UDAB-ViS: User Driven Adaptable Bandwidth Video System

ABSTRACT

Adaptive bitrate (ABR) streaming has become an important and prevalent feature in many multimedia delivery systems, with content providers such as Netflix and Amazon using ABR streaming to increase bandwidth efficiency and provide the maximum user experience when channel conditions are not ideal. Where such systems could see improvement is in the delivery of live video and encoding parameter selection based on user preferences. In addition, such a scheme could have applications in the medical arena with remote patient to doctor consultations in which a patient (user) can choose encoding options differently depending on the content they are displaying for their doctor. For example, a patient may choose relatively low spatial resolution and high temporal resolution when simply talking to their doctor, then switch to high spatial resolution and low temporal resolution when showing the doctor an area of interest such as a burn or lesion. In this research, we present a camera system which provides spatially and temporally adaptive video streams, learning the user’s preferences in order to make intelligent scaling decisions. The system employs a mobile phone (Android) application serving as a live video streaming server with a hardware based H.264/AVC encoder for video compression. The encoding parameters can be configured by the user to change video resolution, frame rate, and bitrate, and the system can adapt these parameters when the bandwidth of the channel changes. A video client written for the desktop learns the user’s preferences (i.e. video size over frame rate) and intelligently selects encoding parameters when the channel is affected. It has been demonstrated that we have the ability to control video bandwidth by altering the spatial and temporal resolution, as well as the ability to make scaling decisions for the user when a sufficient amount to data has been collected.

I. INTRODUCTION

As wireless networks become more ubiquitous and the number of devices capable of accessing these networks increases, the need for more efficient video streaming becomes vastly more important. By 2015, it is expected that approximately 90 percent of online consumer traffic will be video and almost 66 percent of mobile traffic will be video [10]. With such a great amount of traffic, bandwidth efficiency becomes a serious concern in order to deliver the greatest quality of experience (QoE) to each individual user. At the same time, servers should be able to deliver video in such a way that information the user deems important is not lost due to bandwidth constraints and the method by which the video adapts.

To date, numerous solutions have been developed that take a range of approaches to solve this problem. Many of these approaches are based on already existing protocols such as the Real-Time Streaming Protocol RTSP [12], TCP, and HTTP [10][15-17] and achieve bandwidth adaptability in one of a few ways. These bandwidth adaptability techniques include progressive download streaming, adaptive bitrate (ABR) streaming, and stream-switching [13]. In progressive download streaming, video is transmitted as regular data files using TCP and is buffered by the client, playing when a sufficient amount of buffering has been achieved. This technique is employed by the popular video streaming website YouTube. Many solutions such as Netflix and Amazon Video employ ABR, in which the server can select the encoding bitrate in order to maximize the quality for a given channel. The result is a drastic reduction in the need for buffering in online video consumption; from the end user's perspective, the quality resolution of the video will change as network conditions change. Finally, with stream switching, the server encodes the source video with different encoding parameters and allows the client to switch between streams based on network conditions. Examples of stream switching solutions can be found in [13],[15] and [16]. Such techniques are ideal for online video streaming as they can use HTTP to negotiate streaming parameters and transmit the video stream; however, the major pitfall is that in most cases, only the video bitrate will be affected and no control is exercised over the spatial and temporal resolution of the video. In the case where spatial and temporal resolution can be affected, raw video will have to be re-encoded or transcoded at the source which can cause a delay in the video being transmitted [13].

In addition to protocol based approaches which tend to only allow for bitrate scalability, many codec based approached have been developed which allow for easy spatial and temporal scalability [7]. Two prime examples of codec based approaches are H.264/AVC and H.264/SVC [4][8]. With these codecs, video need only be encoded once and, due to the nature of the decoders, can be transmitted as several sub-streams in order to control the temporal or spatial resolution. The primary issue with this approach is that it limits the client to only a certain set of video decoders. A review of the H.264/SVC approach to scalability will be presented later in this paper.

A problem with many these solutions is that only quality resolution is affected in order to control the video bandwidth, which may or may not be acceptable depending on the user or the application. Numerous popularly used streaming solutions such as HTTP Adaptive Streaming (HAS) [10] and Dynamic Adaptive Streaming over HTTP (DASH) [14], will throttle the bitrate of a video stream in order to adapt to changes in channel bandwidth, which will result in a change in video quality. This behavior is exhibited by the widely used video content provider Netflix, which employs DASH to achieve bandwidth adaptability [17]. A prime example where this system will not be viable is in medical teleconferencing. A patient may wish to have a remote consultation with his or her doctor in order to quickly and efficiently receive feedback about a medical problem such as a burn or lesion they may have endured. Were the video bandwidth to be controlled by ABR and the quality resolution be affected in order to fit to the channel, the doctor may not be able to properly diagnose or provide valid feedback to the user because the visual quality is not sufficient. However, if the system were to scale the temporal resolution of the video instead of the quality, visual information would be kept intact and the less important temporal information would be sacrificed. The system we propose uses this kind of scalability in tandem with the development of user profiles in order to make an intelligent determination of how to alter each resolution. In this, video context is able to be taken into consideration to provide different results in different situations. This addresses the issue of how to control the different encoding parameters in such a way as to align with the user's desires without requiring the user to change the parameters themself.

One of the key factors to achieve such a system is the automation of video bandwidth control. In [13], the authors propose a *Quality Adaptation Controller* which uses a proportional integrator (PI) controller at the server to select an appropriate video stream for the client. The system uses stream switching and employs feedback control to maintain the video bandwidth below the available channel bandwidth. Their system succeeds in being a codec independent solution that is able to keep the video level at or below the available bandwidth with a transient time of less than 30 seconds, and is able to share the channel with concurrent streams [13]. However, the solution does not take into account the context of the video and the user's desires in order to select encoding parameters in an intelligent fashion. In this paper, we propose a learning module that uses two support vector machines (SVM) to learn the user's preferences and be able to adapt the video appropriately. These preferences are learned based on contextual video features such as channel bandwidth and video content type, thus allowing for different functionality for different situations.

The rest of the paper is structured as follows; in section II) we review basic H.264 packetization techniques; in section III), we review H.264/SVC as a scalable streaming solution, as well as the scalability solution used in our system; sections IV) and V) will describe our system architecture, as well as the method by which we encode video on our server; section VI) will detail our learning model and how video bandwidth is optimized; and finally, section VII) will go over our experiments and experimental results.

II. PACKETIZATION AND FRAME ENCODING FOR H.264

a) H.264/AVC Basics

H.264/AVC is a video coding standard developed by the ITU-T Video Coding Experts Group (VCEG) and ISO/IEC Moving Pictures Expert Group (MPEG) designed with the goals of enhanced video compression and "network friendly" video representation addressing "conversational" applications such as video conferencing, as well as "non-conversational" applications such as broadcast streaming [1]. In December of 2001 VCEG and MPEG formed a Joint Video Team (JVT) which in March of 2003 finalized the draft of the H.264/AVC video coding standard for formal submission [1]. The standard provides highly efficient video coding and can be used in a breadth of applications, from storage to streaming. It provides bitrate savings of 50% or more over its predecessor video codecs [2]. This makes H.264/AVC especially applicable in wireless environments. In [9] and [11], the authors review and show the effectiveness of H.264/AVC as a tool for video compression over IP and in wireless settings. In addition, H.264 employs a litany of features to enhance the quality and customization of video coding. Some examples of these features are:

- Motion vectors over picture boundaries
- Multiple reference picture motion compensation: P-frames and B-frames can select from a large number of stored reference pictures to determine the values in the current frame [1].
- Weighted prediction

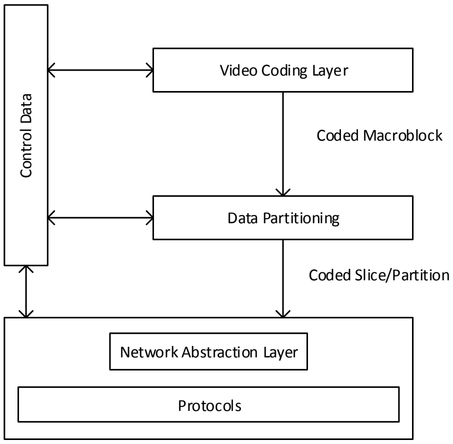

Figure 1 – H.264/AVC Encoder Structure [1]

In addition, H.264/AVC supports a number of features to allow greater error resilience and flexibility over many environments [1]. Some of these important features are:

- NAL unit syntax structure
- Flexible slice size
- Flexible Macroblock Ordering (FMO)
- Arbitrary Slice Ordering

[1] and [3] provide a more detailed overview of each of these features and more. The structure of an H.264/AVC encoder is depicted in Figure 1. In this, the codec covers both a Video Coding Layer (VCL) as well as a Network Abstraction Layer (NAL). The VCL performs the physical encoding and compression of video while the NAL provides a header to packeted data and is exceptionally important to assist the decoder in understanding how to handle the packetized frames. We will now discuss the NAL and give a general overview of how frames are packetized when sent over UDP/RTP.

B) NAL Unit Headers and Packetization

There are a few key concepts associated with the NAL in H.264 video encoding, "including the NAL unit, byte streams, and packet format uses of NAL units, parameter sets, and access units" [1]. The NAL allows the ability to map and packetize data to a multitude of transport layer protocols (i.e. UDP/RTP, file formats, etc.). The most important aspect that we will examine is the NAL unit and its relation to different types of transport layer payloads. In particular, we will look at the NAL unit when streaming video over UDP/RTP. When frames are encoded in the VCL they are organized into NAL units which serve as a wrapper to the data.

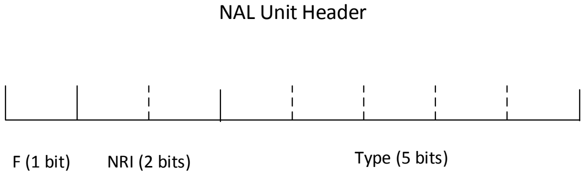

Figure 2 – NAL Unit Header Structure

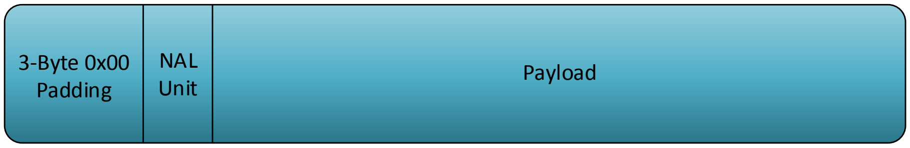

Figure 3 – H.264 Packet Format

Each NAL unit will contain a header byte that will indicate what type of data is contained in this unit. When streaming video using UDP/RTP in the transport layer, the beginning of each packet will contain a NAL unit. The NAL unit conveys relevant information for the decoder about the encoded data. It contains a one byte header and a payload byte string [2]. The header will indicate the type of NAL unit, potential presence of errors, and information about the relative importance of this NAL unit in the decoding process [2]. The structure of the one-byte NAL unit header is shown in Figure 2.

The fields of the header are designated as follows:

- F: forbidden bit; should always be 0
- NRI: used to indicate if the content of this NAL unit should be used to reconstruct reference pictures in inter picture prediction
- Type: Specifies the NAL unit payload type

Important types of NAL units are different parameter sets that contain relevant information about a given video stream or frame. These can be broken down into sequence parameter sets and picture parameter sets. The sequence parameter set can be transmitted well in advance of the actual video stream and will allow robust protection against the loss of information that change infrequently during a specific session. Picture parameter sets will contain relevant information that will remain unchanged for a particular coded picture. Following the NAL unit will be the VCL coded frames which the decoder will handle based on the type of NAL unit. The H.264/AVC specification also defines a set of profiles and levels which specify different sets of required functional support for decoders. They are designed to support a high degree of interoperability between various different applications of the standard. According to [1], “A profile defines a set of coding tools or algorithms that can be used in generating a conforming bit-stream, whereas a level places constraints on certain key parameters of the bitstream.” There exist a number of profiles which provide various features and varying amounts of flexibility and customization points. Some examples of profiles and their applications are:

- Constrained Baseline Profile: low cost applications such as video conferencing in mobile
- Baseline Profile: used in video conferencing
- Main Profile: used in various mainstream broadcast and storage applications
- Extended Profile: intended to be used as a streaming profile
- High Profile: broadcast and disk storage applications (HDTV, Blu-ray, etc.)

For video conferencing applications or streaming from mobile devices such as in our proposed system, the constrained baseline or baseline profile are appropriate choices. More detailed information on profile types and their constraints can be found in section A.2 of [3].

In our system, we are encoding frames using the constrained baseline profile. In this, we have the most basic set of features and can only encode I and P frames (there is no B frame support for this profile) [3]. We can then determine the type of data being transmitted based on the NAL unit header. Figure 3 shows how each packet is formatted in our system. Every packet is padded with three bytes of 0x00. This is immediately followed by the NAL unit for that frame. Finally, we have the encoded data. In the NAL unit, the very first byte will be the NAL unit header. For our packets, we can have one of three types of NAL units based on the NAL unit header:

- 0x67: This packet contains a sequence parameter set for the next segment of video
- 0x61: This packet contains the next I-Frame
- 0x41: This packet contains a P-Frame

We can use this to determine when a new segment of video is started and weigh the relative importance of each incoming packet. In the next section, we will discuss the scalable video coding extension of H.264 and compare it with the method of encoding parameter adaptation that we are using to achieve video bandwidth control.

III. Scalable Video Coding vs. Single Layer Video Streams

A major concern in video streaming systems is that of uncertain channel conditions affecting the quality of the video stream. For example, when transmitting a stream at a certain quality having bandwidth B over a congested channel where the bandwidth fluctuates to less than B, the receiving terminal will experience significant degradation of video quality. In order to combat this, scalable codecs have been developed which can alter a certain resolution of the video to fit the channel. Such an alteration can be to the temporal or spatial resolution of the video or the quality. In H.264/SVC, this is accomplished by removing certain parts of the bit stream in such a way that the underlying streams still represent a valid bit stream for the decoder [4]. For example, a transmitter may send one base layer bit stream and multiple enhancement layer bit streams, and the receiver may select which of these bit streams to receive and send to the decoder. In this, the bandwidth of the video can be controlled by choosing only the necessary bit streams to stay within the channel bandwidth. Contrast this with a single layer video stream in which a single bit stream is transmitted which represents a valid stream for a given decoder. In order to have control over the bandwidth of the video, the transmitter must encode the source video with different encoding parameters and the receiving decoder must be able to adapt to these changes. We will compare these two methods of adapting a video stream to varying channel conditions; first, the Scalable Video Coding extension of

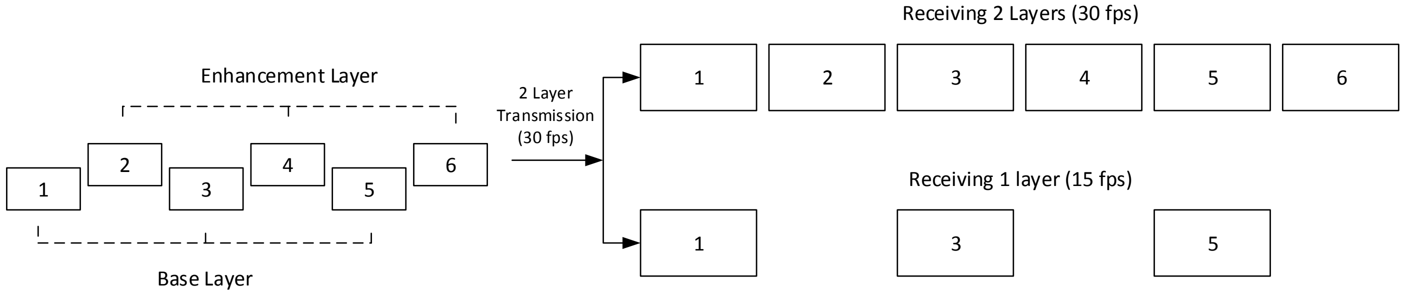
Receiving 2 Layers (30 fps)
Enhancement Layer
2 Layer Transmission (30 fps)
1
2
3
4
5
6
Receiving 1 layer (15 fps)
Base Layer

Figure 4 -- Temporal Scalability with H.264/SVC [4]

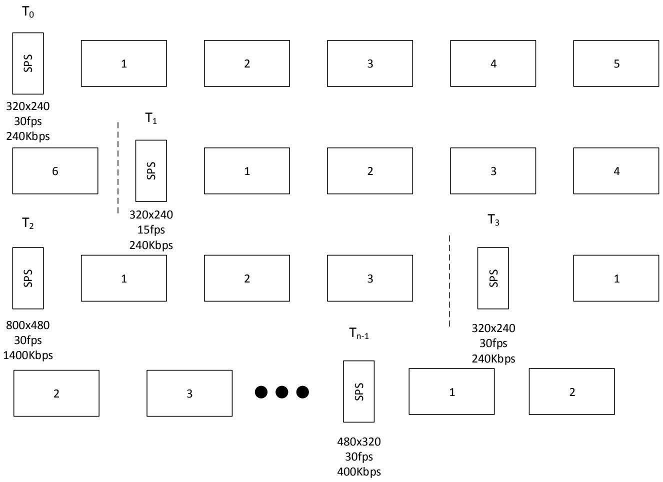

Figure 5 – Single Layer Encoding Parameter Switching

H.264 as described in [4], and second, using a single layer video stream and altering encoding parameters at the source.

The Scalable Video Coding extension of H.264/AVC inherits all of the base functionality of H.264 with only the necessary added features to achieve scalable video streaming. In this, it supports the primary scalable parameters, being temporal, spatial, and quality resolution. To achieve temporal scalability, the transmitter may send multiple temporal streams divided into a temporal base layer and one or more temporal enhancement layers [4]. One may label these streams as T_0 through T_k. A receiving decoder then simply needs to know which of these access units are valid or invalid for the current stream, starting from 0 through n where $n \leq k$. The ability to partition a stream as such and play only the valid streams is already present in the H.264/AVC standard with the employment of reference picture memory control [4]. The partitioning of a stream into multiple temporal streams is illustrated in figure 4. In order to achieve spatial scalability, SVC uses multi-layer coding with inter-layer prediction [4]. Multiple layers will be transmitted, each corresponding to a specific spatial resolution and referred to by an integer valued dependency identifier between 0 and *d-1* where *d* is the number of spatial layers [4]. Quality scalability works on the same principle as spatial scalability with the layers transmitted being of the same spatial resolution.

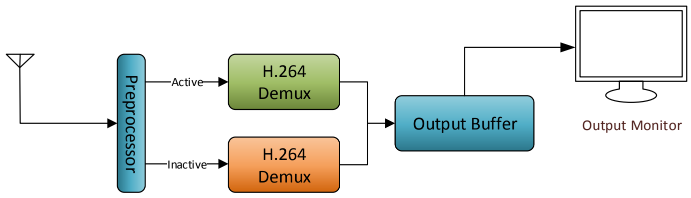

Figure 6 – Receiver Architecture

Another method to control the bandwidth of a video stream is to use single-layer coding and manipulate the encoding parameters at the source. In a single-layer coded video stream, one bit stream is encoded and sent to the receiver. This bit stream is of a fixed spatial, temporal, and quality resolution for the entire sequence of video. There is no built in mechanism to change any of these resolutions midstream. In this, one can control the bandwidth of the video by using different encoding parameters at the source for different segments of video. We can partition a video stream into multiple segments (sequences) labeled $T_k, k = 0 \dots n-1$ where *n* is the number of segments for the given session and T_k is the time instance when the segment *k* begins. Such a partitioning may be know in advance or can be determined as a function of the channel bandwidth if channel bandwidth can be measured within some reasonable degree of accuracy (for example, using a method like DIChirp as laid out in [5]). At each time instance T_k the transmitter may reset the encoding parameters in such a way that the bandwidth of the video is altered to fit to the channel. When this occurs, a new sequence parameter set may be introduced into the stream to signal to the decoder the changes to the encoded video. A sample stream could have the form as depicted in figure 5. The changes to the encoding parameters will insert a delay into the stream for the time it takes to restart the encoding process. To compensate for this and to handle the alteration events at times T_k, figure 6 shows our proposed receiver architecture.

We have a preprocessor which can inspect the NAL unit headers of each incoming packet and determine when the video stream has changed; for our system, this is when the header is 0x67. At this event, the idle decoder thread will be woken up and set up to decode the next sequence of video. The previously active decoder thread will empty its queue prior to the start of the new segment of video. This thread can then signal completion and the new thread will take over.

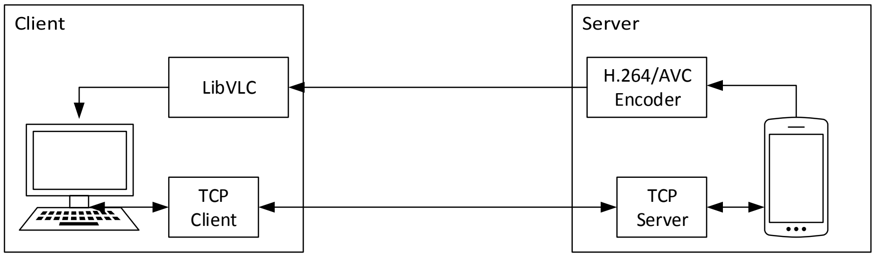

Figure 7 – System Architecture

In our system, we decided to use a single layer video stream with the proposed architecture in figure 6 to achieve video bandwidth control. We are able to accomplish this by using hardware based video codecs; in particular, a hardware H.264/AVC encoder. As a result, we are able to initialize the encoder in real time and there is no CPU overhead when it comes to encoding frames (vs. a software encoder). In addition, this real-time efficiency allows us to achieve rapid encoding parameter switches resulting in minimal delay between video segments. We can control the temporal, spatial, and quality resolution of each video segment, effectively allowing us to control the bandwidth of the video with a range of effects of the output. Next, we will present the underlying architecture of our system which allows us to send and receive the types of we have just discussed.

IV. System Architecture

A basic outline of our system architecture is depicted in figure 7, with the streaming clients, streaming server, and constituent components. The camera system that we have developed consists of a video client and a streaming server. The client connects to the server to request a new streaming session, displaying the video in a media player based on the VideoLAN VLC player. The streaming server is an Android application designed to be run on Qualcomm MSM8960 hardware. Video is encoded on the device using a hardware H.264/AVC encoder and streamed to the client in raw UDP packets. We decided to deliver video without a higher level wrapper protocol (such as the Real-time Transfer Protocol (RTP) [2] and the Real-Time Streaming Protocol (RTSP) [12]) because we are using our own session management which is discussed in section V.C. The server is currently designed to send a unicast stream to the client connected to it, and a client can view any number of video streams from different servers. The following sections will describe how the system works in detail.

a) Client Design

The client is designed as a desktop application consisting primarily of a control center and one or more video windows. The video windows contain a media player for displaying the video, as well as all of the controls necessary for the user to manipulate any of the encoding parameters of the stream.

The media player uses LibVLC, a library used in the popular VLC media player developed by the VideoLAN group. In this, we encapsulate the main functionality needed for our design; namely, configuring the media player and playing a certain network stream. We have configured the media player to be consistent with the architecture described in section II.B, in that we inform the back end to set up two decoder threads that can be used to demux a video stream. This allows us to dynamically change the encoding parameters midstream without seeing a significant effect on video playback. We can then control the bandwidth of the video while still maintaining an optimal user experience. In order to play the video itself, we point the media player to the following MRL:

udp/h264://@:[port_num]

This MRL specifies a raw UDP stream that should be demuxed using H.264/AVC. The value of "port_num" should be the port that this video is being streamed to.

On top of the media player is a TCP client that will interact with the server. When a new video stream is requested, the client will attempt to connect to the server, and upon successful connection, will start the media player. This client will then send all requests to the server and react appropriately to responses.

The client is responsible for determining the correct choice of encoding parameters and for managing the bandwidth of the video based on the bandwidth of channel. In this, the client is equipped with the necessary tools to determine what the current channel bandwidth is and respond to changes appropriately. These decisions should be based on the user's preferences (if an accurate model of the user has been developed) or resort to a default decision function (when developing a user profile). The client should also be smart enough not to interfere with the user when they make their own decisions about what encoding parameters they wish. The method by which we develop user profiles and utilize them will be discussed in section VI.

b) Server Design

Our camera server application runs on a DragonBoard APQ8060A development board utilizing a Qualcomm APQ8060A processor. The application captures live video from one front-facing 8MP camera, at varying spatial and temporal resolutions. When the application launches, we set these to a default of 320x240 at 30 frames per second (fps). The selection of video bitrate is a function of channel bandwidth and these parameters, and will be discussed in section VI. Running as a daemon in the application is a TCP server which will handle any incoming connections from clients and service any requests. On each connection, a thread is forked that acts as an interface between the client and server. A handle to the encoder is given to each of these threads to allow them control over the resolutions of the video streams. The handle is encapsulated in an object we call the encoder activation interface. This object, as the name implies, acts as an interface to the encoder (as well as the camera). The software stack between the interface and the encoder will be described in the next section. Via this interface a consumer may initialize, destroy, and alter an encoder for a certain video stream. This allows the client full control over the parameters of the video stream and enables it to control the bandwidth of the video. Essentially, the server should remain agnostic of channel conditions and act as a slave to the connected clients, reconfiguring the stream as necessary when a client requests it. The reasoning behind this is that the client application will learn the user's preferences and can make more intelligent decisions about the encoding parameters than the server.

c) Session Management

The system consists of a TCP layer for communication between the client and server which acts as a session manager. TCP is used for reliable communication of messages between terminals as well as to signal the beginning and end of a streaming session. A streaming session begins once the server accepts a client's connection, and ends when one of the terminals has disconnected. When the client wishes to receive a particular stream from a particular server, it will first attempt to make a connection with that server via its TCP client. Upon connection, the server will start a new thread to begin servicing the client's requests. The server thread will first start the encoder, which will begin streaming packets containing the encoded H.264 frames. This thread then enters a loop in which it will respond to the client's requests until it is detected that the client has disconnected. We have defined a very simple protocol for submitting such requests in which the client will either request to alter encoding parameters or stop the video stream. The request to update encoding parameters will also serve to start a stream again if it has been previously stopped. To update the encoder, the client will send the following message:

start <frame_width> <frame_height> <frame_rate> <bitrate>

where "frame_width" is the new desired width, "frame_height" is the new desired height, "frame_rate" is the new desired frame rate, and "bitrate" is the new desired video bitrate. To stop the encoder, the request is as follows:

stop

Upon disconnecting, the thread processing the client's requests will shut down the encoder, stop the video stream, and exit. When a camera update is successful, the server will send a response indicating success. The client will also be notified of any malformations in the request string.

V. Snapdragon Video Framework

Video is encoded on the board by a hardware H.264/AVC encoder. The DragonBoard APQ8060A is equipped with numerous hardware based codecs for both audio and video. We decided to use the hardware encoder in order to encode frames in real time. In addition, the extra speed we acquire greatly assists in our video scaling method. We will now describe how the camera server works to capture video frames, encode them, and send them as raw UDP packets.

To use the hardware components, Android provides a wrapper to the OpenMAX Integration Layer called IOMX which can interact with and utilize hardware media codecs. The OpenMAX Integration Layer is a component based API designed to provide a layer of abstraction on top of multimedia hardware and software architecture. It is also designed to give media components portability across a range of devices. With the introduction of the stagefright media framework, Google added the OpenMAX IL functionality to the Android operating system, allowing OEMs the ability to provide software hooks that serve as an interface to the hardware for developers.

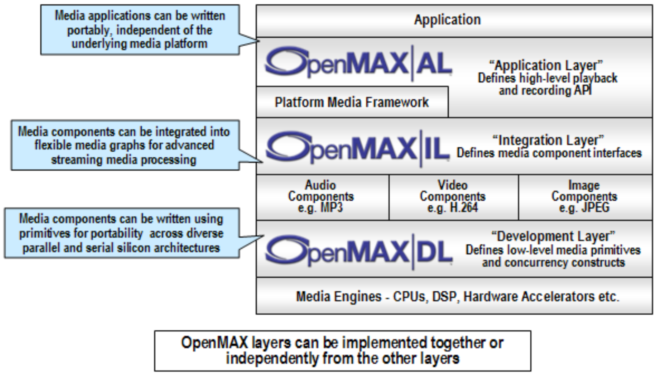
Media applications can be written portably, independent of the underlying media platform
Media components can be integrated into flexible media graphs for advanced streaming media processing
Media components can be written using primitives for portability across diverse parallel and serial silicon architectures
Application
OpenMAX|AL
"Application Layer" Defines high-level playback and recording API
Platform Media Framework
OpenMAX|IL
"Integration Layer" Defines media component interfaces
Audio Components e.g. MP3
Video Components e.g. H.264
Image Components e.g. JPEG
OpenMAX|DL
"Development Layer" Defines low-level media primitives and concurrency constructs
Media Engines - CPUs, DSP, Hardware Accelerators etc.
OpenMAX layers can be implemented together or independently from the other layers

Figure 8 – OpenMAX Layers

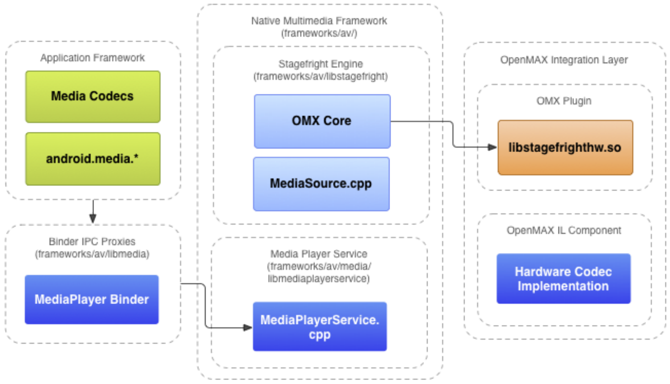
Application Framework
Media Codecs
android.media.*
Binder IPC Proxies (frameworks/av/libmedia)
MediaPlayer Binder
Native Multimedia Framework (frameworks/av/)
Stagefright Engine (frameworks/av/libstagefright)
OMX Core
MediaSource.cpp
Media Player Service (frameworks/av/media/libmediaplayerservice)
MediaPlayerService.cpp
OpenMAX Integration Layer
OMX Plugin
libstagefrighthw.so
OpenMAX IL Component
Hardware Codec Implementation

Figure 9 – Stagefright Media Framework

Qualcomm has developed and provided to users a set of sample code that uses IOMX to encode and decode various audio and video formats. Our server is leveraging this code to use the hardware H.264/AVC encoder present on the device. Qualcomm's implementation can be broken down into a few different levels, as depicted in Figure 10. The lowest levels are the hardware, OpenMAX IL, and IOMX, which we have already discussed; they are not a part of the provided code. Qualcomm has developed a few classes written in C++ which wrap around IOMX, providing an interface for higher level code. Instances of these classes can be used to query available codecs, activate and initialize an encoder/decoder, encode/decode frames, and tear down encoders/decoders. In order to use this interface in an Android application, one must compile the classes into a shared library against the Android source code, and add this shared library to the "libs" directory of the project. The next level is a “public interface” between the shared library and the Java Native Interface (JNI) written in C. Essentially, it contains static C functions which call the various member functions of the encoder/decoder classes. Next is the JNI layer which contains various JNI style C functions that will manage an encoder/decoder object and use the public interface to call any of the specific functions needed during the encoding process. Finally, there is the Android application; this is the camera server that we have defined, with the encoder specific functionality defined in our encoder activation interface class. Here, we simply declare the native functions and use them during the encoding process.

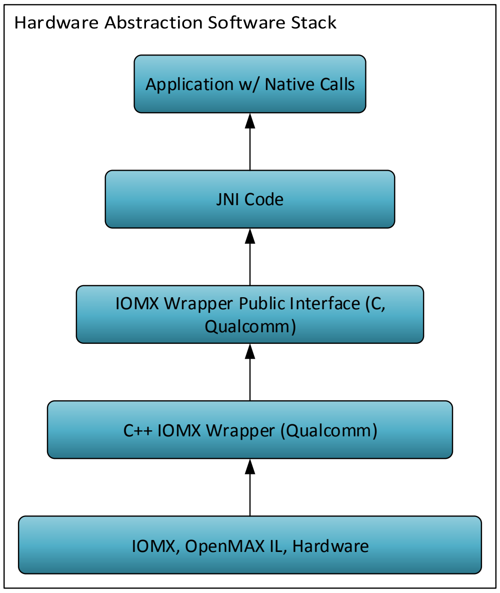
Hardware Abstraction Software Stack
Application w/ Native Calls
JNI Code
IOMX Wrapper Public Interface (C, Qualcomm)
C++ IOMX Wrapper (Qualcomm)
IOMX, OpenMAX IL, Hardware

Figure 10 – Encoder Hardware Abstraction

The first step to encode and stream video is to set the encoding parameters and initialize the encoder. When a client connects to the server, the running thread uses the encoder activation interface to start the streaming session. A few native functions are defined in this class, namely, to set the streaming recipient, start the encoder, stop the encoder, encode a frame, update the bitrate, and close the socket we are streaming video through. These native functions are implemented in a C source file and are called through the JNI. From the encoder activation interface, we first set the recipient client's IP address (which is a global flag in the JNI code's address space). Next, we run another thread to start and initialize the encoder, passing the frame width, frame height, frame rate, and bitrate as parameters. This function will set up a UDP socket and initialize an encoder object. In addition, we register a callback function with this object which will receive the encoded frames and subsequently stream them using the socket we created. Back in the Java code, a handler object will be sent a message from the initialization thread upon completion, indicating success or failure; if successful, we can start sending frames to the encoder.

When the initialization process completes successfully, a flag is set indicating that buffers containing frame data should be sent to the encoder. This flag is checked in the callback function that is invoked on each new frame; when it is set to true, the frames will be sent to be encoded at the rate specified by the frame rate. The format of these buffers is NV21. Currently we are limiting the frame rate to be either 15 or 30 frames per second in order to get the best performance from the encoder. In addition, this greatly simplifies our bandwidth optimization technique which we will discuss in the next section. Next, piping the frames to the encoder is accomplished in a JNI function. This native function will directly send the frame data to the encoder through the software stack, freeing the memory pointed to by the buffers. Once the encoding process is complete, our native callback function will be invoked with the encoded data passed as a parameter. This callback simply sends the data in its buffer to the recipient in a raw UDP packet. This process of encoding frames and sending them to the client continues until the server sees that the client has disconnected. When this occurs, a thread is run to shut down the encoder and close the socket over which UDP packets are being sent. Finally, the flag to encode frames is lowered. For the code samples involved in this process please see Appendix (###). In the next section, we will look at how video bandwidth is optimized in an intelligent way to reflect the user’s desires.

VI. USER PROFILES AND BANDWIDTH OPTIMIZATION

In the following sections we will describe how user profiles are developed and how they are used in order to optimize the bandwidth and make scaling decisions. First we will discuss preferences and profiles; next we will describe how we are using machine learning to develop the profiles; following that, we will describe how the video bandwidth is determined, and how we make intelligent scaling decisions.

a) Preferences and Profiles

We develop a profile of the user which will determine how video will be scaled when it is no longer optimized to the transmission channel. In this, we have designated the user to be one of four discrete classes which represent their preferences in relation to video quality:

- Class 0: User prefers high temporal and spatial resolution, no quality preference.
- Class 1: User prefers high temporal resolution; optimize spatial resolution for quality.
- Class 2: User prefers high spatial resolution; optimize temporal resolution for quality.
- Class 3: User prefers optimal quality; configure temporal and spatial resolution appropriately.

By knowing these preferences we are enabled to make decisions about the different video resolutions in order to optimize the video bandwidth. Transforming these classes to their equivalent binary form we obtain a 2 bit value in which the first bit will convey the temporal resolution preference and the second bit will convey the spatial resolution preference. A set bit indicates the desire for a high value given to this resolution at the expense of quality, and an unset bit indicates the desire to optimize quality resolution at the expense of the resolution in question. In this, we can find the best way to alter the spatial and temporal resolution such that the video bandwidth will fit the channel and align with the user's desires in terms of quality. We will next review the creation of user profiles, which is done using machine learning.

b) Machine Learning

In order to create a profile for each user we are employing a machine learning algorithm to create a decision function based on the user's behavior that will be able to predict the class a user falls into. The decision function is being calculated using the LibSVM implementation of a C support vector machine with a radial basis kernel [6]. We will now describe how each aspect of the learning algorithm works to create a profile for the user.

In supervised machine learning, a training set composed of a series of training samples is presented as input to a certain learning algorithm, the output of which will be a set of coefficients for a decision function. For classification problems such as ours, the training sample will be the values of a set of contextual features which we find relevant to the output, and a label for each of these vectors which denotes what class applies at the instant of the sample. The features that we use are the channel bandwidth and the content type of the video (i.e. personal server, peer to peer, medical video, etc.). For the two preferences being learned we are using two C support vector machines with radial basis kernels. Other methods to solve multiclass problems are using a one-vs.-all algorithm or a one-vs.-one algorithm, but we decided to keep the system simple and leave the two preferences as independent learning problems.

When a new user begins interacting with the system, they are initiated in "learning mode." In learning mode, a change in channel bandwidth will result in a knee-jerk reaction by the system to simply alter the quality resolution of the video, leaving spatial and temporal resolution unchanged. Next, we obtain explicit feedback from the user by asking if the video quality is acceptable. If they answer yes, we sample relevant features and determine which classes the user falls into, adding this sample to the training set. If the user answers no, we do not take action. When the user makes a decision to change the video resolutions in some way, we again issue the prompt to get explicit feedback. This process continues until the user exits learning mode, which can only be done when the training set is sufficiently large to accurately train the support vector machines. Once these conditions are met, the SVMs are trained, and 3-fold cross validation is performed to ensure accurate selection of the C parameter for the SVM and gamma parameter for the kernel [6]. This will generate the decision function which we will be able to use to make predictions.

c) Video Bandwidth Determinations

The catalyst for video bandwidth recalculation is when the channel conditions have changed significantly enough to warrant scaling the video. There are numerous methods to estimate the bandwidth of a channel within a certain degree of accuracy, such as DIChirp [5]. We will now lay out how changing the various video resolutions will affect the video bandwidth in our system.

In order to alter the quality resolution we can change the compression bitrate of the video. We determine two possible rates which can be used as the final bitrate of the video: a maximum rate and an optimal rate. The maximum rate can simply be calculated as a function of the known channel bandwidth and an optimization constant:

$$Bitrate_{max} = Bandwidth_{channel} * K$$

where $Bitrate_{max}$ is the maximum allowable bitrate, $Bandwidth_{channel}$ is the bandwidth of the channel, and K is the optimization constant determining the percentage of available bandwidth that is acceptable to fill.[1] The optimal bitrate is found as a function of the spatial resolution and an optimization constant:

$$Bitrate_{opt} = width * height * Q_{max}$$

where $Bitrate_{opt}$ is the optimum bitrate for the given spatial resolution, $width$ and $height$ are the dimensions of the video, and Q_{max} is the optimization constant which can be configured to find the highest necessary bitrate to deliver high quality video at a given spatial resolution. The best choice of bitrate is either the maximum allowable or maximum necessary bitrate for the given channel, spatial resolution, and temporal resolution, which is determined as the minimum of these two bitrates:

$$Bitrate_{out} = min\left(Bitrate_{max}, Bitrate_{opt}\right)$$

This will give us the greatest achievable and necessary bitrate for delivering the best possible quality to the client.

Determining the temporal and spatial resolution will affect the quality resolution such that higher values will result in lower quality once the bitrate has reached $Bitrate_{max}$. We will inspect the user's class in order to set these two parameters, as the class will tell us if the temporal/spatial resolution should be optimized or if the quality should be optimized. For temporal resolution, we are limiting the possible values to 30 and 15. In the case where the temporal resolution bit is set in the user's class, we simply set the frame rate to 30fps, and if unset, change the frame rate to 15 fps. If the spatial resolution bit is set, the spatial resolution is changed to the maximum value. When the bit is unset, we first calculate $Bitrate_{max}$, then reduce the spatial resolution until

$$Bitrate_{opt} \leq Bitrate_{max}$$

This will result in a spatial resolution that will be small enough to provide optimal quality video.

d) Scaling Decisions

They way that video bandwidth will be optimized depends on the class that the user falls into for the given feature set at the time T_k (the moment when the encoding parameters of the k^{th} segment of video are selected). When channel bandwidth changes significantly enough, the first action we take is to predict the user's preferences using the two decision functions we have already generated (as described in section VII.b.). This provides us with a bitmask that serves as the user's class; the primary purpose of this class is to specify the order in which to scale the encoding parameters. This ordering will also depend on whether or not the video bandwidth is increasing or decreasing. The chart in Figure 11 depicts the possible decisions that can be made about the video bandwidth.

[1] Note: due to limitations in our encoder, the max bitrate must be calculated as a function of the frame rate. We are using the following calculation:

$$Bitrate_{max} = \left(Bitrate_{in} * \left(\frac{Bandwidth_{channel}}{Bitrate_{in} * \left(\frac{T}{15}\right)} \right) \right) * K$$

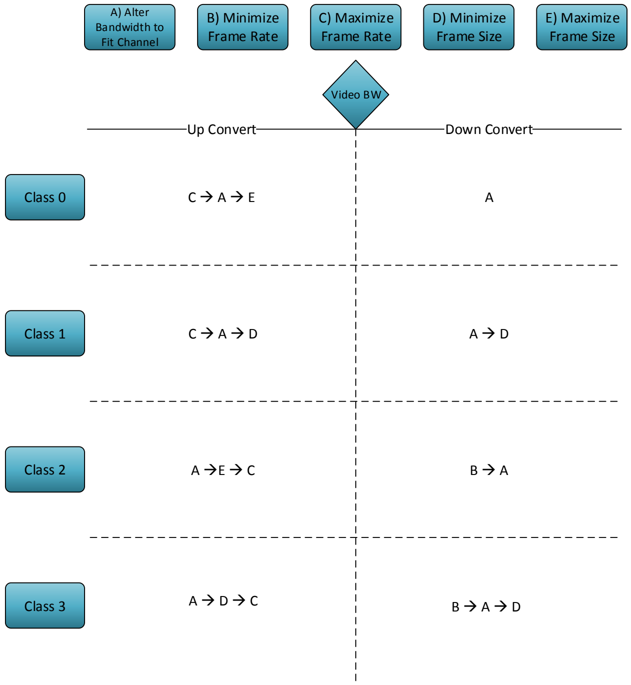

Figure 11 – Bandwidth Decision Module

We can then successfully adapt to channel bandwidth changes and alter the encoding parameters in such a way that the user's desires are fulfilled. In the next sections we will present our experimental set up and results of our experiments.

VII. Experimental Results

a) Test Bed

We are using the following test bed to test our system:

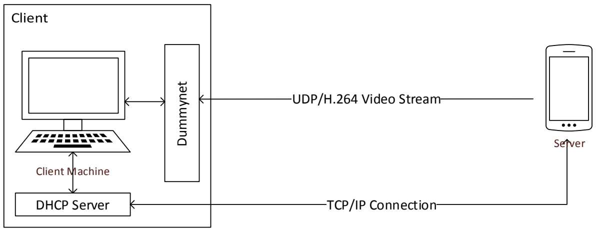

Figure 12 – Camera System Test Bed

We create point to point connection between the client and server by setting up the client machine as a DHCP server and connecting it directly to the Android device, allocating it an IP address on an arbitrary subnet. To simulate network conditions we are using dummynet, a widely used network emulator. With dummynet, one can control the traffic over a specific channel by limiting bandwidth, inserting packet losses, inserting delay, etc. In order to simulate bandwidth change detection the client simply reads from a file that contains channel bandwidth information. In our test set up we run a script which simultaneously sets the bandwidth of the channel to varying values at certain intervals using dummynet, and writes this bandwidth to a flat file that the client can read from. With this, we are able to implement some of the conditions of a real network and be aware of the bandwidth in the client application.

b) Results

Our first experiment tests the ability of the system to control the bandwidth of the video by altering the spatial resolution. Channel bandwidth is kept constant, as well as frame rate which is kept at 30 frames per second. The spatial resolution is changed from 800x480 to 480x320 to 320x240. We used Wireshark to capture and display the bandwidth of the video, obtaining the following results:

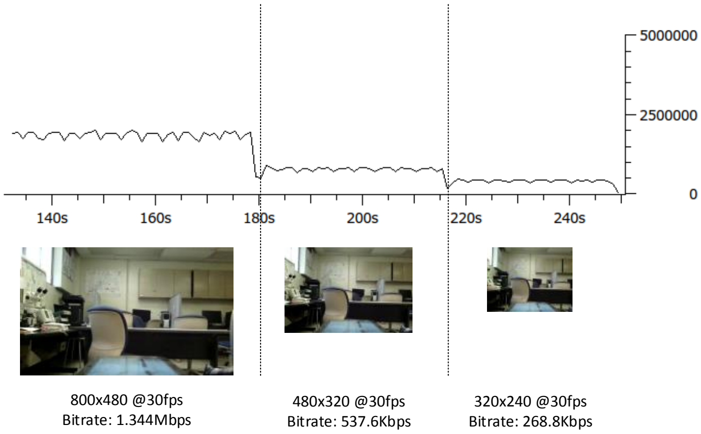

Figure 13 – Spatial Resolution Test

When the spatial resolution is changed, the bandwidth of the video is changed as well. One can easily deduce that this change is directly proportional to the change in resolution. For example, when the spatial resolution changes from 800x480 to 480x320, the total pixel ratio and the ratio of video bandwidths are equivalent:

$$\frac{480*320}{800*480}=\frac{76800}{384000}=0.4$$

$$\frac{537.6Kbps}{1344Kbps}=0.4$$

One can calculate this for the second change and receive similar results. In this, we have successfully exercised control over the video bandwidth by altering the spatial resolution of the video. In our next test, we showed that we can control video bitrate by altering the harshness of compression, resulting in a change in video quality. We tested the system's response to changes in channel bandwidth by altering the bandwidth from 1600Kbps to 800Kbps to 400Kbps. Spatial resolution was kept constant at 800x400 and temporal resolution was kept constant at 30 fps. The resulting changes in video bitrate, as well as playback quality, are depicted below:

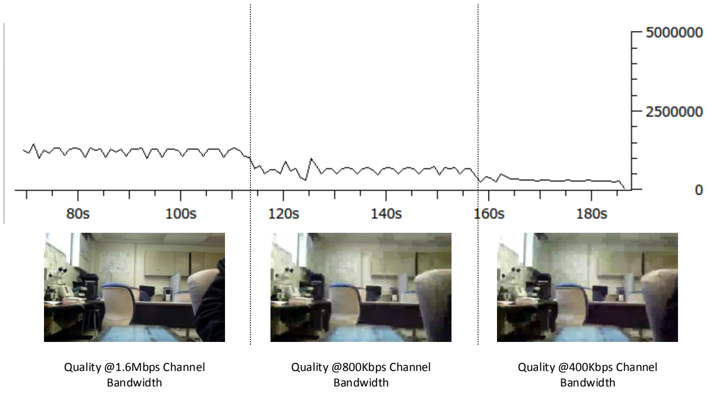

Figure 14 – Quality Resolution Results

The system successfully and immediately responds to changes in channel bandwidth by reducing the quality resolution of the video. In addition, the reduction in video bitrate is directly proportional to the reduction in available bandwidth. Finally, we tested the system's ability to determine the user's preferences and scale the video appropriately when channel bandwidth changes. We first provided the learning mechanism an arbitrary training set.

Channel Bandwidth (Kbps)	Content Type	Class
1200	0	0
1300	0	0
800	0	2
500	0	3
900	0	2
300	0	3
1100	0	0
1300	0	0
1500	0	0
850	0	2
500	0	3
1200	0	0
1600	0	0
1300	0	0
300	0	3
900	0	2
1400	0	0
1600	0	0
700	0	2
1200	0	0

Table 1 – Training Set

The "content type" feature was kept constant and is therefore negligible in this test (it will have no effect on the decision function created by the support vector machines). The primary determining factor in this test is channel bandwidth. This is made evident by looking at the class distribution of the training set:

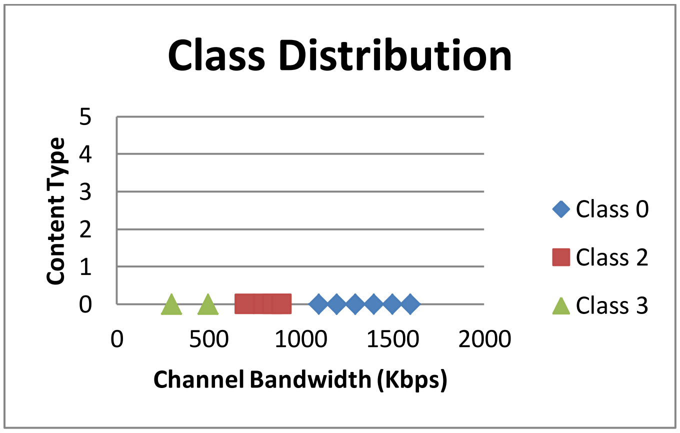

Figure 15 – Class Distribution

Analyzing the data, the expected class predictions based on channel bandwidth should be class 0 when bandwidth exceeds 1000Kbps, class 2 between 600Kbps and 1000Kbps, and class 3 when channel bandwidth is less than 600Kbps. This will translate to high size and high frame rate, high size and low frame rate, and high quality and low frame rate, respectively. We tested the system by changing channel bandwidth from 1200Kbps to 800Kbps to 400Kbps, yielding the following results:

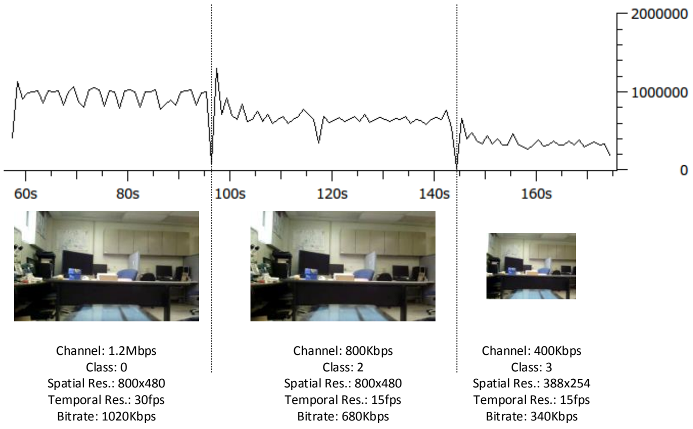

Figure 16 – Trained Decision Function Results

The system accurately predicted the classes at each instant of bandwidth change. This resulted in different decisions being made for each switch. If we label the start of the experiment as T_0 and the change to 1200Kbps as T_1, we see that at time T_1 the system determined that user's preferences most likely fall into class 0, thus changing the spatial resolution and frame rate to their optimal values. At time T_2, the channel bandwidth decreased to 800Kbps, and the learning algorithm placed the user into class 2; again this was the desired outcome. Video bandwidth was then reduced by lowering the frame rate and keeping the spatial resolution high, resulting in moderate quality resolution. Finally, at time T_3, the learning algorithm correctly placed the user into class 3 when the channel bandwidth fell to 400Kbps. This caused a reduction in spatial resolution to the point at which quality would be acceptable, as well as constant frame rate. In this, the learning module successfully determined the user’s class in all three cases. In addition, the resulting changes to video resolutions were exactly in line with what the user would have found acceptable and, at the same time, the video bandwidth was kept within the channel bandwidth.

VIII. Conclusion

As the volume of internet traffic related to video streaming increases, the importance of having exceptional bitrate adaptation schemes grows. In addition, these schemes should be aware of not only the channel bandwidth, but the surrounding context of the video being streamed. This context encapsulates the content type of the video, and can extend into other dimensions such as amount of motion, geospatial location, and more. We have presented a solution that takes rate adaptation beyond simply changing the quality of the video when the channel bandwidth becomes limited. Our system determines the user’s preferences about video quality, taking into account if the user prefers a drop in temporal or spatial resolution versus quality resolution. We have demonstrated the ability to adapt to channel bandwidth changes by altering these resolutions. In addition, we have shown that by using support vector machines, we can learn the user’s preferences and successfully adapt the video bandwidth in line with these preferences. Such a system is a viable solution to the changing atmosphere of video providers, contexts, and the increasing and diverse consumer base.

References

1. T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, “Overview of the H.264/AVC Video Coding Standard” in *IEEE Transactions on Circuits and Systems for Video Technology* , July 2003
2. S. Wenger, M.M. Hannuksela, T. Stockhammer, M. Westerlund, D. Singer, RFC 3984 “RTP Payload Format for H.264 Video”, Network Working Group, February 2005
3. ITU-T Recommendation H.264, "Advanced video coding for generic audiovisual services", May 2003.
4. H. Schwarz, D. Marpe, T. Wiegand, “Overview of the Scalable Video Coding Extension of the H.264/AVC Standard”, *IEEE Transactions on Circuits and Systems for Video Technology*, September 2007.
5. Y. Ozturk ….(get DIChirp paper)
6. C. Chang and C. Lin, “LibSVM: A Library for Support Vector Machines”
7. J. Ohm, “Advances in Scalable Video Coding” in *Proceedings of the IEEE*, Vol. 93, No. 1, January 2005
8. I. Unanue, I Urteaga, et. al., "A Tutorial on H.264/SVC Scalable Video Coding and its Tradeoff between Quality, Coding, Efficiency, and Performance” from www.intechopen.com
9. S. Wenger, “H.264/AVC Over IP” in *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 13, No. 7, July 2003
10. O. Oyman and S. Singh, “Quality of Experience for HTTP Adaptive Streaming Services” in *IEEE Communications Magazine*, April 2012
11. T. Stockhammer, M. Hannuksela, T. Wiegand, “H.264/AVC in Wireless Environments” in *IEEE Transactions on Circuits and Systems for Video Technology*, July 2003
12. H. Schulzrinne, RFC 2326 “Real Time Streaming Protocol (RTSP)”, Network Working Group, April 1998
13. L. De Cicco, S. Mascolo, V. Palmisano, “Feedback Control for Adaptive Live Video Streaming” in *MMSyss ’11 Proceedingsof the Second Annual ACM Conference on Multimedia Systems*, 2011
14. T. Lohmar, T. Einarsson, et. al. “Dynamic Adaptive HTTP Streaming of Live Content”, 2011
15. “HTTP Live Streaming Overview”, Apple whitepaper
16. A. Zambelli, “IIS Smooth Streaming Technical Overview”, Microsoft whitepaper, March 2009
17. V. Adhikari, Y. Guo, et. al. “Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery”,