Project 2 Report:
A Simple Data Link Layer Protocol

Michael Bowyer

University Of Michigan- Dearborn

ECE 570 – Fall 2021

mbowyer@umich.edu

I. Project Description

The project described throughout this report is one of a simple data link layer protocol between two machines using simple python programing. The general idea of this project is to serialize a file from one machine, send the file to another machine which sends the identical file back. The desired result is to receive the identical file which was originally transmitted, without errors. However, the return file transmission occasionally contains errors which must be handled by the original sender upon reception of the retransmission.

The machine which originates the connection and then sends the serialized file will be known as the "local machine" throughout this report, whereas the machine which receives messages and sends them back to the local machine (with the occasional error) will be known as the "server".

The local machine is responsible for a large part of the file transmission and includes the following operations:

1. Reading user arguments for which file to send
2. Serializing the file
3. Splitting the serialized file into small payloads which can be sent one by one
4. Assembling frames using the payloads
5. Calculating a cycle redundancy check (CRC) for each frame
6. Initiating the connection with the server
7. Sending each frame to the server
8. Receiving the return frame from the server
9. Determining if the returned frame from the server contains errors
10. Retransmitting frames if the received frame does contain errors
11. Deserializing the received frame payloads
12. Recreating the original file from the received frames

Where as the server is expected to perform the following operations:

1. Accept incoming frames from the local machine
2. Inject an error to 20% of the received frames
3. Respond to the local machine with the received frame (20% of the time containing injected errors)

The entire sequence of the local machine and the server's interactions can be seen in Figure 1.

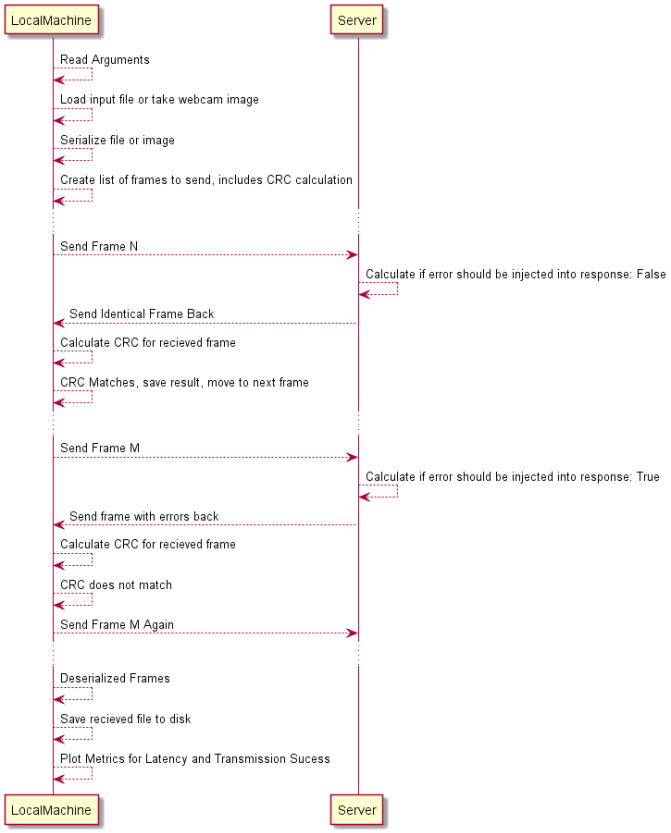


Figure 1: System Sequence Diagram

It can be seen that the local machines second process is to determine if the user of the program would like to use an already existing file or take a new image from the systems

Bit Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	1039	1040	1041	1042	1043	1044	1045	1046	1047
Partition Description	Header								Counter								Payload			Cyclic Redundancy Check							
Length in Bits	8								8								1024			8							

Figure 2: Frame Configuration Diagram

webcam. This feature is what is unique about this implementation versus others within the class and allows for a unique file transmission every single iteration of the program. An example of a webcam image captured and received is shown in Figure 3. It can be seen that each image is identical as expected.

Original Captured Image

Recieved Image By Local Machine From Server

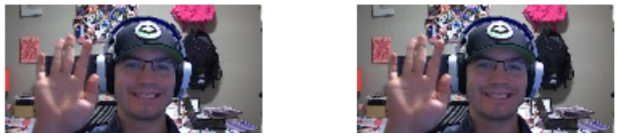


Figure 3: Original and Received Webcam Image Examples

A. *Frame Configuration*

Once the file or image has been serialized by the local machine, it will attempt to assemble a list of frames which can be sent to the server. The breakdown of each frame can be seen in figure 2, which depicts how each frame is partitioned.

Each frame has four partitions, the first being the header which is identical for all frames. The second is the counter, which increments for each frame and rolls over once the maximum value is reached. The counter is utilized to ensure that the received frame is in the expected order. The payload makes up the largest portion of the frame with 1024 bits and contains the actual contents of the file being transmitted.

Finally, the cyclic redundancy check is the final part of the frame. This part is calculated using the preceding 1040 bits from the header, counter, and payload. This is because the goal is to ensure there was no error in the transmission of any part of the frame. The CRC algorithm used a divisor of $x^4 + x^3 + 1$, also symbolized as (11001). The divisor can be changed at compile time and all will still work properly.

II. Problems Experienced

During the development of this project many problems were experienced and are outlined in the following sections.

A. *CRC Calculation using high level types*

The first decision to be made in the development of this project is which type the bitstream would be represented in. This proved to be a difficult decision because python doesn't appear to have any built-in low-level types which enable bit manipulation directly like C or C++. This led to the decision of using multiple types for different applications.

The resulting decision was to store the entire bit stream for a frame as a list of integers taking values of 0 or 1. This made padding additional bits or combining different parts of the frame very simple. However, this led to difficulty in things like bitshifting and XOR, each of which required using a for loop. This decision was difficult because no built in python type served the precise purpose which is needed to handle all of these use cases. The result of this decision to use a list of integers also caused other problems down the line including the slow performance outlined in the next section.

B. *Slow Performance of CRC Calculations*

Due to the slow performance of indexing the bitstream list the resulting program is slower than a user can expect. The initial part of the program which splits the original serialized file bit stream into frames and then calculates the counter and CRC values is the slowest part of the program. For an image of only ~50 kilobytes it takes about 30 seconds just to generate the frames. Once the frames are generated the transmission of them is reasonably quick in comparison. The end result is that the program usually takes at least 45 seconds to complete for a simple image transmission.

In order to overcome this development delay, a smaller text file was created and used in place of images to begin with. This sped up the turn around time from bug discovery to bug resolution allowing for further focus on the webcam image creation and transmission.

C. *Handling of final frame*

Another issue encountered was how to create the final frame of the frame sequence. Often the file being transmitted does not fit perfectly into N number of 1024 bit payloads, which leads to the situation where the final frame cannot exactly fill the 1024 bits usually associated with the payload.

Some solutions explored were zero padding the remaining bits in the front or rear, but this leads to issues the file reconstruction as the added bits can potentially corrupt the file structure. The solution which was decided in the end was to simply reduce the payload size of the final frame to only include the required number of bits which were remaining. This meant that when the final frame was constructed and transmitted, it had to be handled with care to ensure that the frame size we correct and only the exact number of bits are transmitted.

D. *Webcam Image Size Caused Slow Development*

The biggest delay in development of the program came from the introduction of the transmission of webcam images. The original webcam image resolution is Full HD (1920x1080), which when save to disk is roughly 300kilobytes. This file size is much larger than the other files used in this project and often lead to the program taking minutes to finish. To overcome the long feedback loop cycle, the captured webcam image is downsized to 160x80 pixels in

order to make it transmittable in a shorter duration for the sake of the project demonstration. The transmission works properly with the full size image, however the results are not presented here.

III. Results

For each iteration of the program a few metrics are captured. The two main metrics collected are:

1. Frame Transmission Latency: The time in between frame transmission from local machine and frame reception from server to local machine.
2. Packet Transmission Success Rate: The percentage of packets which were sent and received without errors.

Each of these metrics were captured for four different file transmissions shown in the below sections. Each individual section within the appendix contains images of the results of these metrics over time as the frames were transmitted from the local machine.

Each image has two subplots which depict both metrics for the transmission. The x-axis for each graph is the number of frame transmission attempts. The y-axis of the first subplot are the latencies experienced for that frame transmission. The y-axis of the second subplot is the currently experienced frame transmission success rate up until that frames attempted transmission.

The usually experienced latency by the local machine within these tests were averaged around 40 milliseconds. The lowest latency experienced was 30 milliseconds, whereas the maximum was about 90 milliseconds.

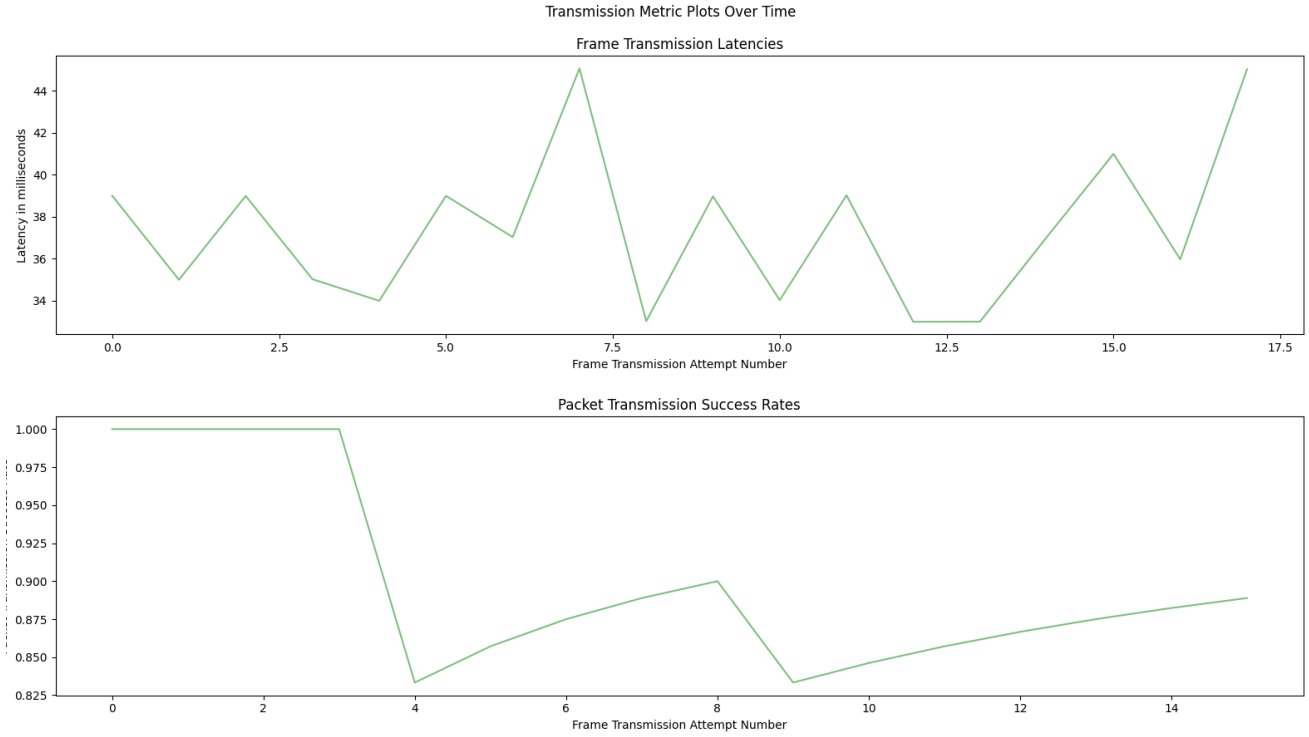
The success rate has an interesting behavior in that it eventually stabilizes around 80%. This is because the server is set to inject errors 20% of the time. When sending small files which contain a few amount of frames, the success rate fluctuates quiet heavily as seen in the simple text file transmission results. This same behavior is exhibited in large file transmissions towards the beginning, however after a sufficiently large number of frames are transmitted the result is that the success rate trends towards 80%, as expected.

IV. Source Code

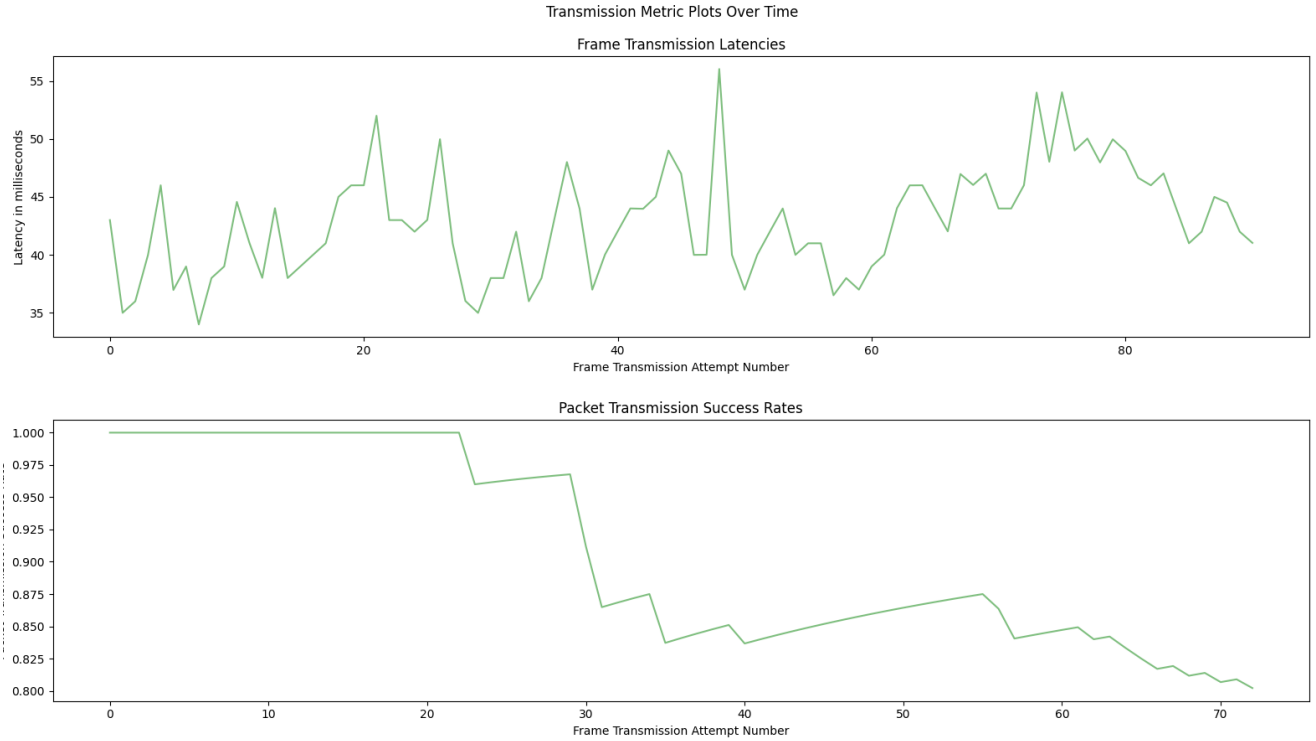
All source code created for this project can be found at Michael Bowyer's GitHub repository: https://github.com/mikebowyer/ECE_570_Project_2.

APPENDIX

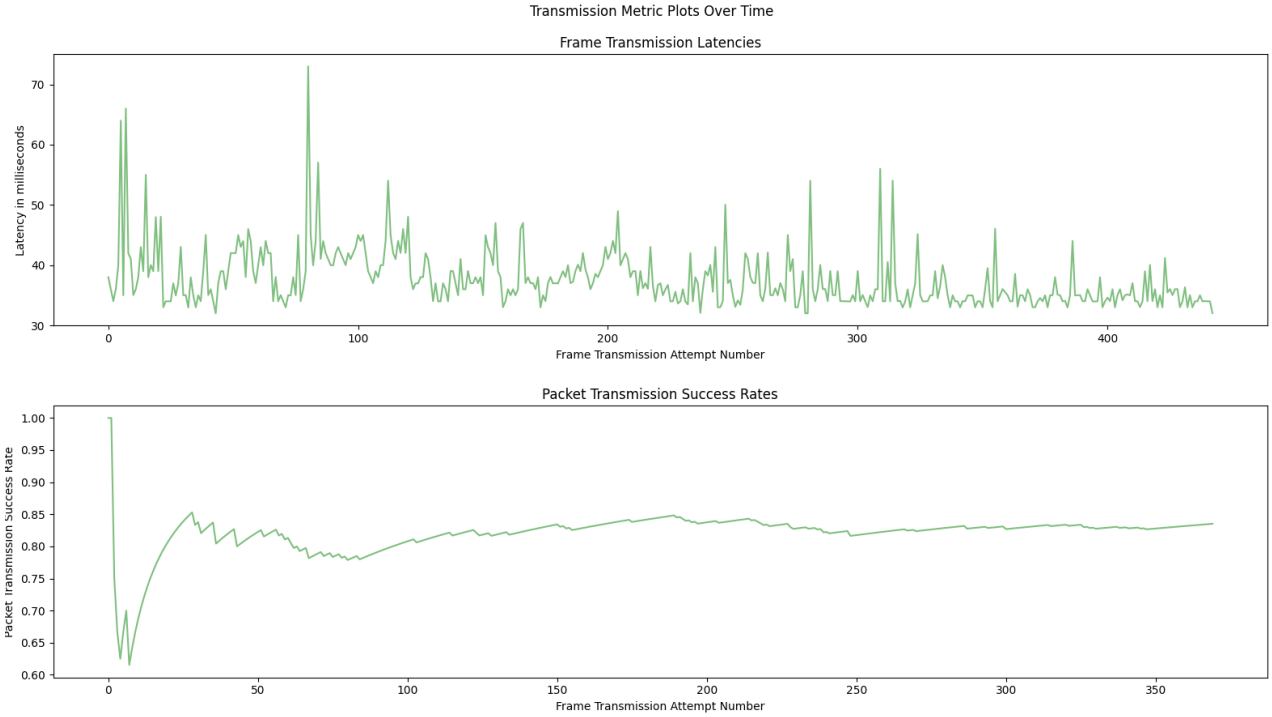
A. *Simple Text File Transmission Metrics*



B. *UM-Dearborn Logo File Transmission Metrics*



C. Bugs Bunny Audio File Transmission Metrics



D. Local Machine Webcam Image File Transmission Metrics