Computer Engineering 4DN4 – Lab #3: Online File Sharing Network Application

Name: Jastine Goyena Student ID: 001307092 Email: goyenaja@mcmaster.ca

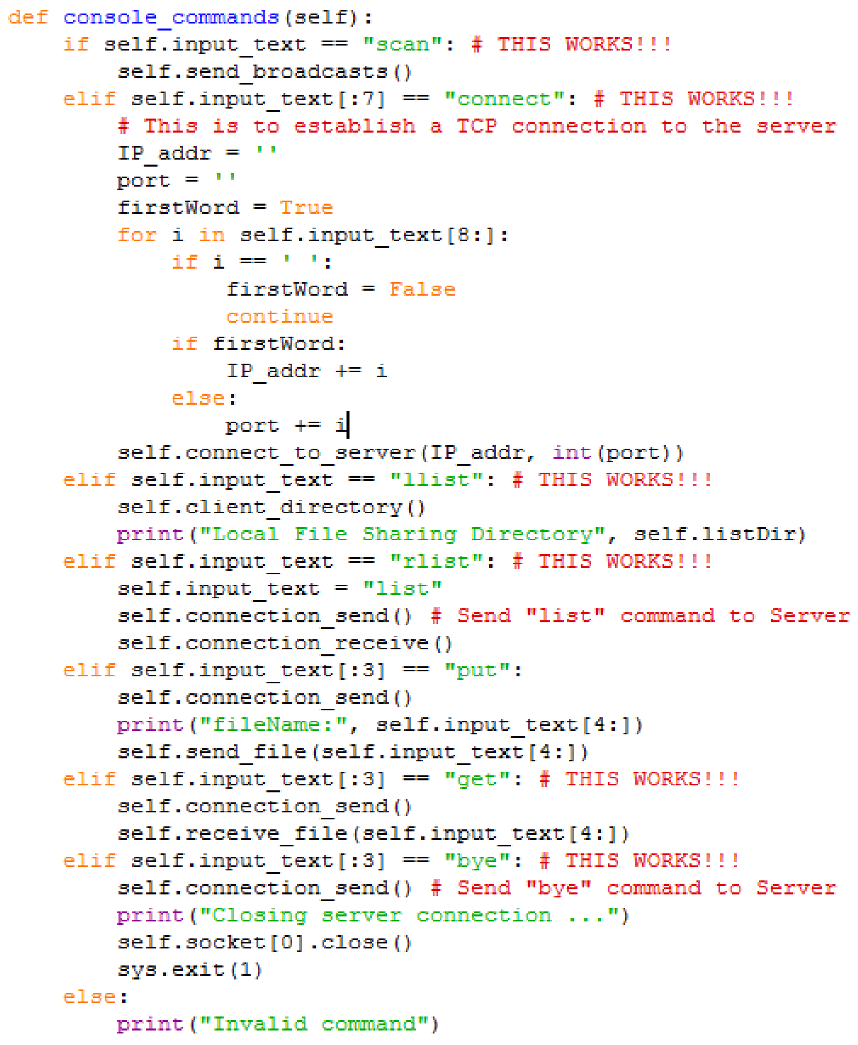
1 INTRODUCTION

Modifications to the Client class in EchoClientServer.py code and the Server class in EchoClientServer_Thread.py provided in the lectures were made such as altering pre-existing methods and addition methods were added to implement the online file sharing network application.

2 CLIENT

2.1 METHODS ADDED AND MODIFIED FOR FUNCTIONALITY

2.1.1 console_commands



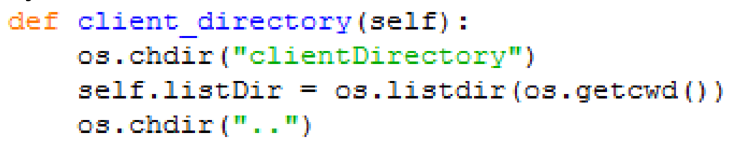
```python
def console_commands(self):
    if self.input_text == "scan": # THIS WORKS!!!
        self.send_broadcasts()
    elif self.input_text[:7] == "connect": # THIS WORKS!!!
        # This is to establish a TCP connection to the server
        IP_addr = ''
        port = ''
        firstWord = True
        for i in self.input_text[8:]:
            if i == ' ':
                firstWord = False
                continue
            if firstWord:
                IP_addr += i
            else:
                port += i
        self.connect_to_server(IP_addr, int(port))
    elif self.input_text == "llist": # THIS WORKS!!!
        self.client_directory()
        print("Local File Sharing Directory", self.listDir)
    elif self.input_text == "rlist": # THIS WORKS!!!
        self.input_text = "list"
        self.connection_send() # Send "list" command to Server
        self.connection_receive()
    elif self.input_text[:3] == "put":
        self.connection_send()
        print("fileName:", self.input_text[4:])
        self.send_file(self.input_text[4:])
    elif self.input_text[:3] == "get": # THIS WORKS!!!
        self.connection_send()
        self.receive_file(self.input_text[4:])
    elif self.input_text[:3] == "bye": # THIS WORKS!!!
        self.connection_send() # Send "bye" command to Server
        print("Closing server connection ...")
        self.socket[0].close()
        sys.exit(1)
    else:
        print("Invalid command")
```

Figure 1. Screenshot of **console_commands** method.

This method was added to make use of **self.input_text** obtained from the console and determines what methods need to be invoked to fulfill a command. For example, if a user enters the **connect** command in the console, a for loop is used to iterate through the **self.input_text** string to obtain the IP address and port number that would be stored in separate variables which will be used as arguments to the **connect_to_server** method within the client.

For the other commands such as **put**, **get**, and **rlist**, the entire **self.input_text** string is sent to the Server where the reader will be read for the command needed to be fulfilled in the Server side.

2.1.2 client_directory

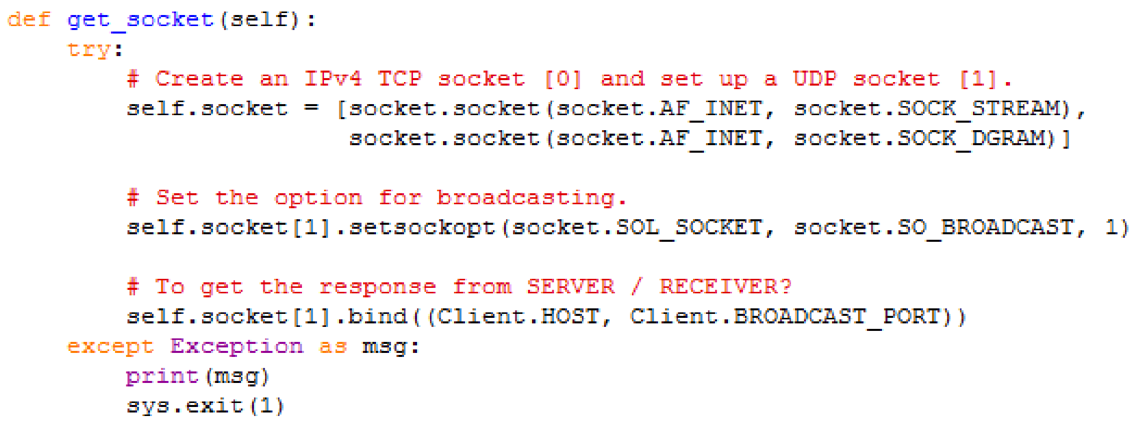


```
def client_directory(self):
    os.chdir("clientDirectory")
    self.listDir = os.listdir(os.getcwd())
    os.chdir("..")
```

Figure 2. Screenshot of **client_directory** method.

This method was added to move in and out of the client directory to access files that need to be sent and received from the Server. The operating systems library used to make use of the operating system interfaces such as changing directories and listing the contents of the directory. The **os.chdir** function takes in a string argument to move into a specified directory. The **os.listdir** function returns a list of the items in the directory and **os.getcwd** was used as the argument for **os.listdir** to get the current working directory as the argument for **os.listdir**.

2.1.3 get_socket



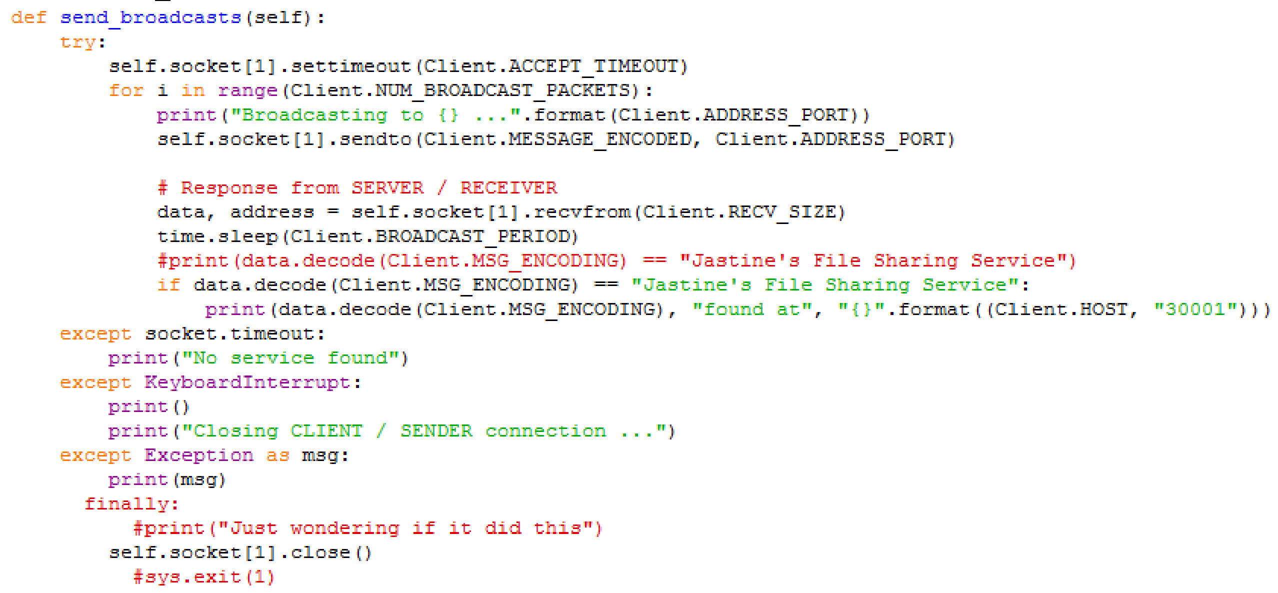
```
def get_socket(self):
    try:
        # Create an IPv4 TCP socket [0] and set up a UDP socket [1].
        self.socket = [socket.socket(socket.AF_INET, socket.SOCK_STREAM),
                       socket.socket(socket.AF_INET, socket.SOCK_DGRAM)]

        # Set the option for broadcasting.
        self.socket[1].setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)

        # To get the response from SERVER / RECEIVER?
        self.socket[1].bind((Client.HOST, Client.BROADCAST_PORT))
    except Exception as msg:
        print(msg)
        sys.exit(1)
```

Figure 3. Screenshot of **get_socket** method.

This method was modified to implement handling two sockets by creating a TCP/IP and UDP socket and storing them as a list in **self.socket** (prior to modification, **self.socket** is only a variable that stores either a TCP/IP or a UDP socket). For my program, the 0th index of **self.socket** is the TCP/IP socket while the 1st index of **self.socket** is the UDP socket. The UDP socket is initialised for broadcasting and bound to the Client host and the Service Discovery Port 30000.

2.2 METHODS ADDED FOR UDP

2.2.1 send_broadcasts



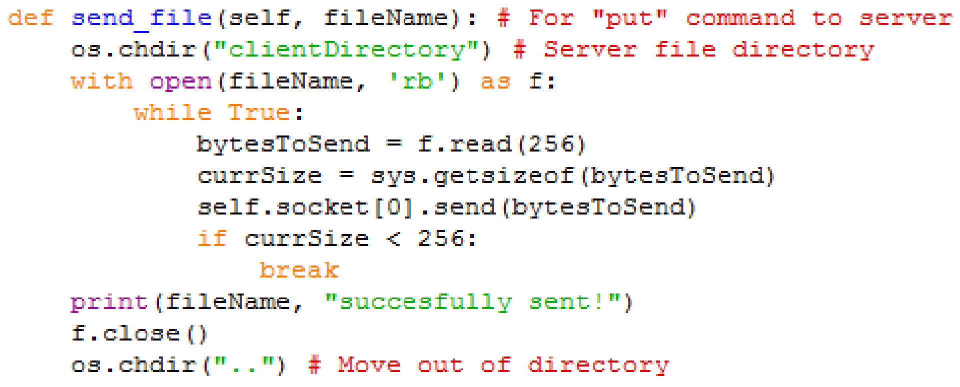
```
def send_broadcasts(self):
    try:
        self.socket[1].settimeout(Client.ACCEPT_TIMEOUT)
        for i in range(Client.NUM_BROADCAST_PACKETS):
            print("Broadcasting to {} ...".format(Client.ADDRESS_PORT))
            self.socket[1].sendto(Client.MESSAGE_ENCODED, Client.ADDRESS_PORT)

            # Response from SERVER / RECEIVER
            data, address = self.socket[1].recvfrom(Client.RECV_SIZE)
            time.sleep(Client.BROADCAST_PERIOD)
            #print(data.decode(Client.MSG_ENCODING) == "Jastine's File Sharing Service")
            if data.decode(Client.MSG_ENCODING) == "Jastine's File Sharing Service":
                print(data.decode(Client.MSG_ENCODING), "found at", "{}".format((Client.HOST, "30001")))
    except socket.timeout:
        print("No service found")
    except KeyboardInterrupt:
        print()
        print("Closing CLIENT / SENDER connection ...")
    except Exception as msg:
        print(msg)
      finally:
          #print("Just wondering if it did this")
        self.socket[1].close()
          #sys.exit(1)
```

Figure 4. Screenshot of **send_broadcasts** method.

This method was added to sent “SERVICE DISCOVERY” packets to the Server 3 times (which is what I set it to in **Client.NUM_BROADCAST_PACKETS**). A timeout is also set so that a “No service found” message will be displayed on the console once a **socket.timeout** exception is flagged after reaching a certain time set in **Client.ACCEPT_TIMEOUT**.

2.3 METHODS MODIFIED FOR TCP/IP

2.3.1 send_file

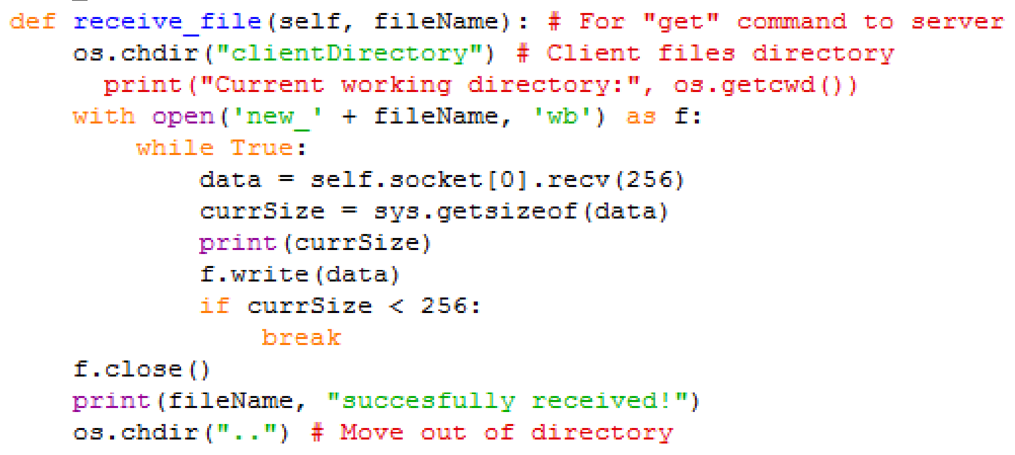


```
def send_file(self, fileName): # For "put" command to server
    os.chdir("clientDirectory") # Server file directory
    with open(fileName, 'rb') as f:
        while True:
            bytesToSend = f.read(256)
            currSize = sys.getsizeof(bytesToSend)
            self.socket[0].send(bytesToSend)
            if currSize < 256:
                break
    print(fileName, "succesfully sent!")
    f.close()
    os.chdir("..") # Move out of directory
```

Figure 5. Screenshot of **send_file** method.

This method was added to implement sending a file to the Server when the command **put** is given in the console. The **os.chrdir** function from the operating system library was used to change into the directory before sending a file. Once the file is opened, an infinite while loop is used to keep reading the file and sent through the TCP/IP socket through the File Sharing Port 30001.

2.3.2 receive_file



```
def receive_file(self, fileName): # For "get" command to server
    os.chdir("clientDirectory") # Client files directory
      print("Current working directory:", os.getcwd())
    with open('new_' + fileName, 'wb') as f:
        while True:
            data = self.socket[0].recv(256)
            currSize = sys.getsizeof(data)
            print(currSize)
            f.write(data)
            if currSize < 256:
                break
    f.close()
    print(fileName, "succesfully received!")
    os.chdir("..") # Move out of directory
```

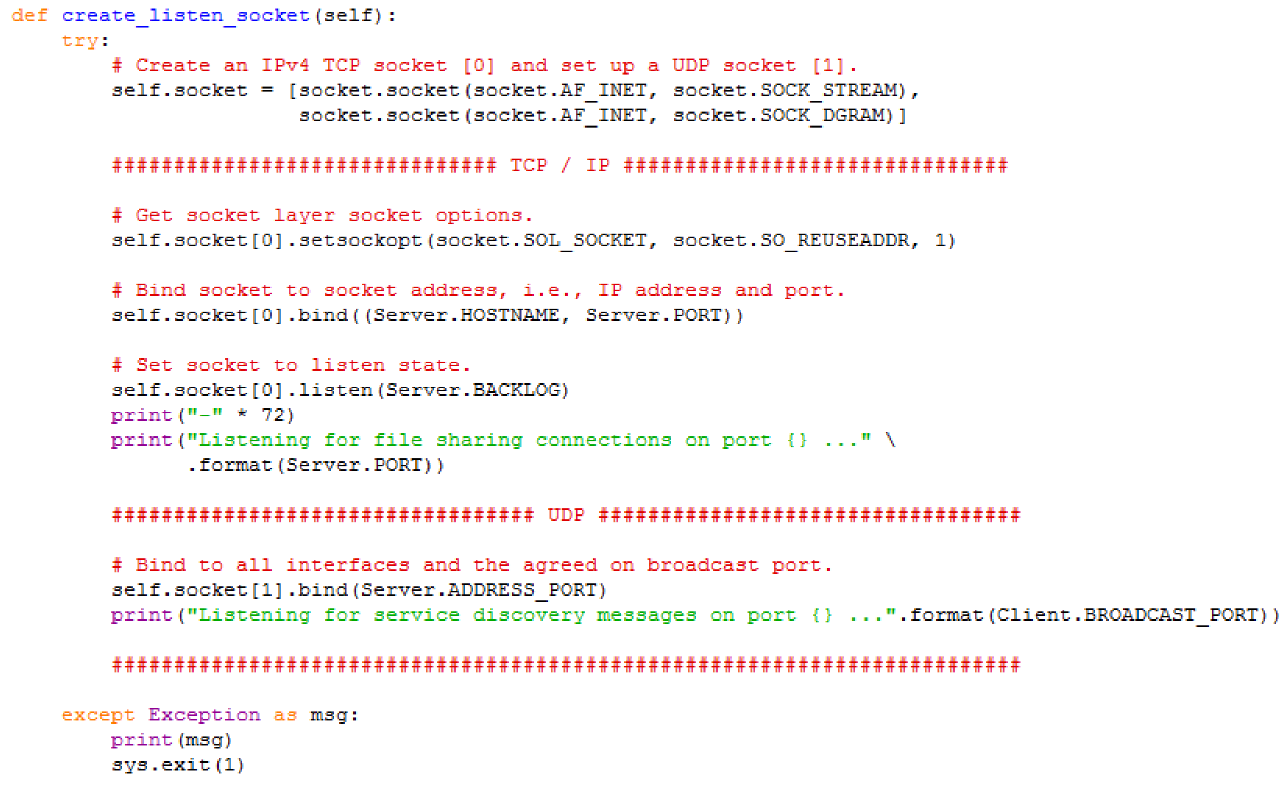
Figure 6. Screenshot of **receive_file** method.

This method was added to implement receiving a file from the Server when the command get is given in the console. Once a file is opened, **'new_'** is added in front of the file name to indicate that it is the new file received from the Server. An infinite while loop is used to keep receiving data packets through the File Sharing Port 30001. The size of every data packet received is checked using **sys.getsizeof** to satisfy the break condition because the last data packet will be smaller therefore indicating that it is the last data packet from the Server.

3 SERVER

3.1 METHODS ADDED AND MODIFIED FOR FUNCTIONALITY

3.1.1 create_listen_socket

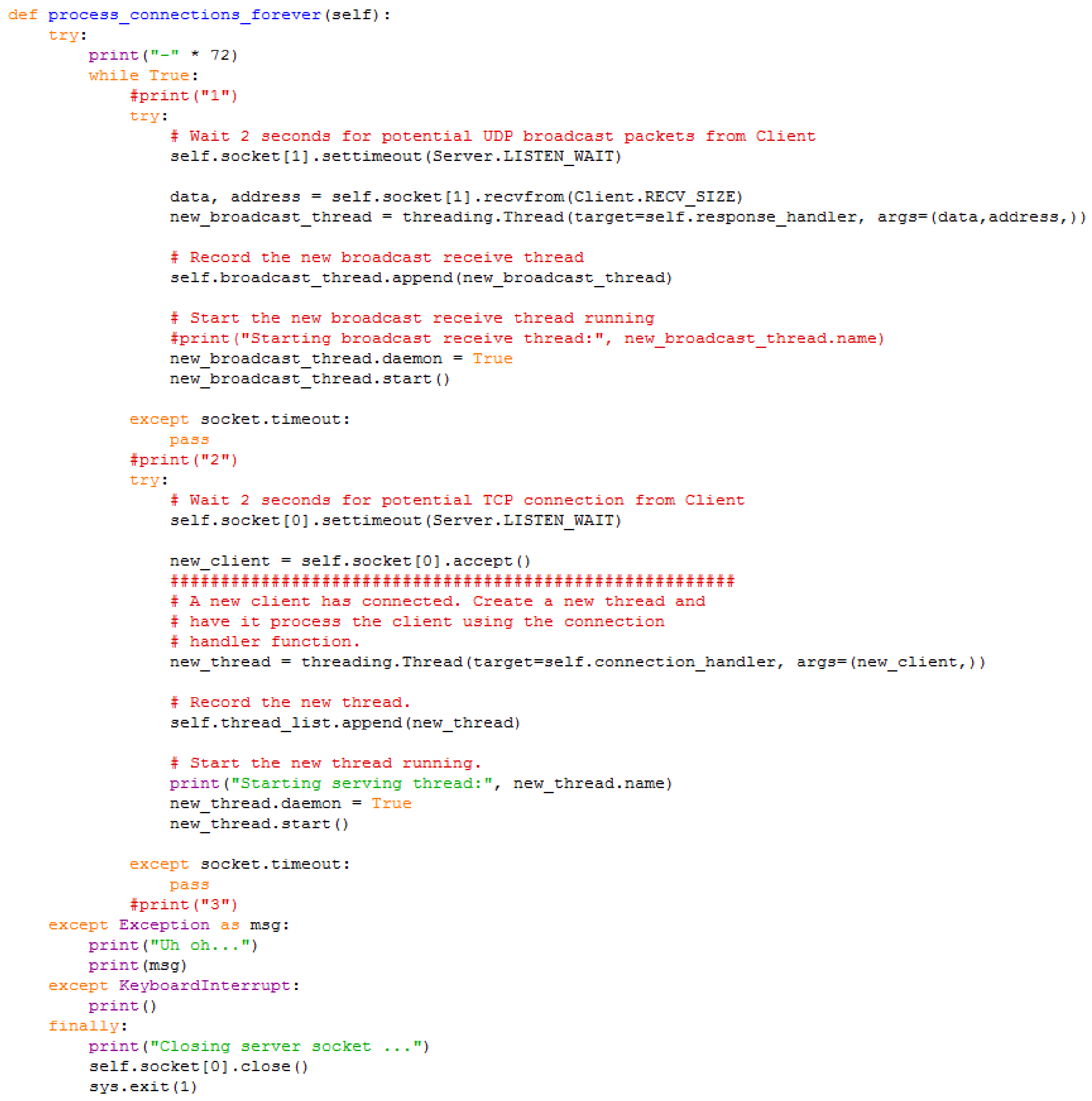


```python
def create_listen_socket(self):
    try:
        # Create an IPv4 TCP socket [0] and set up a UDP socket [1].
        self.socket = [socket.socket(socket.AF_INET, socket.SOCK_STREAM),
                       socket.socket(socket.AF_INET, socket.SOCK_DGRAM)]

        ############################### TCP / IP ###############################

        # Get socket layer socket options.
        self.socket[0].setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

        # Bind socket to socket address, i.e., IP address and port.
        self.socket[0].bind((Server.HOSTNAME, Server.PORT))

        # Set socket to listen state.
        self.socket[0].listen(Server.BACKLOG)
        print("-" * 72)
        print("Listening for file sharing connections on port {} ..." \
              .format(Server.PORT))

        ################################## UDP ##################################

        # Bind to all interfaces and the agreed on broadcast port.
        self.socket[1].bind(Server.ADDRESS_PORT)
        print("Listening for service discovery messages on port {} ...".format(Client.BROADCAST_PORT))

        #########################################################################

    except Exception as msg:
        print(msg)
        sys.exit(1)
```

Figure 7. Screenshot of **create_listen_socket** method.

This method was modified to accommodate the Server to listen to sockets by creating a TCP/IP and UDP socket and storing it in **self.socket** as a list instead of a variable. The UDP socket is bound to the host's IP and the Service Discovery Port 30000 to listen for "SERVICE DISCOVERY" broadcast packets.

3.1.2 process_connections_forever

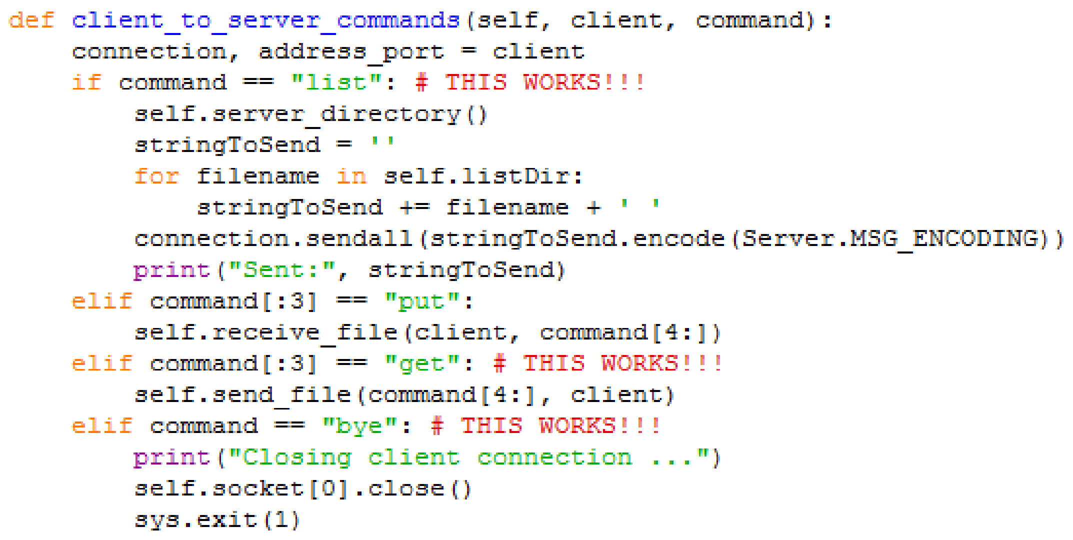


```
def process_connections_forever(self):
    try:
        print("-" * 72)
        while True:
            #print("1")
            try:
                # Wait 2 seconds for potential UDP broadcast packets from Client
                self.socket[1].settimeout(Server.LISTEN_WAIT)

                data, address = self.socket[1].recvfrom(Client.RECV_SIZE)
                new_broadcast_thread = threading.Thread(target=self.response_handler, args=(data,address,))

                # Record the new broadcast receive thread
                self.broadcast_thread.append(new_broadcast_thread)

                # Start the new broadcast receive thread running
                #print("Starting broadcast receive thread:", new_broadcast_thread.name)
                new_broadcast_thread.daemon = True
                new_broadcast_thread.start()

            except socket.timeout:
                pass
            #print("2")
            try:
                # Wait 2 seconds for potential TCP connection from Client
                self.socket[0].settimeout(Server.LISTEN_WAIT)

                new_client = self.socket[0].accept()
                ########################################################
                # A new client has connected. Create a new thread and
                # have it process the client using the connection
                # handler function.
                new_thread = threading.Thread(target=self.connection_handler, args=(new_client,))

                # Record the new thread.
                self.thread_list.append(new_thread)

                # Start the new thread running.
                print("Starting serving thread:", new_thread.name)
                new_thread.daemon = True
                new_thread.start()

            except socket.timeout:
                pass
            #print("3")
    except Exception as msg:
        print("Uh oh...")
        print(msg)
    except KeyboardInterrupt:
        print()
    finally:
        print("Closing server socket ...")
        self.socket[0].close()
        sys.exit(1)
```

Figure 8. Screenshot of **process_connections_forever** method.

This method was modified to create a new thread object for each of the UDP broadcast packets received and TCP/IP connections accepted through the Service Discovery Port. A socket timeout was also implemented on both so that it will alternate every 2 seconds to listen between ports 30000 and 30001. Once a thread is created, the responding handler methods (**response_handler** for UDP, and **connection_handler** for TCP/IP) are called.

3.1.3 client_to_server_commands

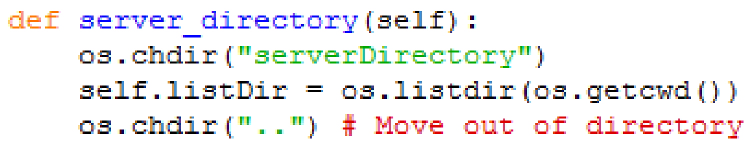


```
def client_to_server_commands(self, client, command):
    connection, address_port = client
    if command == "list": # THIS WORKS!!!
        self.server_directory()
        stringToSend = ''
        for filename in self.listDir:
            stringToSend += filename + ' '
        connection.sendall(stringToSend.encode(Server.MSG_ENCODING))
        print("Sent:", stringToSend)
    elif command[:3] == "put":
        self.receive_file(client, command[4:])
    elif command[:3] == "get": # THIS WORKS!!!
        self.send_file(command[4:], client)
    elif command == "bye": # THIS WORKS!!!
        print("Closing client connection ...")
        self.socket[0].close()
        sys.exit(1)
```

Figure 9. Screenshot of **client_to_server_commands** method.

This method was added to handle the commands received from the Client from the File Sharing Port 30001. The header of the TCP/IP packet received is checked for the appropriate commands such as **list**, **put**, **get**, and **bye**. Then based on the command, the method needed to fulfill the command is invoked.

3.1.4 server_directory



```
def server_directory(self):
    os.chdir("serverDirectory")
    self.listDir = os.listdir(os.getcwd())
    os.chdir("..") # Move out of directory
```

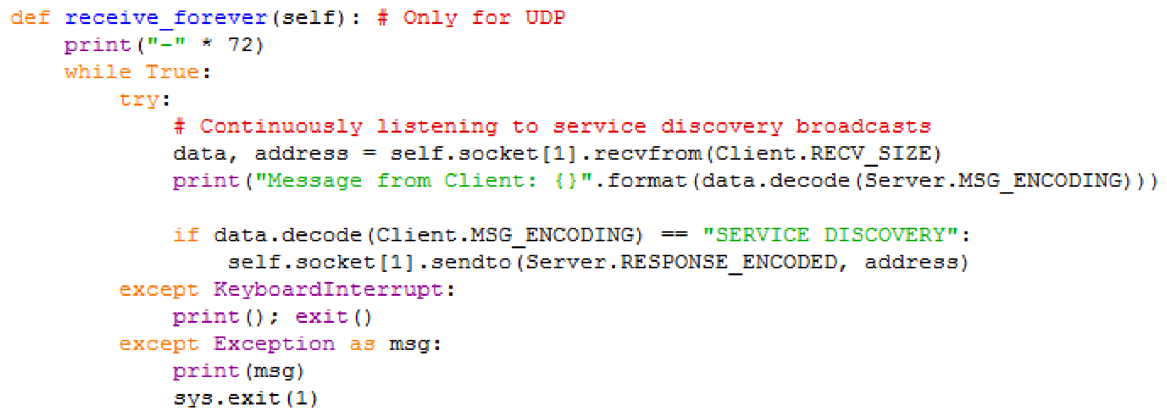
Figure 10. Screenshot of **server_directory** method.

The concept is identical to the **client_directory** method in the Client class.

This method was added to move in and out of the client directory to access files that need to be sent and received from the Client. The operating systems library used to make use of the operating system interfaces such as changing directories and listing the contents of the directory. The **os.chdir** function takes in a string argument to move into a specified directory. The **os.listdir** function returns a list of the items in the directory and **os.getcwd** was used as the argument for **os.listdir** to get the current working directory as the argument for **os.listdir**.

3.2 METHODS ADDED FOR UDP

3.2.1 receive_forever

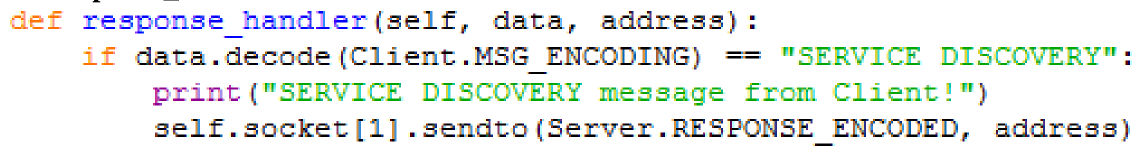


```
def receive_forever(self): # Only for UDP
    print("-" * 72)
    while True:
        try:
            # Continuously listening to service discovery broadcasts
            data, address = self.socket[1].recvfrom(Client.RECV_SIZE)
            print("Message from Client: {}".format(data.decode(Server.MSG_ENCODING)))

            if data.decode(Client.MSG_ENCODING) == "SERVICE DISCOVERY":
                self.socket[1].sendto(Server.RESPONSE_ENCODED, address)
        except KeyboardInterrupt:
            print(); exit()
        except Exception as msg:
            print(msg)
            sys.exit(1)
```

Figure 11. Screenshot of **receive_forever** method.

This method was modified by adding a conditional to check if the broadcast packet received is a “SERVICE DISCOVERY” packet to indicate that there is a Client that wants to connect to the Server for file sharing. One the condition is true, the Server sends back “Jastine’s File Sharing Service” to the Client to indicate that the Server is available.

3.2.2 response_handler



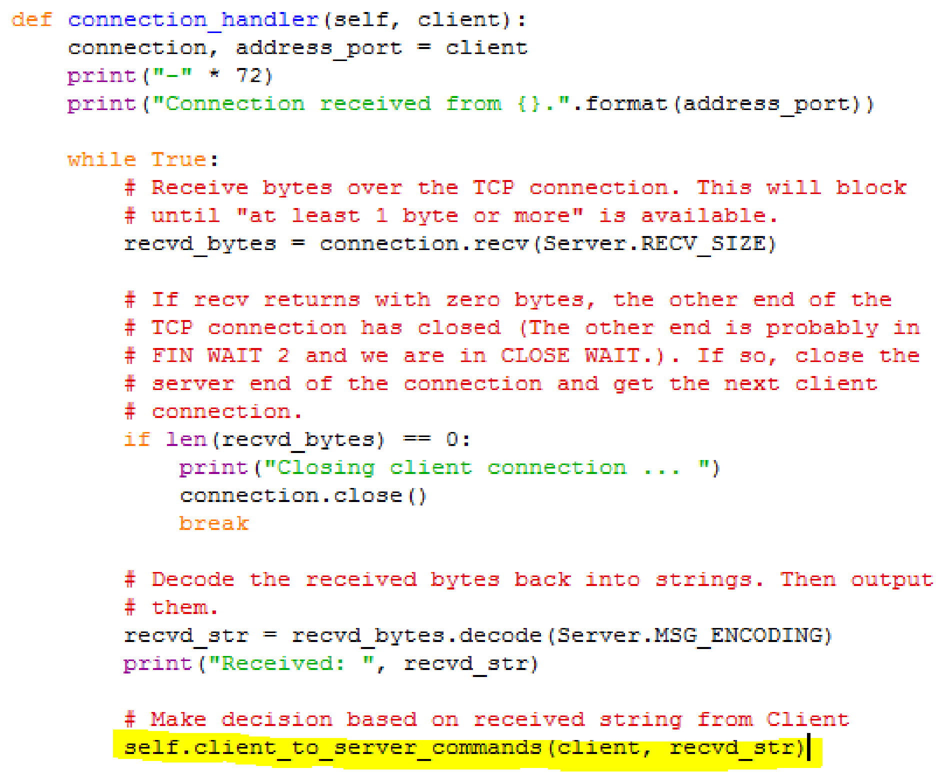
```
def response_handler(self, data, address):
    if data.decode(Client.MSG_ENCODING) == "SERVICE DISCOVERY":
        print("SERVICE DISCOVERY message from Client!")
        self.socket[1].sendto(Server.RESPONSE_ENCODED, address)
```

Figure 12. Screenshot of **response_handler** method.

This method is added to handle the new broadcast packet received thread to check if the broadcast packet received is “SERVICE DISCOVERY”. If true, it sends back the response “Jastine’s File Sharing Service” to the Client through the UDP socket on the Service Discover Port 30000.

3.3 Methods Modified for TCP/IP

3.3.1 connection_handler

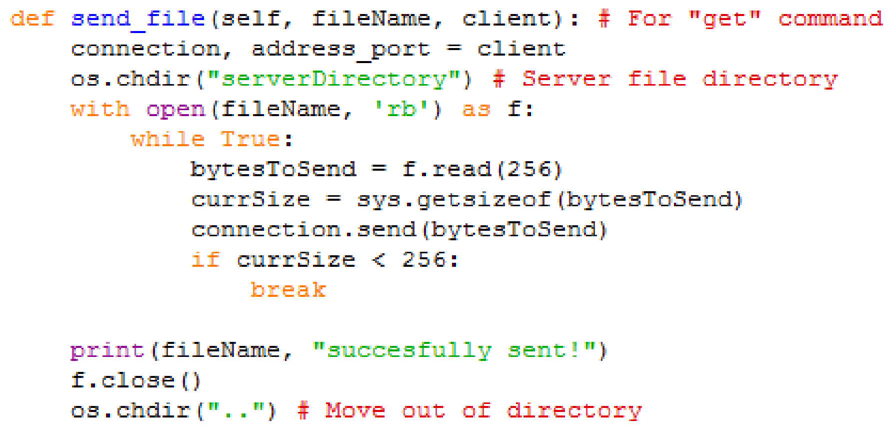


```
def connection_handler(self, client):
    connection, address_port = client
    print("-" * 72)
    print("Connection received from {}.".format(address_port))

    while True:
        # Receive bytes over the TCP connection. This will block
        # until "at least 1 byte or more" is available.
        recvd_bytes = connection.recv(Server.RECV_SIZE)

        # If recv returns with zero bytes, the other end of the
        # TCP connection has closed (The other end is probably in
        # FIN WAIT 2 and we are in CLOSE WAIT.). If so, close the
        # server end of the connection and get the next client
        # connection.
        if len(recvd_bytes) == 0:
            print("Closing client connection ... ")
            connection.close()
            break

        # Decode the received bytes back into strings. Then output
        # them.
        recvd_str = recvd_bytes.decode(Server.MSG_ENCODING)
        print("Received: ", recvd_str)

        # Make decision based on received string from Client
        self.client_to_server_commands(client, recvd_str)
```

Figure 13. Screenshot of **connection_handler** method.

This method was modified to make use of the received packets from the Client through the File Sharing Port 30001 and invoke the method that handles the commands from the client to the server. The last part of the code that echoes back the string received is eliminated and replaced with a call on the **self.client_to_server_commands** method to handle the command and filename in the string received in **recvd_bytes**.

3.3.2 send_file



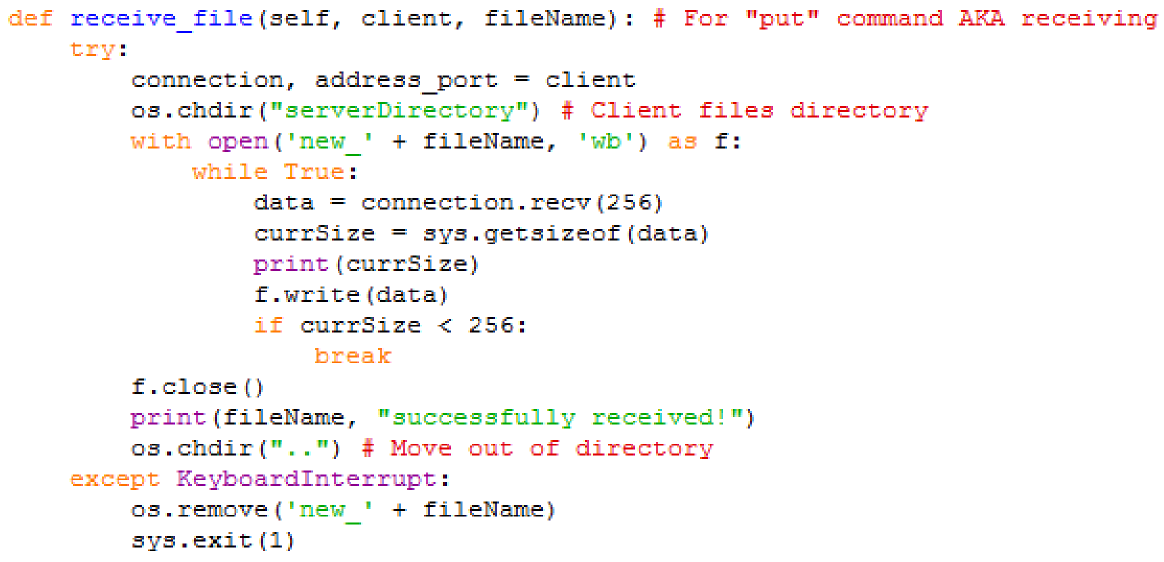
```
def send_file(self, fileName, client): # For "get" command
    connection, address_port = client
    os.chdir("serverDirectory") # Server file directory
    with open(fileName, 'rb') as f:
        while True:
            bytesToSend = f.read(256)
            currSize = sys.getsizeof(bytesToSend)
            connection.send(bytesToSend)
            if currSize < 256:
                break

    print(fileName, "succesfully sent!")
    f.close()
    os.chdir("..") # Move out of directory
```

Figure 14. Screenshot of **send_file** method.

The concept is identical to the **send_file** method in the Client class.

This method was added to implement sending a file to the Client when the command **put** is received from the Client. The **os.chrdir** function from the operating system library was used to change into the directory before sending a file. Once the file is opened, an infinite while loop is used to keep reading the file and sent through the TCP/IP socket through the File Sharing Port 30001.

3.3.3 receive_file



```
def receive_file(self, client, fileName): # For "put" command AKA receiving
    try:
        connection, address_port = client
        os.chdir("serverDirectory") # Client files directory
        with open('new_' + fileName, 'wb') as f:
            while True:
                data = connection.recv(256)
                currSize = sys.getsizeof(data)
                print(currSize)
                f.write(data)
                if currSize < 256:
                    break
        f.close()
        print(fileName, "successfully received!")
        os.chdir("..") # Move out of directory
    except KeyboardInterrupt:
        os.remove('new_' + fileName)
        sys.exit(1)
```

Figure 15. Screenshot of **receive_file** method.

The concept is identical to the **receive_file** method in the Client class.

This method was added to implement receiving a file from the Client when the command **get** is received in the header of the first packet received from the Client. Once a file is opened, **'new_'** is added in front of the file name to indicate that it is the new file received from the Client. An infinite while loop is used to keep receiving data packets through the File Sharing Port 30001. The size of every data packet received is checked using **sys.getsizeof** to satisfy the break condition because the last data packet will be smaller therefore indicating that it is the last data packet from the Server.