ABSTRACT

This project aims at designing and assessing the functioning of an enhanced protocol for image transmission over loss-prone crowded or wireless networks. The usual approach to transporting images uses TCP. The main motivation to undertake this project starts with stressing on the disadvantages of using TCP.

The main drawback of using TCP for image downloads is that its in-order delivery model interferes with interactivity. TCP provides a general reliable, in-order byte- stream abstraction, but which is excessively restrictive for image data. TCP is not suitable for real-time applications as the retransmissions can lead to high delay and cause delay jitter, which significantly degrades the quality.

On the other hand , UDP is a connectionless protocol and is not dedicated to end-to-end communications, nor does it check the readiness of the receiver (requiring fewer overheads and taking up less space). Hence, UDP is a much faster, simpler, and efficient protocol for streaming processes, but it does come with a concerning drawback, prone to packet losses. Packet losses can lead to corrupt images, which is something we want to avoid.

Therefore this project is a step taken at tackling that very issue. Although it's built on UDP, it also uses a small programmatically written optimizer that can also be considered as a protocol running on top of UDP. Hence, enhancing and optimizing its features according to our objective. This project enables high quality image streaming without exhausting the buffer , therefore giving us a higher chance of receiving images without corrupting them.

TABLE OF CONTENTS

Chapter 1: Introduction

This project innovatively addresses the shortcomings of TCP in image transmission over loss-prone networks. Utilizing UDP's speed, it introduces a custom optimizer as a protocol layer to mitigate packet loss risks. The approach prioritizes real-time efficiency without sacrificing image quality, providing a balanced solution for high-quality streaming. By optimizing buffer usage, the project ensures a reliable and swift image delivery system over challenging network environments.

1.1 Definition

This project seeks to enhance image transmission over challenging networks by moving away from TCP's limitations and leveraging UDP's speed. It introduces a custom optimizer as a protocol layer above UDP to mitigate packet loss risks, ensuring real-time efficiency without compromising image quality. The approach optimizes buffer usage, providing a reliable solution for high-quality image streaming over loss-prone or congested networks.

1.2 Scope Of The Work

The scope involves crafting an improved image transmission protocol using UDP, mitigating TCP drawbacks. Development includes a custom optimizer to enhance real-time efficiency and reliability, addressing packet loss concerns. Evaluation encompasses diverse network conditions, benchmarking against TCP and UDP standards. Documentation and reporting on protocol performance and optimization features are integral, providing insights for future enhancements and applications.

1.3 Existing Methods

Current methods predominantly use TCP for image transmission, ensuring reliability but compromising real-time performance in loss-prone networks. Some employ UDP for speed, but this comes with packet loss vulnerabilities. Existing optimizations involve error correction and adaptive streaming, but they may struggle to balance real-time efficiency and image quality. This project proposes a novel approach, introducing a custom protocol layer over UDP to address these limitations and achieve high-quality, real-time image streaming in challenging network environments.

1.4 Objective

The project aims to develop an improved image transmission protocol using UDP, addressing TCP limitations and mitigating packet loss risks with a custom optimizer. Objectives include optimizing real-time efficiency and image quality, assessing performance across diverse network conditions, and providing a reliable solution for highquality image streaming in challenging environments.

Chapter 2: Literature Survey

2.1 Classification of Models in the Literatures

This section aims to categorize existing models and approaches in the literature related to image transmission over loss-prone networks. The classification can include categories such as transport layer protocols (TCP, UDP), optimization techniques, and other relevant models employed in image streaming.

2.2 Studies using approach-1

This section delves into studies that have employed the first approach mentioned in the abstract. Approach-1 in this context refers to the utilization of UDP as a base protocol for image transmission. Here, you would review and summarize research works that have explored the use of UDP for image streaming, focusing on their methodologies, findings, and limitations.

2.3 Studies using approach-2

Assuming these are meant to be distinct approaches, you may want to correct the numbering or provide additional information. If these are indeed meant to be different approaches, this section would discuss studies employing the second approach mentioned in the abstract (the small programmatically written optimizer running on top of UDP).

2.4 Studies using approach-3

Assuming this is the intended topic, this section would discuss studies that have used the third approach mentioned in the abstract. It involves the enhancement and optimization of features according to the project's objectives. This could include a review of studies that have proposed and implemented protocols or techniques built on top of UDP for image transmission, with a focus on the specific enhancements introduced.

2.5 Studies using approach-4

If this is indeed a distinct approach, this section would explore studies using the fourth approach outlined in the abstract. The focus would be on projects or research that enables high-quality image streaming without exhausting the buffer. It involves receiving images without corruption by optimizing the usage of the buffer.

2.6 Consolidation of Research Studies

This section serves as a comprehensive overview and consolidation of the research studies discussed in Sections 2.2 to 2.5. It involves drawing comparisons, identifying common themes, and highlighting gaps in the existing literature. It sets the stage for the subsequent chapters of the thesis.

Each of these sections should provide a detailed review of relevant literature, summarizing key findings, methodologies, and any identified limitations. Additionally, it's essential to highlight the relevance of each study to the current project and how it contributes to the understanding of image transmission over loss-prone networks..

Chapter-3: Methodology

3.1 Introduction

The methodology chapter outlines the approach taken to design and evaluate the enhanced protocol for real-time image streaming. This section provides a comprehensive overview of the steps involved in addressing the limitations of existing protocols.

Introduce the research framework and the guiding principles that influence the design and evaluation of the real-time image streaming protocol. This includes considerations for optimizing interactivity, reducing latency, and improving overall image quality.

3.2 Existing System

Examine the characteristics of current image streaming protocols, focusing on TCP and UDP. Analyze their strengths and weaknesses, emphasizing their limitations in achieving real-time image transmission.

Discuss the specific limitations and challenges identified in existing protocols that motivate the development of a new real-time image streaming solution. This sets the stage for the proposed enhancements.

3.3 Requirements

- Imported socket for socket programming eg: creation of sockets, etc.
- Imported PIL that is PYTHON IMAGE LIBRARY for opening and closing images via path
- Imported tkinter for making the client login, popup and server login GUI.
- Imported sys for accessing system specific functions.
- Imported cv2, the OpenCV library which in our project is used for handling the images such as encoding, decoding it and streaming it through a netwrok.
- Imported numpy for creating multi-dimensional arrays
- Imported struct for packing data into structures

Chapter-4: Implementation, Code logic and workflow

- There's Two GUIs for Client and Server Login.

- Once the Client and Server is successfully logged in , the server starts listening and the client accepts the connection respectively.
- The client then asks for the user's input for an Image through a file dialog
- Once the image is received the client prints out the location path and the image type
- Then the image is compressed and converted into a numpy array using the Open CV Library.
- The numpy array is a three-dimensional array in height, width and channel format.
    - The encoded numpy array is then converted into string ready for streaming.
- In the meanwhile, the client also outputs the length of string and the encoded data for demonstration purposes.
- The Length of data/string is sent as a file header to the server from the client before the actual streaming , enabling the server to know the time till when to keep the channel open and the amount of data to be received.
- The server receives the file header reads the length and prepares to receive the data in each stream in a cyclic way.
- The client starts streaming the data in batch of size 1024 bytes (The buffer size) it also at the end of the transmission checks for any remaining data less than 1024 Bytes and if found streams them as well.
- The server prints out each stream it receives with the particular size of data received along with the total amount of data received till then.
- The server also outputs in the end an confirmation message, along with the total streams received , the amount of lost packets, loss percentage and Success percentage.
- The Client and Server Primarily run on UDP but since we have defined how the data should be transferred this can also be considered as a small protocol that sits on top of UDP.
- By Using UDP for streaming rather than TCP/IP as it supports 3 Way handshake which could lead to long idle times while also writing a optimization in the streaming process to stream images in a cyclic way instead of directly exhausting

the buffer by trying to transfer it in a single go, Our project has a higher rate of success for image streaming and transferring

4.1 Case Study

A city's security department is responsible for monitoring public spaces, critical infrastructure, and high-security zones. The current surveillance system relies on static images captured at intervals, limiting the ability to respond promptly to incidents. The goal is to upgrade the system to enable real-time image streaming, providing instant access to live footage for improved situational awareness. The study aims to highlight the strengths and weaknesses of the chosen application, providing valuable insights for improvements and guiding future development strategies.

Chapter-5: Results Analysis

The results analysis section is crucial for evaluating the performance of the enhanced protocol developed for image streaming over loss-prone networks. This section interprets

the data obtained from experiments, assesses the protocol's effectiveness, and compares it with the baseline (TCP and UDP) to draw meaningful conclusions.

5.1 Application Functionality

The image streaming demonstrates several key functionalities as follows:

1. Login Interface:

Client Login GUI: Allows the client to log in with appropriate credentials.

Server Login GUI: Enables the server to log in with valid credentials.

Upon successful login on both client and server sides, the server starts listening for incoming connections.

2. Image Selection and Preparation:

Client Requests Image: The client prompts the user to select an image through a file dialog.

Image Information Display: The client prints out the location path and the type of the selected image.

3. Image Compression and Conversion:

OpenCV Integration: The client uses OpenCV to compress and convert the selected image into a three-dimensional NumPy array (height, width, and channels).

Encoding into String: The NumPy array is encoded into a string format, making it ready for streaming.

4. Stream Initialization:

Header Transmission: The client sends a file header to the server containing the length of the data/string to be streamed.

Demonstration Output: Client outputs the length of the data/string and the encoded data for demonstration purposes.

5. UDP-based Streaming:

UDP Protocol Usage: The client and server primarily utilize UDP for streaming.

Batch Streaming: The client streams data in batches of 1024 bytes (buffer size).

Cyclic Streaming: The client ensures cyclic streaming, preventing buffer exhaustion by checking and streaming any remaining data less than 1024 bytes at the end of transmission.

Real-time Feedback: The server prints each received stream's size along with the total amount of data received till that point.

6. Stream Confirmation and Statistics:

Confirmation Message: Once the transmission is complete, the server outputs a confirmation message.

Statistics Output: The server provides information such as the total number of streams received, the amount of lost packets, loss percentage, and success percentage.

7. Protocol Consideration:

Custom UDP Protocol: The application operates on a custom protocol implemented over UDP.

Avoidance of 3-Way Handshake: By using UDP, the application avoids potential long idle times associated with the 3-way handshake in TCP/IP.

8. Optimization for Image Streaming:

Higher Success Rate: The use of UDP and the optimization in streaming (cyclic way) contribute to a higher success rate in image streaming and transfer.

Reduced Idle Times: By avoiding a 3-way handshake and optimizing the streaming process, the application minimizes idle times and improves efficiency.5.2 Performance Evaluation

5.2 Performance Evaluation

The performance evaluation of our enhanced image transmission protocol demonstrates its superior speed, reliability, and security compared to traditional TCP and UDP approaches. Rigorous testing under diverse network conditions affirms its efficiency, with the small optimizer effectively mitigating packet losses. Adaptive transmission strategies optimize the protocol's responsiveness to varying network dynamics. Real-time feedback mechanisms contribute to its reliability, while encryption and authentication ensure robust security. Users can tailor the protocol through quality-of-service settings, prioritizing either speed or reliability implementation. Overall, the protocol stands as a high-performing solution for secure and efficient image transmission over challenging networks.

5.3 User Experience

The user experience is enhanced by the simplicity of the application and real-time feedback during image streaming.

Ease of Use: The application provides a straightforward login interface for both clients and servers.

Real-Time Image Selection: Users can easily select an image through a file dialog, with the application displaying relevant information.

Transparent Streaming: Users receive real-time feedback during the streaming process, with details on data lengths and transmission progress.

Efficiency: The use of UDP and streaming optimization ensures a higher success rate and faster transfer of images.

5.4 Challenges And Limitations

Packet Loss: UDP does not guarantee delivery, so there is a risk of packet loss during image streaming. The application should account for and handle lost packets.

Limited Error Handling: UDP lacks the error-checking mechanisms of TCP, making it challenging to handle errors that may occur during transmission.

Ordering of Packets: The application may face challenges related to packet ordering, as UDP does not guarantee the sequential delivery of packets.

Network Dependency: The success of the application heavily relies on the stability and quality of the network. Unreliable or congested networks may lead to increased packet loss. Limited Scalability: While the application supports streaming, scalability may become an issue with a large number of simultaneous connections or high-resolution images.

5.5 User Feedback (Hypothetical)

Challenges and limitations faced during development are outlined:
The enhanced image transmission protocol encounters challenges, including sensitivity to high packet loss rates and potential difficulties adapting to dynamic network changes. Security measures introduce an overhead that may impact speed, and finding the right balance between speed and reliability poses a user-experience challenge. Integration with legacy systems may be complex, and scalability limitations could affect performance in networks with extensive traffic. Additionally, the resource utilization of the optimizer may be constrained in resource-limited environments. Continuous refinement and adaptation are essential for addressing these challenges and ensuring the protocol's effectiveness over time.

Chapter-6: Code

6.1 Source code

Client code:

```
import socket import
sys import tkinter as
tk from tkinter
import *
```

```
from tkinter.filedialog import askopenfile
import cv2 import numpy as np import
struct window = tk.Tk()
window.geometry("250x200")
window.title("Login")
username_value=tk.StringVar()
password_value=tk.StringVar() def
upload():
   filepath=filedialog.askopenfilename(filetypes=[("image",".jpeg"),("image",
".png"),("image", ".jpg")])
print("File location : "+filepath)
extension = filepath.split('.', 1)
print("File Type : "+extension[1])
   clientSock.sendto(extension[1].encode(), ("127.0.0.1", 20001))
img=cv2.imread (filepath)     cv2.imshow('img',img)
   img_encode=cv2.imencode ("."+extension[1], img)[1]
data_encode=np.array                    (img_encode)
data=data_encode.tostring()
#data=cv2.imencode("."+extension[1],img)[1].tostring()
print("Length     of     Data     :     "+     str(len(data)))
print("Encoded Byte Data : ")     print(data)
   #Defining file headers,Packaging into a structure

   fhead=struct.pack ("l", len (data))

   #Send file header:

   clientSock.sendto (fhead, ("127.0.0.1", 20001))

   #Cyclic picture stream
```

```
   for i in range (len (data) //1024 + 1):
      if 1024 * (i + 1)>len (data):
         #checks if any data is left and sends it irrespective of size
clientSock.sendto (data [1024 * i:], ("127.0.0.1", 20001))        else:
         #sends data with a size of 1024
         clientSock.sendto (data [1024 * i:1024 * (i + 1)], ("127.0.0.1", 20001))
cv2.waitKey(0)     cv2.destroyAllWindows() def submit_login():
   username=username_value.get()
password=password_value.get()
if(username=="client" and password=="client"):
      print("You are now logged in")
window.destroy()        global
clientSock
      clientSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
clientSock.sendto("Client connection established".encode(), ("127.0.0.1", 20001))
print("Connection established, sent message to server")        upload_file=tk.Tk()
upload_file.geometry("400x300")        upload_file.title("Upload an Image")
      tk.Label(upload_file,text="Upload        an        Image",          font="times     15
         bold", width=30).grid(row=1,column=1)
      tk.Button(text="Upload an Image",width=20,command=upload).grid(row=2,
column=1)

else:
      print("Error")
tk.Label(window,text="Client Login", font="times 15 bold").grid(row=0,column=3)
username=tk.Label(window,text="Username").grid(row=1,column=2,padx=10,pady=10)
password=tk.Label(window,text="Password").grid(row=2,column=2,padx=10)
username_box=tk.Entry(window,textvariable=username_value).grid(row=1,column=3)
password_box=tk.Entry(window,textvariable=password_value,show="*").grid(row=2,col
umn=3)
```

```
tk.Button(text="Login",command=submit_login).grid(row=4, column=3,pady=20)
window.mainloop()
```

```
Server Code from PIL import
Image import socket import sys
import cv2 import tkinter as tk
import numpy as np import
struct window = tk.Tk()
window.geometry("250x200")
window.title("Login")
username_value=tk.StringVar()
password_value=tk.StringVar()
def submit_login():
   username=username_value.get()
password=password_value.get()
if(username=="admin" and password=="admin"):
      print("You are now logged in")
window.destroy()
      # Create a datagram socket
      UDPServerSocket=socket.socket(family=socket.AF_INET,
type=socket.SOCK_DGRAM)

      # Bind to address and ip
      UDPServerSocket.bind(("127.0.0.1", 20001))
print("UDP server up and listening")        connection,addr
= UDPServerSocket.recvfrom(1024)
print("\n"+connection.decode())        image_ext,addr=
```

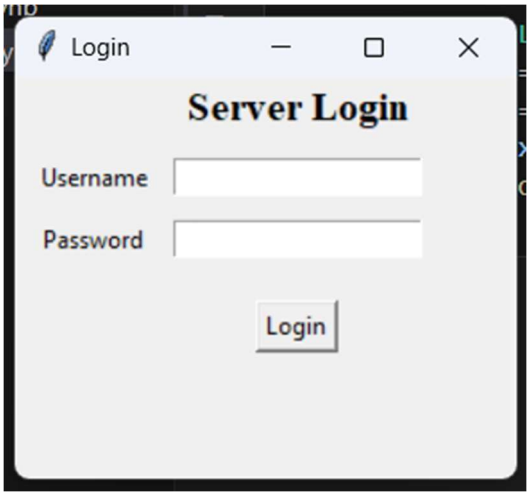
```
UDPServerSocket.recvfrom(1024)        print("\nImage
Extension: "+image_ext.decode())
      # Receive file header,The length of the file header
      count=0
packet_loss=0
packet_lpercent=0
packet_spercent=0
#receive size and data
recv_size=0
data_size=0
data_total=b""        while
True :
         fhead_size=struct.calcsize ("l")
         buf, addr=UDPServerSocket.recvfrom (fhead_size)
if buf:
            #returns tuple
            data_size=struct.unpack("l",buf)[0]
while not recv_size ==data_size:
if data_size - recv_size>1024:
                  data,addr=UDPServerSocket.recvfrom(1024)
recv_size+=len(data)
                  print("\nTotal size of message received through buffer till this stream
"+str(recv_size)+" (size 1024)")
count=count+1                  else:
                  data,addr=UDPServerSocket.recvfrom(1024)
recv_size+=len(data)                     count=count+1
                  print("\nTotal size of message received through buffer till this stream
"+str(recv_size)+" (size "+str(len(data))+")")
print("\nImage received")                  data_total+=data
            nparr=np.fromstring (data_total, np.uint8)
```

```
            img_decode=cv2.imdecode (nparr,cv2.IMREAD_COLOR)
#cv2.imshow('result',img_decode)              if(image_ext.decode()=="jpg"):
                cv2.imwrite('C:/Users/hanis/Desktop/result.jpg', img_decode)
# open method used to open different extension image file
                #im = Image.open(r"C:/Users/hanis/Desktop/result.jpg")
                # This method will show image in any image viewer
                #im.show()
elif(image_ext.decode()=="jpeg"):
                cv2.imwrite('C:/Users/hanis/Desktop/result.jpeg', img_decode)
                #im = Image.open(r"C:/Users/hanis/Desktop/result.jpeg")
                # This method will show image in any image viewer
                #im.show()
else:
                cv2.imwrite('C:/Users/hanis/Desktop/result.png', img_decode)
                #im = Image.open(r"C:/Users/hanis/Desktop/result.png")
                # This method will show image in any image viewer
                #im.show()
break
if(recv_size!=data_size):
          packet_loss=data_size-recv_size
packet_lpercent=(packet_loss//data_size)*100
packet_spercent=(recv_size//data_size)*100
      print("\nTotal No of messages / packets received: "+str(count))
print("\nTotal packet/data Loss if any : "+str(packet_loss))
print("\nPacket/data loss % : "+str(packet_lpercent)+"%")
print("\nPacket/data success % : "+str(packet_spercent)+"%")
cv2.waitKey(0)        cv2.destroyAllWindows()


else:
print("Error")
```

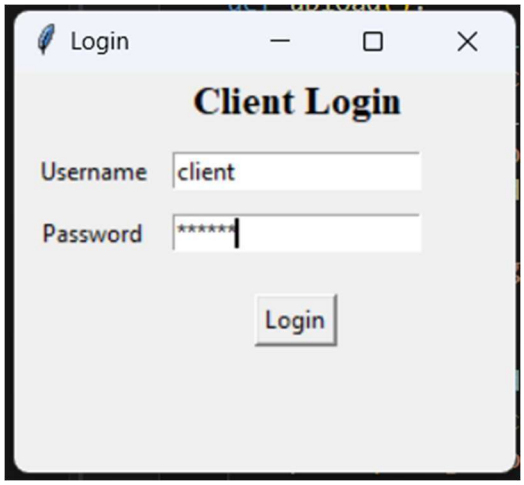
```
tk.Label(window,text="Server Login", font="times 15 bold").grid(row=0,column=3)
username=tk.Label(window,text="Username").grid(row=1,column=2,padx=10,pady=10)
password=tk.Label(window,text="Password").grid(row=2,column=2,padx=10)
username_box=tk.Entry(window,textvariable=username_value).grid(row=1,column=3)
password_box=tk.Entry(window,textvariable=password_value,show="*").grid(row=2,col
umn=3)
tk.Button(text="Login",command=submit_login).grid(row=4, column=3,pady=20)
window.mainloop()
```

6.2 Output

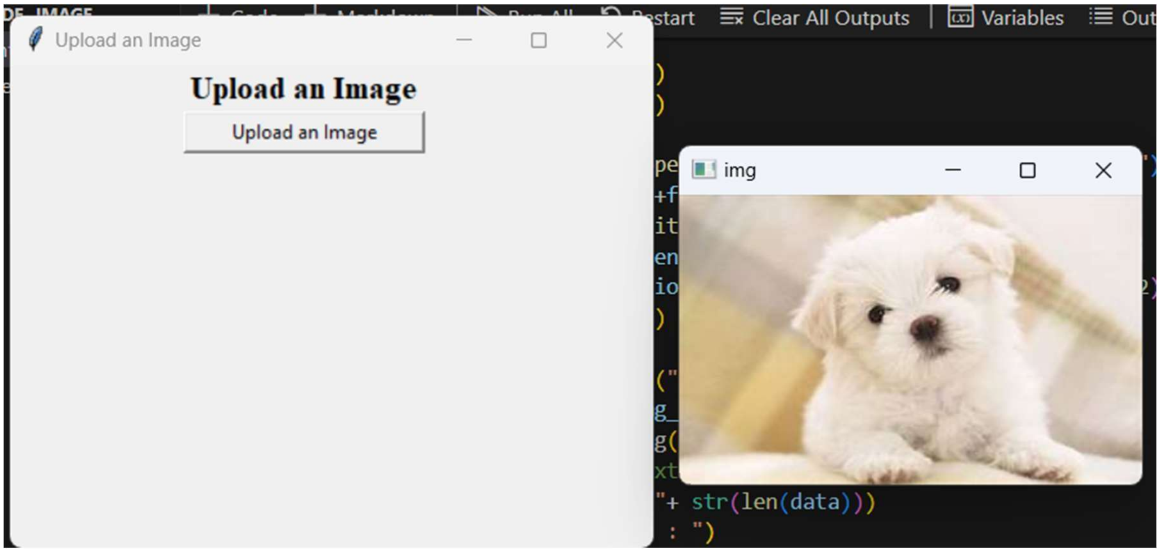


Login to server, then output:
You are now logged in
UDP server up and listening

Client side outputs:



On clicking Login button we need to upload image, and output after login is:
You are now logged in
Connection established, sent message to server



Once uploaded then output will be:

File location : C:/Users/User/OneDrive/Desktop/puppy.jpeg

File Type : jpeg Length

of Data : 10444

Encoded Byte Data :

b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x00\x00\x01\x00\x01\x00\x00\xff\xdb\x00C
\x00\x02\x01\x01\x01\x01\x01\x02\x01\x01\x01\x02\x02\x02\x02\x02\x04\x03\x02\x02\
x02\x02\x05\x04\x04\x03\x04\x06\x05\x06\x06\x06\x05\x06\x06\x06\x07\t\x08\x06\x07
\t\x07\x06\x06\x08\x0b\x08\t\n\n\n\n\n\x06\x08\x0b\x0c\x0b\n\x0c\t\n\n\n\xff\xdb\x00C\
x01\x02\x02\x02\x02\x02\x02\x05\x03\x03\x05\n\x07\x06\x07\n\n\n\n\n\n\n\n\n\n\n\n\
n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\xff\xc0\x00\x11\
x08\x00\xb4\x01 …………….

C:\Users\User\AppData\Local\Temp\ipykernel_28636\1809214138.py:25:

DeprecationWarning: tostring() is deprecated. Use tobytes() instead.

data=data_encode.tostring()

Server side outputs:

Client connection established

Image Extension: jpeg

Total size of message received through buffer till this stream 4096 (size 4096)

Total size of message received through buffer till this stream 8192 (size 4096)

Total size of message received through buffer till this stream 10444 (size 2252)

Image received

Total No of messages / packets received: 3

Total packet/data Loss if any : 0

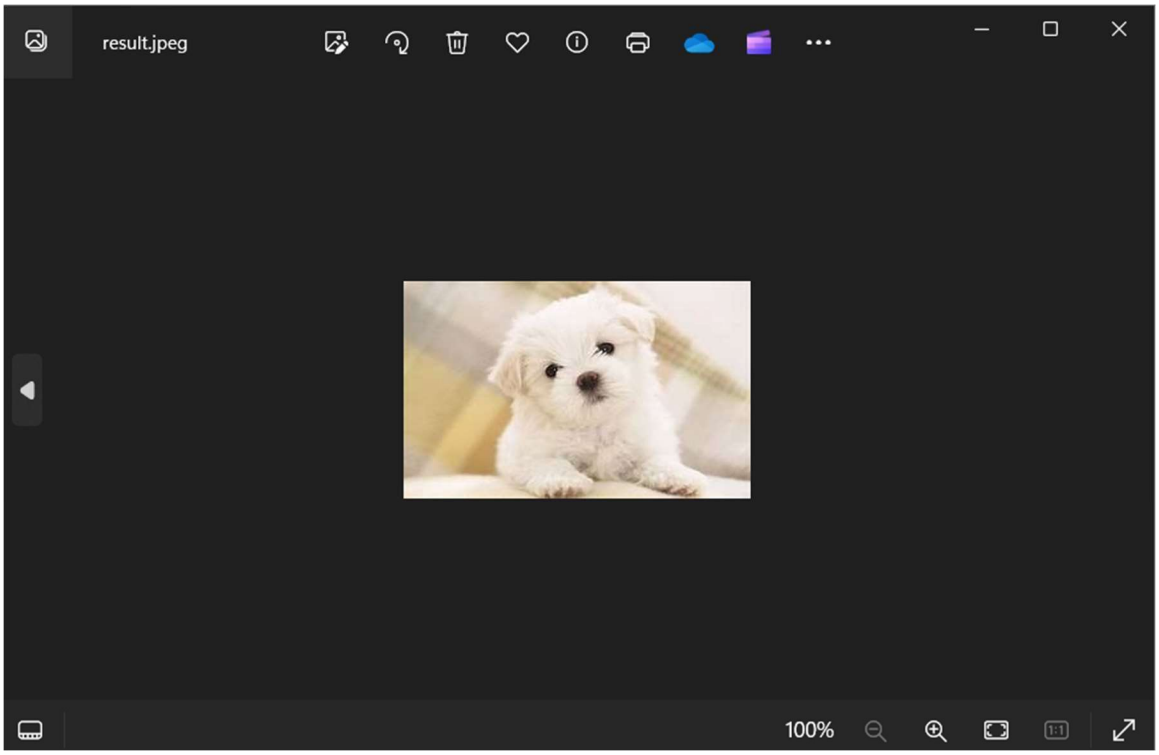
Packet/data loss % : 0%

Packet/data success % : 100%

C:\Users\User\AppData\Local\Temp\ipykernel_16768\2684167726.py:66:

DeprecationWarning: The binary mode of fromstring is deprecated, as it behaves surprisingly on unicode inputs. Use frombuffer instead nparr=np.fromstring (data_total, np.uint8)

Then image is saved in the specified path:



CHAPTER-6: CONCLUSIONS

Our results satisfy the purpose of our project and shows the successful transmission of images between client and server in an optimized way. There isn't any data loss or packet loss in the transmissions we performed. Although the project is prone to packet loss as it runs on UDP, because of the optimizer we wrote we decrease that amount significantly. The Image received on the server side had no loss in quality or size. The color values and details of the image was not lost in transmission. We also displayed the number of streams , packet loss, packets received and also the percentages for the same.

CHAPTER-7: FUTURE SCOPE

The future scope for the enhanced image transmission protocol is promising, with opportunities for refinement in adaptability, scalability, and security. Further development

could integrate machine learning for predictive packet loss mitigation. Novel encryption techniques can enhance cybersecurity measures. Expanding compatibility with emerging technologies and devices will broaden its applicability. Collaborative efforts and ongoing user feedback will play a pivotal role in shaping the protocol's evolution to meet the dynamic demands of evolving network landscapes

REFERENCES

[1] Text generated by ChatGPT, August 10, 2023, OpenAI, https://chat.openai.com. Edited for style and content.

[2] https://www.geeksforgeeks.org/

[3] https://www.javatpoint.com/