# Intelligent Camera System Technical Report

## I. Introduction

## II. Background

## III. Camera System

The camera system that we have developed consists of a video client and a streaming server. The video client has been developed for the desktop and is written in C++ using the Qt framework. We chose to design it this way in order to attain the speed we need with a native application while maintaining the cross-platform flexibility offered by Qt. 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, while a client can view any number of video streams. The following sections will describe how the system works in detail.

#### a) Video Streaming System

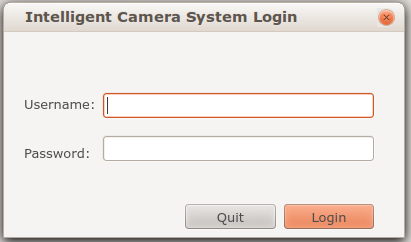
###### i) Camera Server Configuration

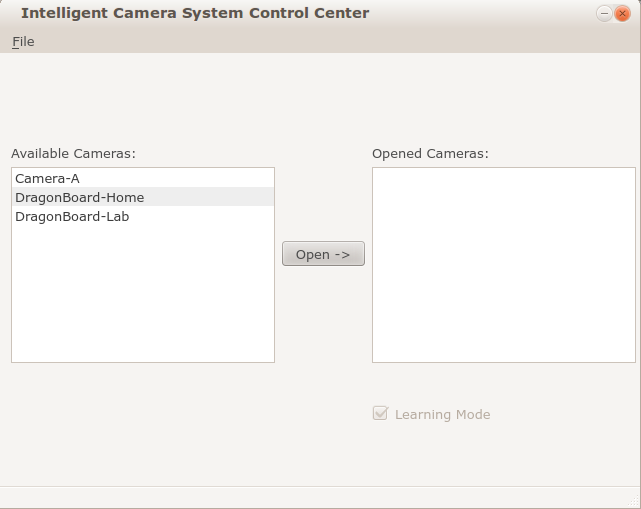
Our camera server application runs on a DragonBoard APQ8060A development board utilizing a Qualcomm APQ8060A processor. The application captures live video from one front-facing 8 megapixel camera, at a specified spatial and temporal resolution. When the application launches we set these to a default of 320x240 at 30 frames per second (fps). Frame data can be captured and processed via a callback which is invoked on each new frame. In addition, there are callbacks for when the camera is created and destroyed. Spatial and temporal resolution of the captured frames can be changed as desired by simply restarting the camera with the new parameters.

**(Insert Figure of Sequence Diagram?)**

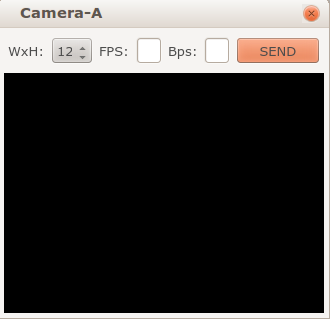
###### ii) Client Configuration

The video client is a desktop application that we have developed using Qt for C++. In terms of the user interface, it consists of a login dialog, a control center, and a video window (media player).

**Figure – Login Dialog**

**Figure – Control Center**

The control center allows the user to connect to and stream from various camera servers. A list box of available cameras displays which server this client is configured to be able to stream from, and a list box of opened cameras indicates which servers the client is already connected to. The items in the list box are Camera objects that we have defined to contain server information such as DNS, IP address, and port. This widget is populated via an XML configuration file for which we have defined a parser class.

**Figure – Video Window**

The video window provides all of the controls necessary during a streaming session and displays the video in a media player. The media player has been designed using LibVLC, a set of media player libraries provided by the VideoLAN group used in their popular VLC player [citation needed]. We encapsulated the LibVLC functionality required by the client in a wrapper class and added a second wrapper to this class to provide Qt specific functionality on top of VLC. A custom widget encapsulates this VLC/Qt interface class to display the video stream desired. The media controls are held in a custom widget and allow the user to send requests to the server in order to scale the video. The way in which this is accomplished is described as follows.

Qt provides a mechanism for allowing callback functions between objects, using what they call signals and slots. A signal is a function with a certain signature that can be “emitted” from an object; another object may have a slot with the same function signature which the signal can be connected to. When the signal is emitted, the slot that is connected to it is invoked. In our application, because the network client is held in a video window object and not the controls widget object, we use this feature to send our requests. The video window connects a signal of the control widget to one of its slots, both of which take four Qstring objects as parameters. When the “SEND” button is clicked, the controls widget object emits this signal, passing the width, height, frame rate, and bitrate as parameters. The slot in the video window is then invoked, and the client sends a request to the server. The network and protocol aspects of this functionality will be discussed in the next section.

###### iii) Network Layer

The system consists of a TCP layer for communication between the client and server, and a UDP layer for streaming video from the server to the client. We will discuss these two layers and their connection in depth.

TCP is used for reliable communication between the client and server, as well as to begin and alter a video streaming session. On the server, when the application starts, we instantiate a daemon (using an Android AsyncTask object) which will listen on port 7744 for incoming connections. When a connection is accepted, the daemon spawns a new thread to process this client's requests. When the server thread is initially created, the first action it will take will be to get the client's IP address and set the encoder interface to stream to this address. The server thread will then start the encoder which will begin streaming H.264 packets to the client. This thread then enters a loop in which it will service the client's request until it is detected that the client has disconnected. Such a request will either be to update or stop the encoder. We have defined a very simple protocol for submitting such requests. To update the encoder (as in, to scale the video), the client will send the following:

**start <frame\_width> <frame\_height> <frame\_rate> <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 and stop the video stream, then exit. An example streaming session could be like this:

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

The client will attempt to connect to a server when a video window is opened via an instance of a TCP client class that we have defined. If unable to connect, a useful error message is displayed. Upon successful connection, the client starts the media player by configuring necessary parameters (i.e. cache size, playback rate, etc.), and informing VLC to play a raw UDP stream with the following MRL:

**“udp/h264://@:[port\_num]”**

Where [port\_num] is the port that the server will be streaming from. As packets begin to stream in, the video is demuxed and displayed in the widget.

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. We then wait for a response for a certain period of time; if no response is received within this time period, 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. This TCP layer is closely related to the UDP layer, which we will now explain.

UDP is used for the actual video stream. We decided to use UDP due to its fast and connectionless nature; we did not wish to have all of the overhead of a transmission protocol like TCP [citation needed].

## IV. Experimental Results

## V. Conclusions and Future Work