for and dispatches events or messages in a program. Applications developed using Live555 library set are event driven in nature. The application goes into a wait state constantly monitoring a few parameters. When a client say, VLC requests for 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 stream 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. This will be explained in depth in the coming sections. The raw video captured can account to 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 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.

III. RTP-RTCP-RTCP Protocols

The Internet primitively was designed to support data communications. Much of the data transfers were accounted from e-mails and file transfers. As the number of nodes and users started increasing exponentially, the need for multimedia communication over the Internet emerged[2]. 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).

RTP, developed by 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 transmission of media across the network, RTCP is used for exchange of Quality Of Service (QOS) parameters. Though RTP runs over UDP/IP in most scenarios, special techniques can be incorporated to encapsulate them over the TCP/IP layer.

This has to be done with utmost care due to the stern nature of latter transport layer.

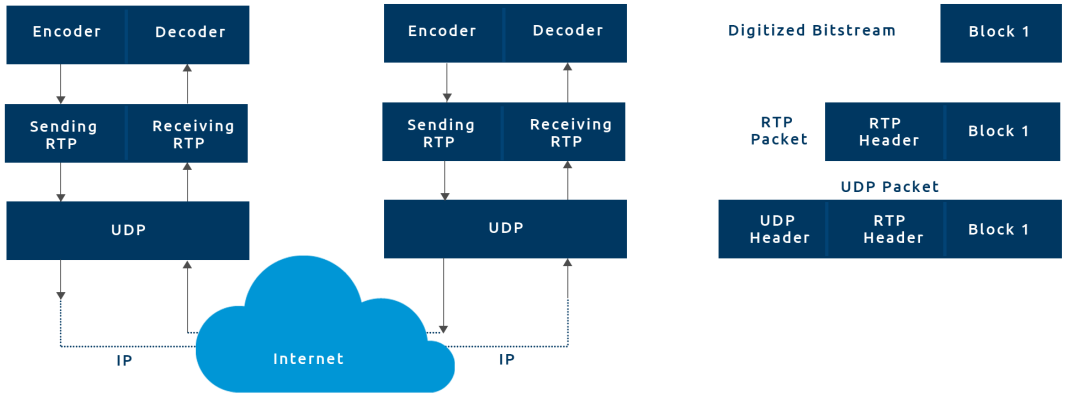

Fig 4. RTP Protocol

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 send, number of packets lost, inter arrival jitter etc. If the application is adaptive in nature, the RTCP feedback could save it from a predicted congestion by increasing the compression ratio. Thus the feedback RTCP plays a crucial role in diagnostic purposes to localize eventual problems[2].

RTSP as in RFC 2326[3], states that it is an application layer protocol which 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 resembles the available functionalities of the present day iPods, Mp3Players etc. Though RTSP stays out of the packet transmission liabilities, it could enable 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 alternatively, it may use connectionless UDP[3]. Real Networks, Columbia University and Netscape Communication jointly came up with RTSP[2].

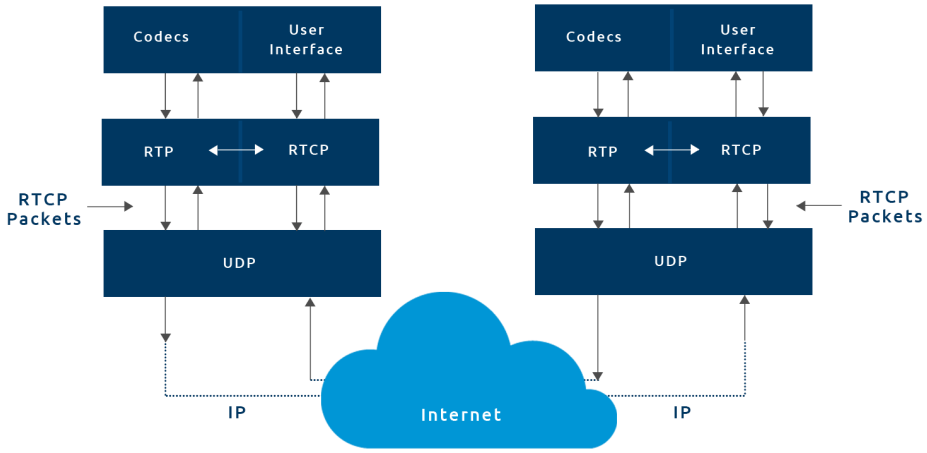

Fig 5: RTCP Protocol

V. Capture/ Encode Section

A. Capturing Block

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 on to any buffer/ queue. A good understanding of V4L2 API set could come handy while dealing with capture modules. An essential criteria to be noted here is the platform on which the webcam streamer is running. 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 number and few other details [4]. 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. videodev acts as a thin shell around the V4L2 drivers [4].

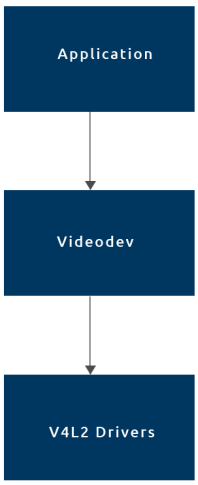

Fig 6: Interaction between User application and V4L2 driver

The user mode view of V4L2 is totally different. A large set of interface calls that could be used with C, Cpp and python have been officially documented. A programmer's perspective description of the same is presented below [5].

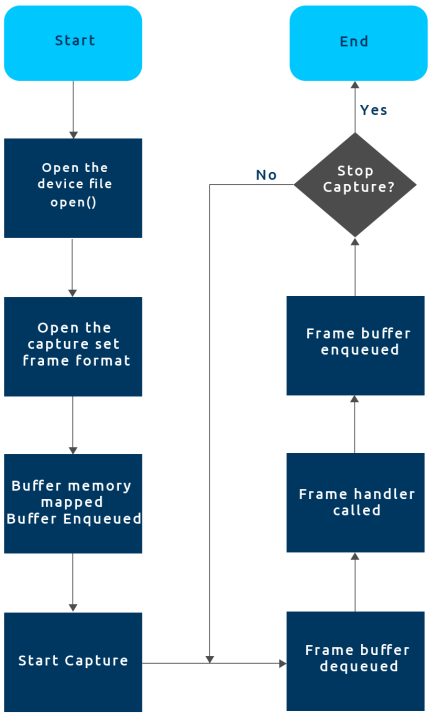

Fig 7: capture Flow Diagram

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 have to be implicitly added. Most capture modules allow 2 to 3 different formats by default, though, some cameras adamantly cling onto YUYV format. The webcam streamer application has been developed on a platform that outputted YUYV formats irrespective of software manipulations. YUYV format comes under the YUV colour space in contrast to the RGB format groups present under RGB colour space. A detailed account on colour spaces would be presented later on. Once the format has been specified, buffers are requested and subsequentlty 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-dequeueing continues unti the capture is implicitly stopped.

```
fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.width       = M;
fmt.fmt.pix.height      = N;
fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
fmt.fmt.pix.field       = V4L2_FIELD_INTERLACED;

if (-1 ==  this->xioctl(fd, VIDIOC_S_FMT, &fmt))
    this->errno_exit("VIDIOC_S_FMT");
```

Fig 8: Code Snippet

B. 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) [7]. 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 RGB color model uses 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 colour models have 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 colour spaces, one must have a basic understanding of how the human brain perceives images. RGB stores video intuitively. For each pixel, RGB holds value for red, blue and green. One of the most commonly used formats, RGB24 allocates 24 bits for each pixels, with 8 bits for each colour. Now that is 256 (0-255) different shades of red, green and blue. The colour span for RGB24 can be found out with some easy permutations, which would account to 16777216 different colours!

YUV in contrast stores colour the same way as human brain works. The core component that brain acknowledges is brightness or luma. Y represents the luma component and can be found out from RGB by averaging colour 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 stands 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 colours' which states that it is almost impossible to perceive certain pairs of colour as a single colour (red+green and yellow+blue).

When dealing with an image, Y can be taken as the 'black and white' part of the image while U - V as the 'coloring' part. Looking at the above cluster, it is evident that it is much more easier for a person to make out the image from the luma alone than from the chroma counterpart. This is exactly what YUV color model intends to do; to pass on the luma as such with lower amounts of chroma content (different for different chroma subsampling).

Why use YUV can be a question pondering many, after getting to know about the different colour models. The older TV sets could not support colour components. Plugging in the Y channel could easily do the trick in this scenario as the B/W television takes luma as black and white ignoring U-V channels. Thus YUV is backward compatible. Yet another reason is size reduction per pixel. Chroma components can be averaged out by a factor, without causing much quality loss for the frame/ video.

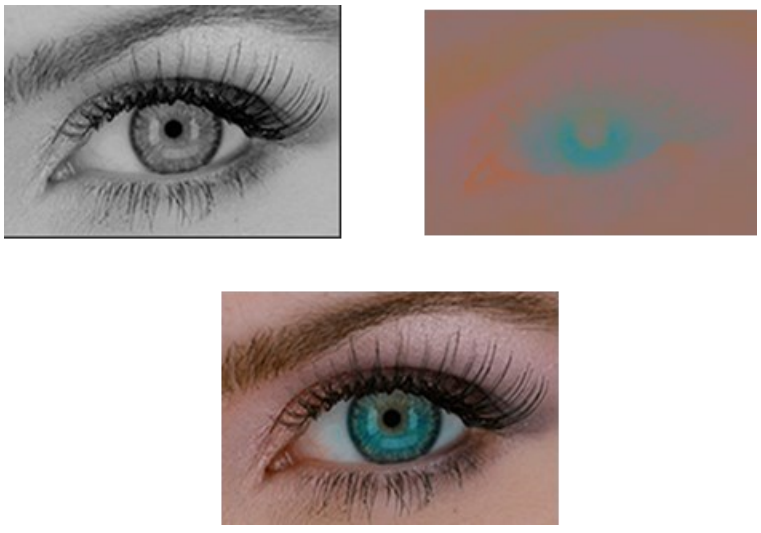

Fig 9: YUV Perception

C. 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) are stored in three different arrays and are later on fused together to form the final image, from three distinctive planes[13]. 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. In order develop a suitable algorithm, an in depth understanding of both these formats are necessary.

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

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 together has 2 Y bytes, Y0 corresponding to pixel 1 and Y1 corresponding 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 in to a single array in an ordered manner. Packed format YUV actually finds similarity to YUV4:2:2 in planar format.

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

Fig 10: YUYV Representation

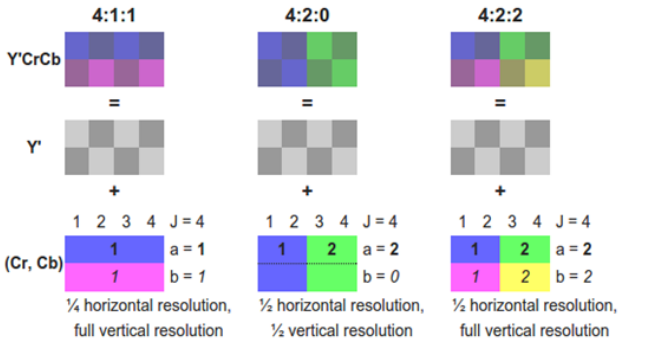

Fig 11: Sub-sampling Scheme

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

b: Number of chrominance sample changes between row 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 is 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 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.

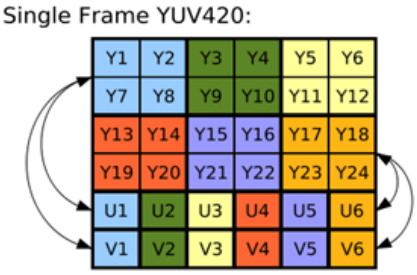

Fig 12: YUV4:2:0 Representation

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

- Let the resolution of the video be considered as M*N
- The input frame array can be safely assumed to be of size [M*N*2] (16 bytes per pixel).
- The output frame smaller in size and can be taken as [M*N*3/2] (12 bytes per pixel). This is separately stored as 3 different a rays.
- The Y array would take up [M*N] bytes leaving behind [M*N/2] for U and V together
- From the pixel sub-sampling scheme for YUV4:2:0, it is clear that U and V hold [M*N/4] bytes each.

- From the packed cluster of bytes, take out all the even bytes starting from index 0 in an ordered manner to form the Y plane.
- Average out the U and V component from the odd and even lines to reduce 4:2:2 chroma sub-sampling to 4:2:0 sub-sampling. The vertical resolution is reduced to half.

```
for (i = 0; i != M * N ; ++i)
    pic.img.plane[0][i] = frame3->YUYVSource[2*i];
for (j = 0; j != M * N / 4 ; j++)
    {
        pic.img.plane[1][j]     =
        (frame3->YUYVSource[4*j + 1] +
        frame3->YUYVSource[M + 4*j + 1]) /2;

        pic.img.plane[2][j]     =
        (frame3->YUYVSource[4*j + 3] +
        frame3->YUYVSource[M + 4*j + 3]) /2;
    }
```

Fig 13: Algorithm Implementation

D. ENCODING BLOCK

Since the early ages of digital image, there had been a demand for compression. Though it is practical to store raw digital images, storing raw video sequence is just not feasible. An hour long raw High Definition video at 25 frames per seconds would draw upto 500 GB[9]. Compressing each image on its own would initially seem to soothe out the issues, but actually it wont. To overcome this problem, video compression algorithms have made use of the temporal redundancies in video frames. Using such an approach, the value of current frame could be predicted taking into account the previously encoded/decoded frame.

This technology of video compression is known as motion compensation approach. MPEG-4 advanced video coding (AVC) standard is one such motion compensation standard[9]. Some of the earlier standards are MPEG-1, MPEG-2, MPEG-4 advanced simple profile (ASP). MPEG-4 AVC gives almost double the compression rate at the same quality, on comparison with MPEG-4 ASP. MPEG-4 AVC was jointly developed by Moving Pictures Expert Group (MPEG) and ITU-T video Coding Experts Group (VCEG). MPEG-4 AVC is also called as H.264.

Webcam Streamer Application has adopted H.264 encoding scheme to compress video data. The coding structure can be explained as follows. The video sequence is split up into frames which are further dissected into slices. A slice is a group of macroblocks, which is a block of 16x16 pixels. Macroblocks are subdivided into sub-macroblocks of 16x8, 8x16, 8x8 pixels. Sub-macro blocks are further broken down to structures of 8x4, 4x8, 4x4 pixels. The 4x4 is the smallest block and the basic encoding unit in MPEG-4 AVC standard [9]. Depending on the Encoding Format, chrominance – luminance informations vary. For YUV4:2:0, a 2x2 pixel block would have chroma stored in a sub-sampled scheme. Each such block would have 1 byte of luma per pixel and 1 byte of chroma per block.

Depending on the H.264 encoding scheme, different frames such as Intra-frame (I-frame), Predictive-frame (P-frame), Bi-Directional frame (B-frame) maybe 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, 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) frame. 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 reference, along with motion prediction. B-frames use one slice from the past (reference) along with one slice from the future (Prediction) [8].

For a single frame, data can be removed by simply moving out unnecessary information. 'Difference coding' is a methods used for for reduction of video data between a series of frames, used by H.264 as well as many other encoding standards [8]. Difference coding algorithms carefully compare frames to find out redundant data. All the unchanged video data in the current frame on 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.

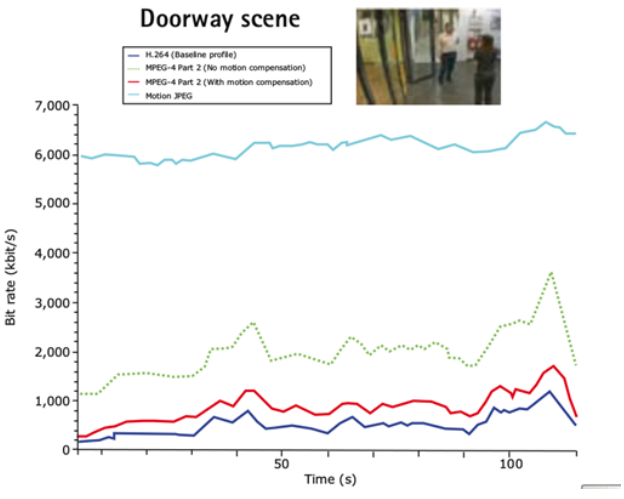

Fig 14: Compression Efficiency

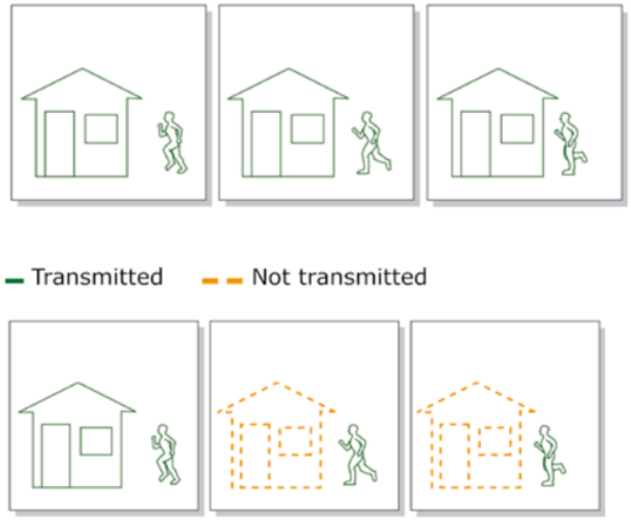

Fig 15: Difference Coding

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

The tedious task of encoding the pixel manipulated YUV4:2:0 video still 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 );
```

Fig 16: 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 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 [14].

x264_encoder_encode API does the encoding based on all the above mentioned criterion. It converts the video data into Network Abstraction Layer (NAL) units, which then can be used for network streaming. According to RFC-3984 [10], a distinction has been made between Video Coding layer (VCL) and Network Abstraction Layer (NAL). VCL incorporates the signal processing functionality of the Codec. This includes transform, quantization, motion compensation prediction and loop filter. VCL encoder outputs encoded slices, which are subdivisions of frame. Webcam Streamer application needs H.264 encoded packets ready to be transmitted on the network. This calls for an additional step of data handling. The NAL encoder comes into action when H.264 encoded slice has to be further encapsulated to be pushed on to the network. X264 library set has a NAL encoder section, which takes in the VCL encoded slice units to give out NAL encoded NAL units [10].

VI. Streaming Section

A. Initializing Block

The former half of this paper had included the application level flow incorporating various blocks. One such unit is the Initialization Block. This block helps in setting up an RTSP server. Webcam Streamer Application extensively depends on Live555, a set of libraries for multimedia streaming purpose. To exactly understand the need for such a server, the application has to be viewed in an alternate manner. Webcam Streamer stands as a proxy between the Webcam module and the end client (say VLC). Thus whenever VLC requests for 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 are 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);
}
```

Fig 17: Event Loop Model

The server initialization is succeeded by an infinite loop, constantly waiting for events. 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(), epoll(). Live555 has incorporated select for monitoring events.

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);
```

Fig 18: Select API

select() API has some features worth mentioning; Primarily, it checks whether the descriptors can be 'read' from or 'written' to [11]. 'read' lets the server know a new packet has arrived while 'write' notifies the server its 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.

B. Streaming Block

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

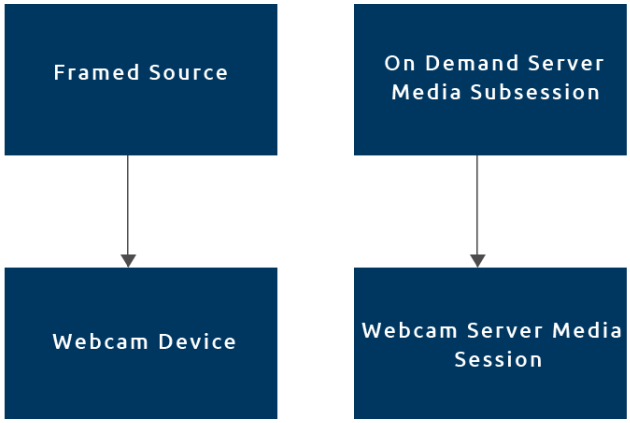

Fig 19: Live555 Sub- classing

Framed source is one among many classes from 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 on push it for streaming . Thus direct bridging between the encoder output and Live555 libraries can be made feasible. As suggested by Ross, architect of Live555 library set, FramedSource has been sub-classed to exhibit a suitable implementation for the webcam module. This sub-class, WebcamDevice, holds functions which 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 are 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.

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

```
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);
```

Fig 20: NAL Unit Transfer

The memmove() call copies NAL unit address onto fTo, a char pointer member within FramedSource. Note that after-Getting() had been called from WebcamDevice class to pass on this object containing the NAL unit address onto the parent class FramedSource. If doGetNextFrame() results in an empty queue with no NAL units, it returns without blocking. Later when NAL units arrive into the queue, 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:
192.168.239.27	192.168.239.190	RTSP	222	DESCRIBE
192.168.239.190	192.168.239.27	RTSP/SDP	687	Reply:
192.168.239.27	192.168.239.190	RTSP	249	SETUP rt
192.168.239.190	192.168.239.27	RTSP	261	Reply:
192.168.239.27	192.168.239.190	RTSP	232	PLAY rts
192.168.239.190	192.168.239.27	RTCP	1516	Sender
192.168.239.190	192.168.239.27	RTP	1516	PT=Dynam

Fig 21: 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 few rounds of requests and responses, a SETUP is being passed by the client to the server. Each and every RTSP command is handled by the event loop, as mentioned in the previous section. On handling SETUP, createNewStream-Source(), a virtual function inside OnDemandServerMediaSub-Session is called. The derived class, WebcamServerMedia-Session 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().

VII. Practical Issues

Pixel manipulation as such is a challenging task. The accuracy of chroma – luma merge depends on the stability of the algorithm used. One of the noted issues was an out-of sync, at times, for the chroma luma boundaries. Colour spread also appeared randomly clogging the video sequence.

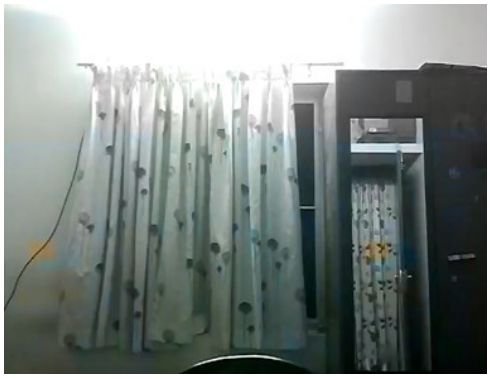

Fig 22: Slightly Distorted Chroma Channel

Yet another issue; latency. This problem had been evident mainly for the first client requesting the stream, as webcam is turned on only on request from the client. The delay is an accumulated sum of network delay, playback delay, transmission delay. Delay reduces with lower quality encoded video transmission.

VIII. Furure Works

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

REFERENCES

Pixel manipulation as such is a challenging task. The accuracy of chroma – luma merge depends on the stability of the algorithm used. One of the noted issues was an out-of sync, at times, for the chroma luma boundaries. Colour spread also appeared randomly clogging the video sequence.

- *Andrew Fecheyr-Lippens, "A review of HTTP live streaming"*
- *Arjan Durresi and Raj Jain, RTP RTCP and RTSP - Internet protocols for real-time multimedia communications.*
- *H. Schulzrinne, A. Rao and R. Lanphier, "Real time streaming protocol (RTSP) " RFC 2326, April 1998.*
- *Bill Dirks, Luc Gallant , "Video for Linux two – driver writers guide", November 9 2005.*
- *Li Hui-jun, "Design and implementation of mobile video surveillance system based on V4L2" in press.*
- *Douglas A. Kerr, "Chrominance sub-sampling in digital images," Copyright 2005, 2012 Douglas A. Kerr. May be reproduced and/or distributed but only intact, including this notice. Brief excerpts may be reproduced with credit, January 19 2012*
- *Keith Jack, Video Demystified: A handbook for Digital Engineer, Fifth Edition, Newnes publicaiton*
- *Axis Communication, "H.264 compression standard, New possibilities within video surveillance ".*
- *Alexander Hermans, "H.264/MPEG-4 advanced video coding", Matriculation number: 284141, September 11, 2012.*
- *S .Wenger, M.M. Hannuksela, T. Stockhammer, M. Westerlund and D.Singer, "RTP payload format for H.264 video", RFC 3984, February 2005*
- *Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, Operating Systems: Three easy pieces, Arpaci-Dusseau Books , March 2015 (version 0.90)*
- *http://lists.live555.com/pipermail/live-devel/2008-April/008417.html*
- *http://www.fourcc.org/*
- *https://trac.ffmpeg.org/wiki/Encode/H.264*

About the Author

Rohit Philip Mathew is an Embedded Software Engineer within VVDN Technologies. He worked with Cognizant Business Consulting prior to joining VVDN Technologies. Being an Embedded Enthusiast, He is focused on learning the current trends surrounding EE-CS Domain. Rohit has a Bachelors Degree in Applied Electronics and Instrumentation Engineering from Rajagiri School of Engineering and Technology, Kerala.