# Camera System Paper (Name Pending)

Abstract

one paragraph , what has been done. Why is it important. What is claimed to be success ( measurable values , improvements over other work , the needs fullfilled

## I. Introduction

## The domain of the problem. Come up with use cases.

Discuss the need, answer why existing systems are not offering a solution. Reference all major work

more about who has done what , more towards the domain

cite things as [1]

1. **Packetization and Frame Encoding for H.264**

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]. In addition, it 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

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

- NAL unit syntax structure

- Flexible slice size

- Flexible Macroblock Ordering (FMO)

- Arbitrary Slice Ordering

For a more detailed overview of each of these features and more, please refer to [1] and [3]. The structure of an H.264/AVC encoder can be broken down as follows:



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

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 layers (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. 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 as follows:



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 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. Each packet we send is formatted in the following way:



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 how video bandwidth control is accomplished and what method we use in our system.

1. **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, a method of video scaling can be employed which will 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 scalable video streaming, 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 a certain 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 one single bit stream is transmitted at a time and 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 receiver decoder must be able to adapt to these changes. We will compare two methods of adapting a video stream to varying channel conditions; first, the Scalable Video Coding extension of 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 main methods of scalability, being temporal, spatial, and quality scalability. 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 T0 through Tk. 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 . 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 can be illustrated as follows:



**Figure ## -- Temporal Scalability**

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]. [more elaboration] . Quality scalability works on the same principle as spatial scalability where the layers transmitted are of the same spatial resolution.

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 one bit stream is of a fixed spatial, temporal, and quality resolution for a given sequence of video. There is no built in mechanism to change any of these resolutions midstream. In this, one method to control the bandwidth of the video would be to use different compression parameters at the source encoder prior to transmission. We can partition a video stream into multiple segments (sequences) labeled where *n* is the number of segments for the given video stream and *Tk* 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 *Tk* 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 following form:

**<Diagram of changing encoding parameters (frames changing, label T, sps insertion)>**



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 *Tk,* we propose the following software architecture for the receiver:



**Figure ## -- Decoder 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 decoder thread not currently in use will be set up to decode the next sequence of video. The decoder thread currently in use 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.

In our system, we decided to use a single layer video stream with the proposed architecture above 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. We gain a significant speed boost in the initialization and encoding process by using the encoder in hardware. 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. The effect of altering the different resolutions on the bandwidth is illustrated as follows:

**<Images of bandwidth changes>**

Our experiments using this method have shown us the following:

- For a given input video bandwidth *B* changing the compression bitrate by a factor *K* will result in an output video bandwidth of *K\*B*.

- For a given input video bandwidth *B,* changing the frame rate by a factor *K* will result in an output video bandwidth of *K\*B*.

- We can control the quality of the video by scaling the spatial resolution and leaving the compression bitrate constant; scaling the frame size independently has a limited effect on the bandwidth of the video stream.

1. **The need for Unequal protection**
2. **System Architecture**

Our system architecture can be depicted as follows, with the streaming clients, streaming server, and constituent components:

**<Diagram of system architecture>**



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. A 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. The following sections will describe how the system works in detail.

a) Client Design

The client is designed as a desktop application consisting mainly 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 and playing a certain media 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 we keep a TCP client that will interact with the server. When a new video stream needs to be requested, the client will attempt to connect to the server, and upon successful connection will start the media player. This client will then submit all requests to the server and react appropriately to responses. Within this client/server configuration we have defined our own session management which we will discuss in section V.C.

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 she makes her own decisions about what encoding parameters she wishes. We will go into more detail about user profiling and bandwidth optimization in a later section.

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 8 megapixel camera, at a specified spatial and temporal resolution. When the application launches we set these to a default of 320x240 at 30 frames per second (fps). 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.

c) Session Management

The system consists of a TCP layer for communication between the client and server; this layer also serves 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. The server has a daemon (as previously described) which will be listening for these incoming connections and fork a new thread to service this client. The server thread will first start the encoder which will begin streaming packets containing the encoded H.264 frames to the client. 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 (as in, to scale the video), 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 compression bandwidth. 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, then exit. An example streaming session could be like this:

**(Insert figure of example streaming session)**

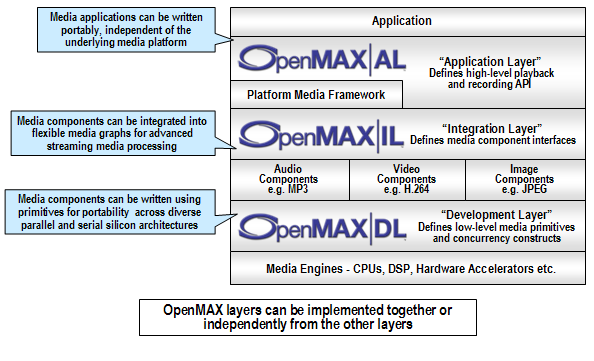
When a camera update is successful the server will send back a response indicating success. The client will also be notified of any malformations in the request.

From the client's perspective, the TCP client object is used to send requests and wait for responses. When the user wishes to send a request, the object sends them in another thread, waiting for a response for a certain period of time; if no response is received, the client times out and an error message is displayed to the user. In addition, an error message will be displayed if we have somehow been disconnected from the server. If a response is successfully received, this response is then displayed to the user. Such a response will indicate either success (if the video has been updated) or failure (i.e. the request was malformed, the video could not update successfully, etc.). This will continue until the user has closed the video window, at which time the media player will stop and the TCP client will be shut down. The server will sense this and stop streaming video to the client, destroying the servicing thread in the process.

1. **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.

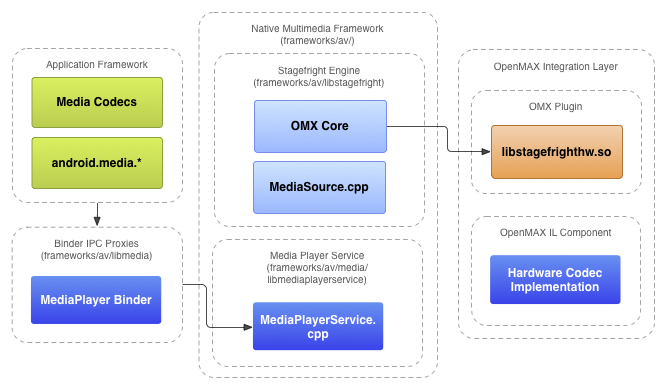
Video is physically encoded on the hardware; to access these components, Android uses a wrapper to the OpenMAX Integration Layer [citation needed] called IOMX which can interact with and utilize hardware multimedia codecs. The OpenMAX Integration Layer is a component based API designed to provide a layer of abstraction on top of multimedia hardware and software architecture. It is also designed to give media components portability across a range of devices [OMX Spec citation needed].



**Figure ## – OpenMAX Layers**

**(May be subject to copyright)**

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 to developers that serve as an interface to the hardware.



**Figure ## – Stagefright Media Framework**

**(May be subject to copyright)**

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; the structure of the sample code and the full sequence of events to encode video is as follows.

Qualcomm's implementation can be broken down into a few different levels:



**Figure ## – Encoder Abstraction Layers**

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 in order to use the hardware codecs. 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 the EncoderActivationInterface class. Here, we simply declare the native functions and use them during the encoding process.

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 the encoder activation interface 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 [Citation?] . From the encoder activation interface, we first set the recipient client's IP address which is set as 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 of type QcomOmxInterfaceEncode, one of the classes developed at Qualcomm for their sample code. In addition, we register a callback function with the encoder interface which will receive the encoded frames. 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. The work to encode the frame is done in a JNI code 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. The 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.

1. **User profiles and bandwidth optimization**

In the proposed system, we wish to be able to optimize the bandwidth of the video stream in order to fit to the channel at all times. Many modern streaming solutions such as Netflix and Amazon Video accomplish this by employing Adaptive Bitrate Streaming (ABR). With ABR, multiple streams encoded at different bitrates are available for the client to consume. Depending on current network conditions, the client will select which of these streams will be appropriate to view. 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. The issue we seek to address with such a system is that only the quality resolution can be affected in order to control the video bandwidth, which may or may not be acceptable depending on the consumer or the application. 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 visual quality is a premium in such a situation. However, if the system were to scale the temporal resolution of the video instead of the quality, visual information would be able to be kept intact at the cost of the less important temporal information. 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 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.

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 defined as follows:

* Class 0: User prefers frame rate over bitrate and size over quality.
* Class 1: User prefers frame rate over bitrate and quality over size.
* Class 2: User prefers bitrate over frame rate and size over quality.
* Class 3: User prefers bitrate over frame rate and quality over size.

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 spatial vs. temporal resolution preference and the second bit will convey the quality resolution preference. This effectively separates the resolutions that will affect the bandwidth and the resolution that will only affect the quality of the video. In this, we can find an optimal way to alter the spatial and temporal resolution such that it will fit the channel and align with the user's desires, as well as alter the quality resolution such that the user will find the video quality acceptable. The creation of the user profile is done using machine learning which we will review next.

b) Machine Learning

In order to create a profile for each user we are using a machine learning algorithm to create a decision function based on the behavior of the user that will be able to predict the class a user falls into given a set of contextual features. 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 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 which denotes what class applies at the instant of the sample. The features that are being used 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 we wish to learn about 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 spatial resolution of the video. Upon this change, we obtain explicit feedback from the user by asking if the video quality is acceptable. If they answer yes, we take the current relevant features and determine which classes the user falls into, adding this sample to the training set. If the user answers no we take no 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 will continue until the user exits learning mode; the user can only exit learning mode 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 on them to ensure accurate selection of the C and gamma parameters. This will create for us 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 spatial resolution of the video we can change the frame size and bitrate of the video. In our system, the main controlling factor is the bitrate. We know that changing the bitrate will change the bandwidth of the video in a linear fashion, namely:

Where is the output video bandwidth and is the output video bitrate. Using this, we defined two bitrate calculations which we can use to determine how to scale the bitrate:

Where is the maximum allowable bitrate, is the input video bitrate, is the bandwidth of the channel, is the input video bandwidth, and is a configurable parameter determining the percentage of available bandwidth is acceptable to fill. The next possible video bitrate is found as follows:

Where is the optimum bitrate for the given spatial resolution, and are the dimensions of the video, and is a parameter which can be configured to find the highest necessary bitrate to deliver high quality video at a given spatial resolution. In order to pick the output bitrate, we take the minimum of the two bitrates above:

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

Calculating the frame rate to find the temporal resolution is similar to the spatial resolution, as frame rate has a linear relationship with video bandwidth. In our system, we are limiting the possible frame rates to 15 and 30 frames per second. In this, a change from one frame rate to the other will simply double or halve the video bandwidth. The quality resolution of the video can be controlled by altering either the frame dimensions or bitrate independently of each other. When changing the frame size independent of the bitrate, no change to the bandwidth is induced. The effects of this are illustrated as follows:

**<Images of different quality videos>**

We can then control the quality of the video without affecting the bandwidth by simply selecting the bitrate prior to selecting the video size. We will next explain how these calculations are used in conjunction with the user profiles to scale the video for the user in a way that they find acceptable.

d) Scaling Decisions

They way in which video bandwidth will be optimized depends on the class that the user falls into for the given feature set. 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 bitmask we can use 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 be dependent on whether or not the video bandwidth is increasing or decreasing. The following graphic depicts the eight possible decisions we can thusly make:

****

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.

1. **Experimental Results**
2. Test Bed

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



We create a network 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 an actual network we are using dummynet [citation?] to perform network emulation. With dummynet, one can control the traffic over a specific channel by limiting bandwidth, inserting packet losses, inserting delay, etc. On Linux, dummynet runs with the operating system as a kernel module and can be configured using the ipfw command line interface application. 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 ipfw, 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 what the bandwidth is. In addition, the client can detect changes in channel bandwidth and react accordingly.

1. Results
   1. bandwidth measurement tool etc
   2. results
   3. Conclusion

Appendix

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. LibSVM