SWE4211 Lab 11: Camera Streaming

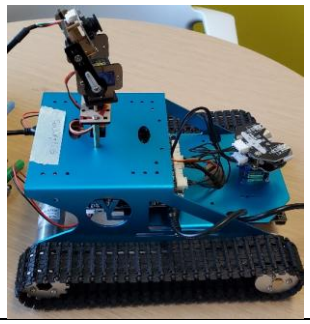
1 Introduction

In this lab, our goal is to start using the camera on our robot and receiving images across the network. We also want to see how the camera impacts robot performance. The camera, obviously, has the potential to use a lot more CPU bandwidth than other parts of the code. In this lab, we will see how that impacts the performance of the system.

2 Lab Objectives

- Capture images from a CCD camera using OpenCV
- Transmit images across the network to base computer
- Cross compile with libraries on a local machine
- Develop multithreaded software subject to real time performance constraints
- Identify performance capabilities for a real time system

3 Laboratory Equipment Needed

Robert Charger 	Yahboom Robot for Raspberry Pi Smart Car AI Starter Kit 
	1 Micro SD Card with Rasperbby PI OS Installed 

4 Deliverables / Submission

4.1 Video Submission

Submit a brief video which:

1. Demonstrates that you can receive video from your camera with the Java application.
2. Demonstrates some form of motion in the video. This can be simple, like a wave or some other motion.[1]
3. Demonstrates that you can successfully connect and disconnect from the image stream and then reconnect again.

4.2 Writeup Submission

Submitting a report with the following contents:

1. Introduction -> What are you trying to accomplish with this lab? This section shall be written IN YOUR OWN WORDS. DO NOT copy directly from the assignment.
2. Screen captures
   a. Include a screen capture of your software building without warnings.
3. Measurements
   a. Include your raw data tables. Be sure to label data with units as is appropriate. Also indicate whether datagrams are being dropped in transmission, and what your networking characteristics were (wired / wireless, laptop wired or wireless, etc.)
4. Plots from data
   a. Include the plots described in the report writeup, appropriately labeled.
5. Discussion
   a. Based on your plot and analysis, is the data transfer rate roughly constant? Should it be? What computing phenomenon governs this aspect of the system? Could the way transmission timing is handled be skewing your results?
   b. Would performance differ if you used a wired network connection instead of a wireless one? Why or why not?
   c. Based on your image quality adjustments, which quality setting is optimal? What evidence in your graph data supports this choice?
   d. When encoding your images, is more or less compression preferable? Justify your answer.
   e. To stream images reliably, what resolution, quality, and frame rate would you choose? Why? In selecting, remember that you need to consider CPU usage, meeting deadlines, bandwidth usage, and image quality settings.
   f. What are the sources of jitter in transmitting images in this system, and how would jitter appear to a user?
   g. As it currently behaves, is this system a hard, firm, or soft real-time system?
   h. Could you measure this system's latency from the demo video provided by your instructor, or can you devise a method that uses the robot, a computer screen, the Java program, and a second video camera?
6. Things gone right / Things gone wrong
   a. This section shall discuss the things which went correctly with this experiment as well as the things which posed problems during this lab.

[1] Yes, one team in the past did the Macarena.

7. Conclusions
   a. What have you learned with this experience? What improvements can be made in this experience in the future?

This material should be submitted as a single pdf file.

5 Raspberry Pi Changes

5.1 Installing OpenCV on the raspberry pi and configuring the Camera

Before we can really get started with this lab, we need to set up the Raspberry Pi so that the camera is active and that we have the image processing libraries installed on the Raspberry Pi. To do this, start by installing OpenCV. OpenCV is a set of libraries that handle image manipulation and processing. We will install these by using apt-get. Because there may be changes in dependencies and source locations, we will first need to do an update and then install the libraries using the commands below. This installation will take roughly 5 – 10 minutes depending on network bandwidth.

```
sudo apt-get update
sudo apt-get install libopencv-dev
```

Once this is completed, you should be OK with your robot.

If for some reason you receive linker errors later, it is likely because your virtual box development environment is not rsynched with the libraries on the Pi. To fix this, run the commands we did in lab 2 on your virtual machine to synchronize libraries. These commands are:

```
cd /
sudo mkdir rpi_sysroot
sudo rsync -azP pi@<yourpi>:/usr/ /rpi_sysroot/usr/
```

6 Compiling the code

OpenCV poses some challenges to cross compilation. It uses a set of different libraries than we have used before. However, through the magic of CMAKE, most of these oddities are taken care of for us and we can develop the program in Eclipse as we have done for previous labs. However, developing outside of Eclipse may be more challenging due to the number and format of the OpenCV libraries. In the past, no student has been successful at compiling using WSL due to these issues.

7 UML

The design for this project is relatively simple, as is shown in Figure 2.

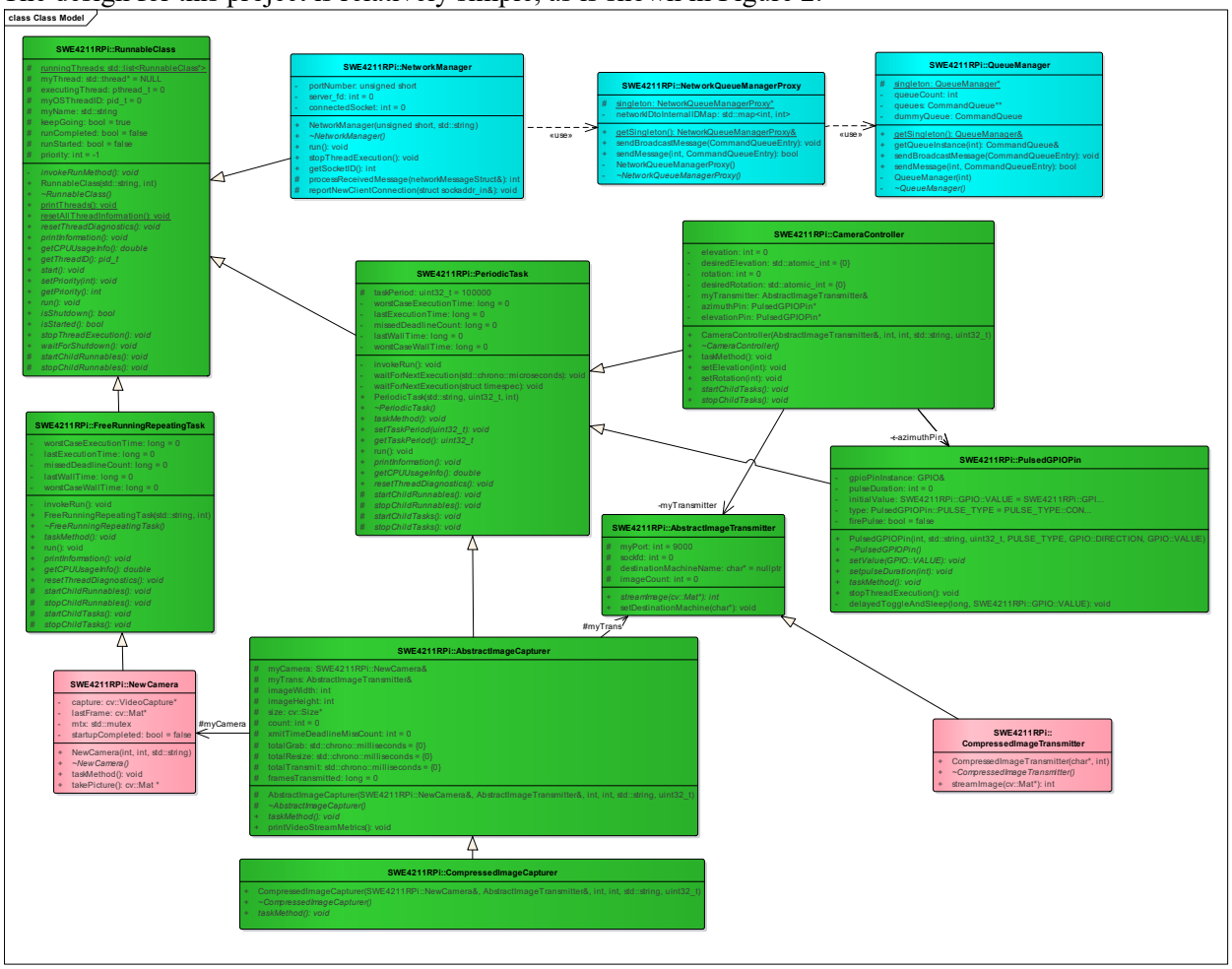


Figure 1 Simplified Lab UML. This UML excludes some of the classes you have used extensively before that are not directly relevant to the code you will be implementing. The pink are the main files you will work on. The green files that have been provided to you and been used before.

In the baseline_docs folder is a set of comments for each of the classes which you are to create.

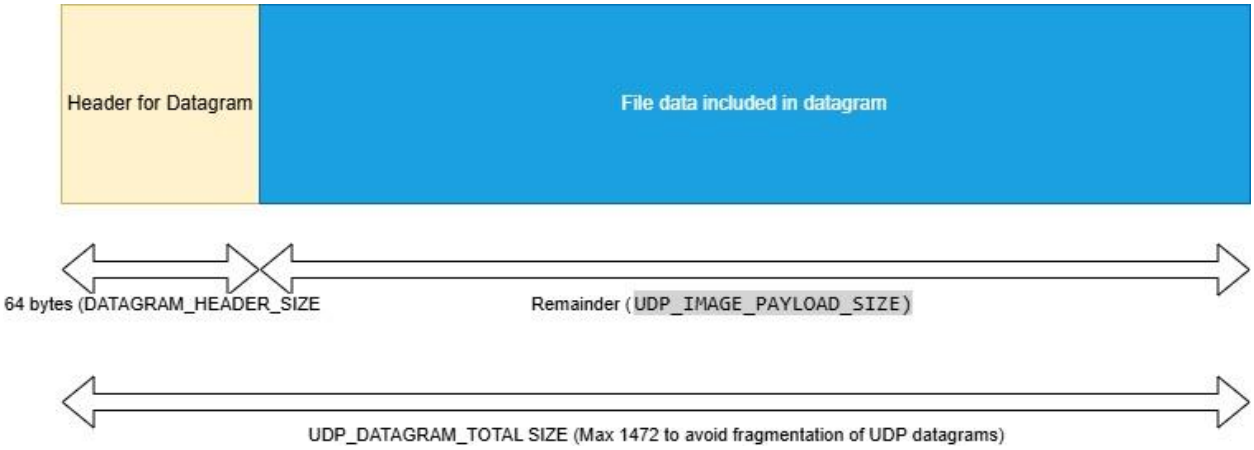


Figure 2 The contents of an individual UDP datagram. The header indicates which portion of the image is being sent in the given datagram and the data portion is the subset of the image that is being sent in this datagram.

8 The Image Transmission Format

The images to transmit are written out to a buffer by your code, and then those compressed images are transmitted via a UDP protocol. To do this, your code will use the OpenCV Image write methods. You will add a compression factor to the file write for jpg images as is shown in Figure 3. The higher the compression factor the smaller the file will be, but also there is an increased likelihood of there being loss in the image. The compression factor you will use is passed in as a parameter to the class.

```
imencode(".jpg", *image, buf, { IMWRITE_JPEG_QUALITY, 10 }); // Quality of 10
imencode(".jpg", *image, buf, { IMWRITE_JPEG_QUALITY, 90 }); // Quality of 90.  Higher
```

Figure 3 Writing compressed image files. The quality parameter can range between 0 and 100. Default value is 90. A higher value will mean a higher quality image, but also a larger image (i.e. less compression). You will also need to include #include <opencv2/imgcodecs.hpp> to obtain the appropriate definitions.

Once the image has been written out to a buffer, the image is transmitted using multiple UDP datagrams from the Raspberry Pi to the Java application, as is shown in Figure 2. Each datagram holds a segment of the overall file. The datagrams are sent in order and are received by the Java program, which will then recombine the datagrams together to form the file and display the image. In essence, the udp protocol is somewhat of a simple ftp protocol.

Offset (Bytes)	Index (int32_t)	Parameter	Size	Description
0	0	Zero	Int32_t	This parameter is always zero.
4	1	Datagram count	Int32_t	This is the index of the given datagram in the context of the entire file. This value will start at 0 and increase until the entire image has been sent.
8	2	Bytes to copy	Int32_t	This is the number of bytes in the datagram that are valid and related to the image. For each datagram except for the last one, this value should match UDP_PAYLOAD_IMAGE_SIZE. For the last datagram, it will be whatever is left over.
12	3	Total Size	Int32_t	This is the total size, in bytes, of the image that is to be transmitted. This value is from the struct stat st_size element.
16	4	rows	Int32_t	This is the number of rows in the image.
20	5	columns	Int32_t	This is the number of columns in the image.
24	6	startTime	Int32_t	This is the start time for the image transmission.
28	7	imageCount	Int32_t	This is an increasing number that increased by 1 each time an image is transmitted. For a given image, all of these are the same. However, when the next image is sent, this value will be increased by one.
32	8	Zero	Int32_t	Always zero.
36	9	Max bytes in datagram	Int32_t	This entry always is MAX_BYTES_IN_DATAGRAM
40	10	Zero	Int32_t	Always zero.
44	11	Zero	Int32_t	Always zero.
48	12	Zero	Int32_t	Always zero.
52	13	Zero	Int32_t	Always zero.
56	14	Zero	Int32_t	Always zero.
60	15	Zero	Int32_t	Always zero.

Figure 3: Header structure for UDP messages.

The Java application which receives the images has several features which will help you when debugging the program. To start the program, run the main, passing along 1 parameter to the main program. The first parameter is the port that the Java code will be listening to. The program can be run from the raw source code. As is shown in Figure 4, the program shows the images received in soft real time (Java by its very nature and garbage collected environment is not a solid real time environment.) The program also offers several features to help with data analysis. The average frame rate (frames pers second) is shown to the right of the image, as well as the average transmission bandwidth.

The Pi Image streamer program you will be working with also offers some features that are useful for data analytics, as is shown in Figure 5.

\

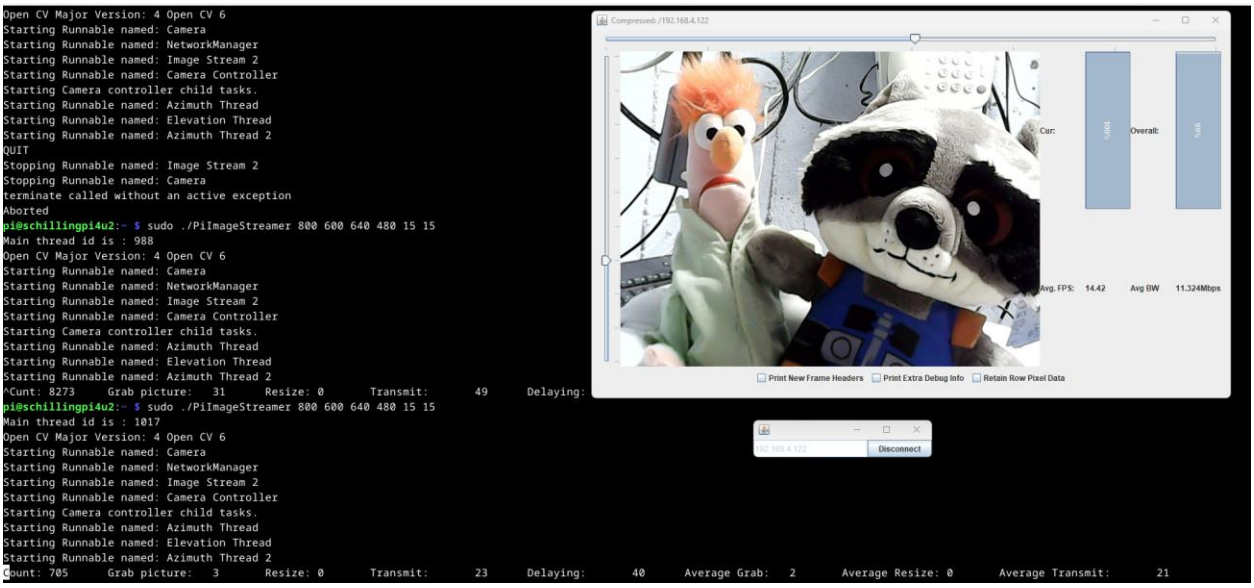


Figure 4: The Java image reception app receiving images with the Linux shell in the background. The image being transmitted is 640 x 480. The shell window shows the times for grabbing the picture from the camera, resizing it from the 800 x 600 raw image, transmitting via UDP, and then delaying for the next transmission. In this case, the images are to be sent via UDP at 15 frames per second. The average times are also shown. The Java program shows the image, the average frames per second it is receiving (14.42), and the average bandwidth associated with image streaming in Mbps (mega bits per second). Note that both characters are happy to be starring in this photo, even though they are from completely different universes. The UI for this year is slightly different than this photo. See the video demo for full details.

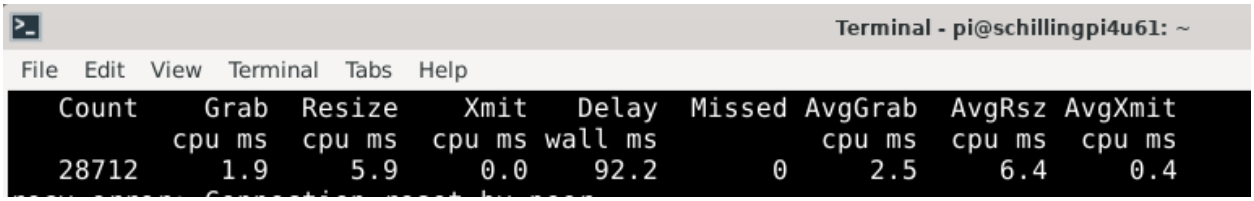


Figure 5: Basic console output.

9 Performance Analysis

9.1 Part 1

First, from a performance standpoint, we want to measure the impact of compression on the image, ignoring any real-time aspects for now. Start by setting both the camera resolution and the stream resolution to 1280 x 960, and set the frame rate to 10 frames per second. Set the image quality to 100%, and then start the camera streaming. Record the average bytes, the stream bandwidth, and the average transmit time in ms. Next, set the image quality to 80% and repeat the measurements. Continue in decrements of 20, so that you have measurements at 100, 80, 60, 40, 20, and 0. In addition to these quantitative measurements, qualitatively examine the image at each quality level and note any observations.

When finished, plot the average bandwidth versus image quality, the average transmit time versus image quality, and the average kilobytes per frame versus image quality. Take note of what you see.

9.2 Part 2

After you have completed the implementation, you are to measure the time it takes for the various activities to occur. Your code as it is running on the pi should be printing out how long it takes to capture the image from the camera, resize the image, and transmit the data. We would like to see the impact of changing this size. To do this, start with the camera capturing 1280 x 960 images. Then, complete the following table, recording the times to the nearest ms. Note that the number of bytes transmitted is calculated, as is the Transmission rate. When doing this, choose the image quality that you feel is most optimum based upon the graphs from Part 1.

Native Camera Resolution	Trans. Width	Trans. Height	Average Grab Time (ms)	Average Resize Time (ms)	Average Transmit Time (ms)	CPU Utilization (htop for all threads)	Average Bandwidth (Mbps)	Average Bytes Transmitted
1280 x 960	**1280**	**960**						
	1024	**768**						
	800	**600**						
	640	**480**						
	320	**240**						
1024 x 768	**1024**	**768**						
	800	**600**						
	640	**480**						
	320	**240**						
800 x 600	**800**	**600**						
	640	**480**						
	320	**240**						
640 x 480	**640**	**480**						
	320	**240**						

The CPU utilization should be measured for all threads that are running and should reflect the total amount. You can obtain this from htop. Note that all these values all will bounce around a bit as the program is executing. Try to pick a value that is in the center that is representative of what you see, but do not worry about exact precision.

When finished, plot the image grab time versus camera resolution in megapixels, the grab time in ms versus the streamed image size in megapixels, the resize time versus camera image resolution in megapixels, the resize time versus the streamed image resolution in megapixels, the transmit time versus streamed image size in megapixels, the bandwidth in Mbps versus Streamed Image resolution, and the average bytes transmitted versus streamed image resolution in megapixels. You'll likely want to use a scatter plot to get these.

After you have completed this, you will have some information that can be used to do a bit of performance analysis.