Design File

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Title and Authors

* This is for Phase 2 of the Network Design group project.
* Group Members:
  + Alexander Nunez Pepen
  + Md Zahidul Islam
  + Pablo Ruiz
  + Joseph Ayoka

Purpose of the Phase

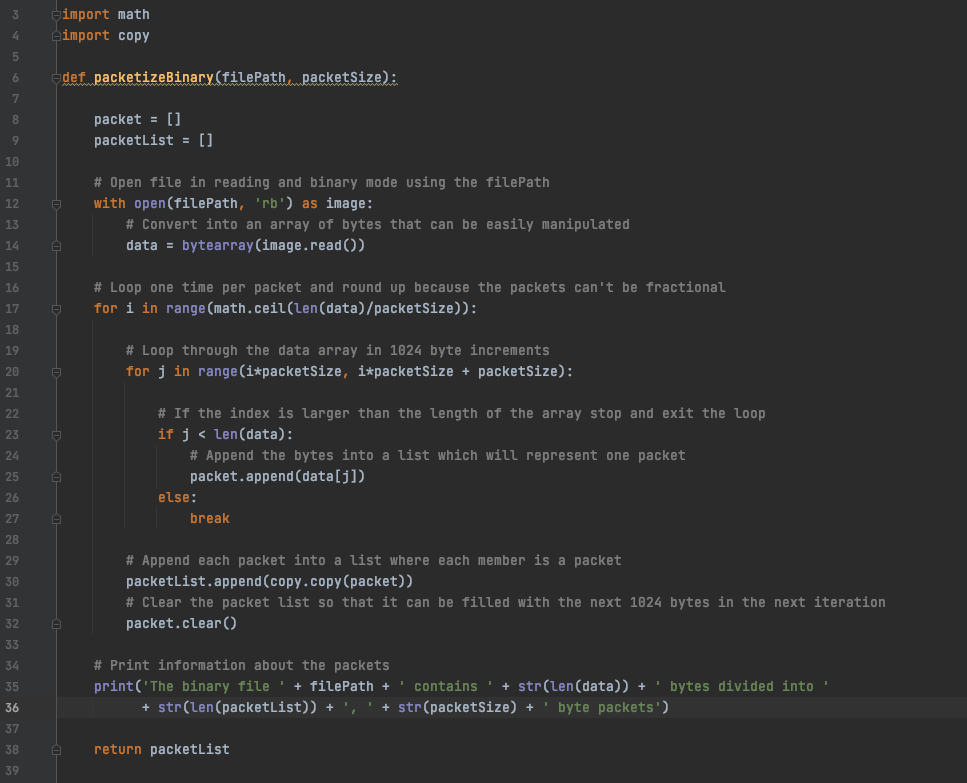
In phase 2, we are implementing the RDT 1.0 protocol using the UDP server and client process. By doing this we want to have the ability to transfer a (.BMP) image file from the UDP client to the UDP server. RDT 1.0 stands for reliable data transfer protocol version 1.0. This protocol can be modeled by two simple finite state machine (FSM) diagrams. One FSM, for the sending side (client) and one for the receiving side (server). When the receiving side receives data (a packet) it extracts the packet’s contents and then delivers the data to the upper layer of the application. After this, the receiving side returns to its initial state which is waiting for a call from the communication channel. On the other hand, the sending side’s initial state is waiting for a call from above, which means it is waiting for data from the application layer. It then manipulates the data to form packets that are then sent into the unreliable channel to the receiving side.

This phase required us to test this protocol by having us implement the RDT 1.0 protocol using the code already developed in phase 1. In phase 1, client and server programs were developed using UDP (user datagram protocol) sockets. For this phase, the phase 1 programs were modified in order for them to send and receive packets.

The main purpose of this phase is to introduce us to reliable data transfer using packets of data. This phase also helps us gain a better understanding of how the application network interacts with the transport layer and then how the transport layer interacts with the network layer. The application layer breaks down the .BMP image file into a string of data. The transport layer needs this data to be converted into data packets to be sent through an unreliable channel in the network layer.

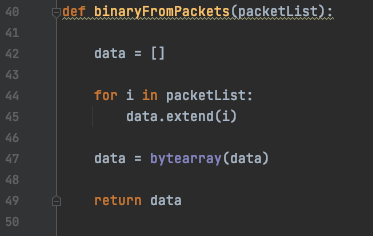
Code Explanation

**Screenshot of “*packetizeBinary”* function in the “*packet.py”* file:**



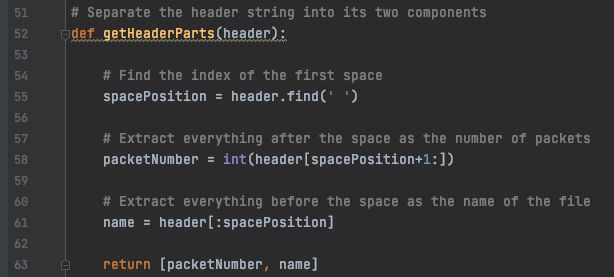
The main purpose of this function is to convert the .BMP image file into byte packets of data. For this function to work the “math” and “copy” libraries need to be imported. The packetizeBinary function takes in two parameters, which are the “filepath”(the file’s path) and “packetSize” (desired size of packets. )The function has two string arrays. One named “packet” and the other named “packetList” that will be used during the data manipulation from image data to byte packets. The function “open” (line 12) takes in the image file’s path and is set to read binary data mode (meaning it will read the image’s data into binary values). The “data” variable is then used to store a “bytearray” of the image data (storing the data as bytes). This is done because it is easier to work with data in a byte array form rather than to used the extracted data from the image file. The first for loop (line 17) is used to iterate the value of the length of the image file’s data size divided by the desired packet size. While this loop iterates there is a nested for loop (line 20) that iterates by i times the desired packet size. The if statement (line 23) appends data into the packet array by using the for loops iteration variable j. This is where the packets are formed. They are appended by desired packet size (which is a parameter of the function) into the packet array. Once j is less than the length of the data string the loop breaks. On line 30, the packetList list copies the packets in the packet array where each member of the packerList list is a packet copied from the packet array. On line 32, the packet array is cleared every iteration so that it can store another packet being formed. On line 35-36, a print statement was added to display useful information about the packets formed and their size. The “PacketizeBinary” function returns the packetList list.

**Screenshot of “*binaryFromPackets”* function in the “*packet.py”* file:**



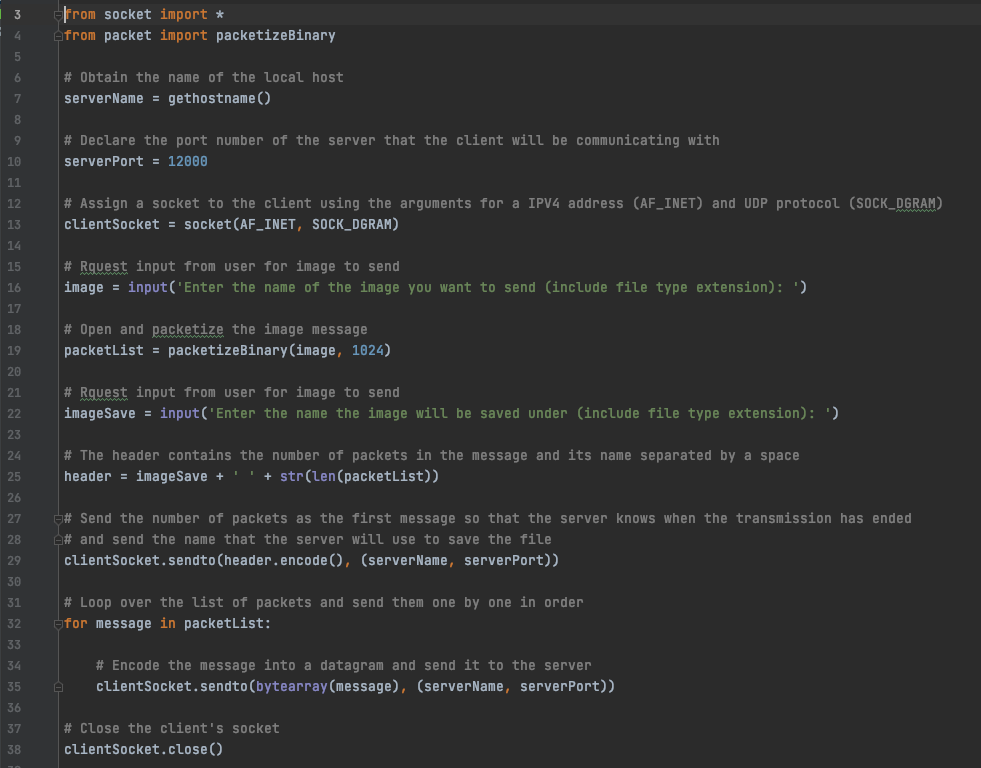
The “binaryFromPackets” function takes in the packetList list and converts the packets within back into data in form of a byte array. To do this the “data” list is used to .extend bits from the packetList list. The for loop scans through the packetList list and .extends the bits into the data array. The “bytearray” function is used on the data array to convert it into an array of bytes. The binaryFromPackets function returns the data array. FYI, each index in the packetList list is a packet of data.

**Screenshot of “*getHeaderParts”* function in the “*packet.py”* file:**



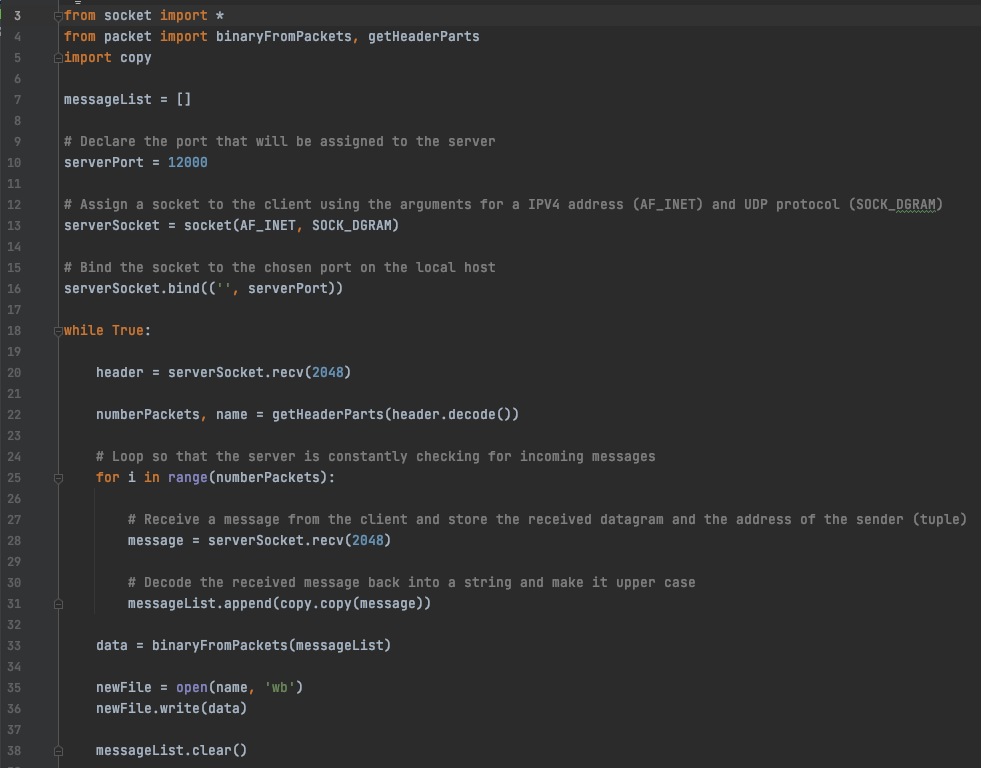
This code was developed to get useful header information before working on the transmitted packets. The header is sent containing two pieces of useful information. The number of packets being sent and the name the user would like to give the image file that will be saved on the server. These two pieces of information are divided by a blank space. So, line 55 searches for that blank space. This allows for like 56 to determine that the integer value (# of packets) is to the right of the space and load the value into packetNumber. Line 61, then loads the string before the space of the name of the file into variable name. The getHeaderParts function then returns the packetNumber and name variable contents.

**Screenshot of “client.py” program**



For this client program, the library “socket” is needed as well as importing the “packetizeBinary” function from “packets.py”. Most of this code is used from phase 1, but there is some added functionality so that it can handle packets. On line 7, the hostname is fetched. On line 10, the serverPort is set to 12000. On line 13, the UDP socket is created into clientSocket. Line 16, prompts the user to enter the name of the image they’d like to send and its extension (.BMP). The user input is store in the variable image. Line 19, calls the function “packetizeBinary” and sends in the arguments image and packet size of 1024. Line 22, prompts the user to input the name they’d like the image sent to the server to be saved under and stores the user’s input into the imageSave variable. Line 25, creates the packet’s header and the useful information (imageSave and the number of packets) and stores it into the variable header. Line 29, then encodes the packet header and sends it through UDP to the server. The for loop on line 32 sends the packet stored in the list “packetList” indexes on each iteration and stores it into the variable message. The content’s in message is sent through the client socket to the server port in the form of a byte array. The client socket is then closed and the program stops running.

**Screenshot of “server.py” program**

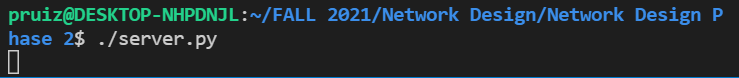
The server.py starts with import socket and our defined function from packet.py. The “messageList” list will accumulate the received packets from the client. The “serverPort” and “serverSocket” variables are the port and the server socket, respectively. In the “while” loop, the “numberPackets” is the total packets of the BMP image and “name” is the filename of BMP file, where incoming packets are stored. Finally in the “for” loop, each received packet from the server is temporarily saved into “message” variable and then appended in the “messageList” list. After receiving all packets, the packets are combined in byte format by calling the “binaryFromPackets” function from “packet.py” and stored in “data”. Finally, the data is written in a new BMP file named “newFile”. This creates a new file on the server-side that can be opened and viewed. The messageList is then cleared so that a new image file may be received. The server stays running so that many images files can be sent to it.

Execution Example

All the procedures for enabling communication between the server and the client are shown in detail in the following example.

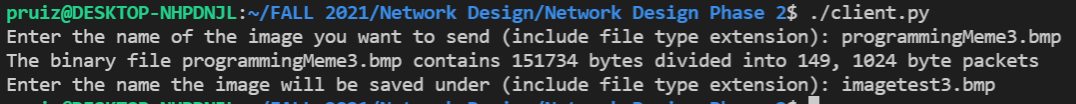
**Step 1:**

Run server.py

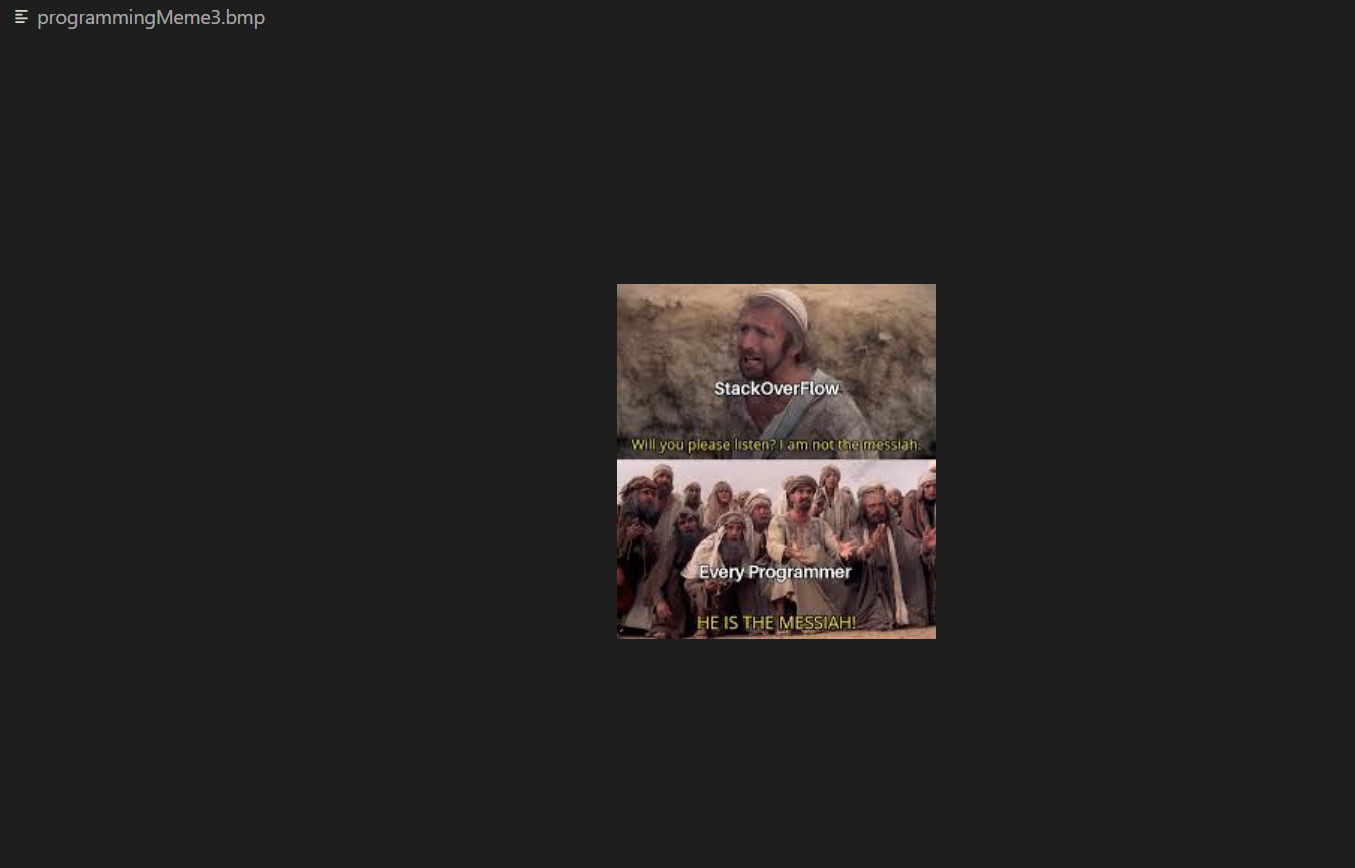


**Step 2:**

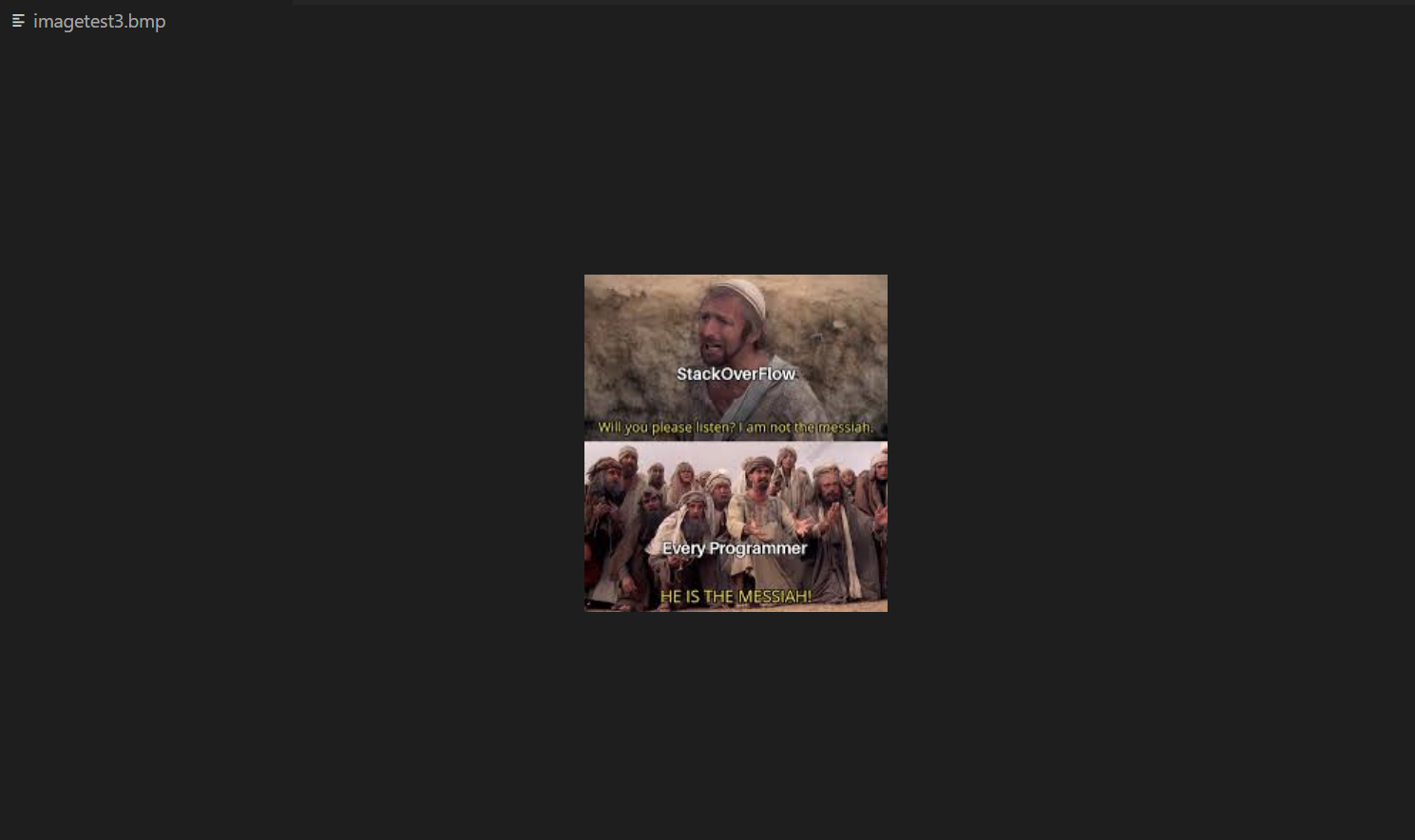
Run the client.py



**Original BMP file**



**Received BMP file:**

****

**Other Examples:**

| **Original BMP file** | **Received BMP file** |
| --- | --- |
|  |  |
|  |  |

**Test results when the codes are run in another Computer with different BMP images:**

For these examples, the original python files are renamed as follows,

server.py -> server\_phs2.py

client.py -> client\_phs2.py

packet.py -> packet\_phs2.py

| **Original BMP file** | **Received BMP file** |
| --- | --- |
|  |  |
|  |  |