2022

Distributed Computing: Assignment 2

PEER-TO-PEER CHAT SYSTEM

JAMES CASSIDY 40267110

Table of Contents

Introduction

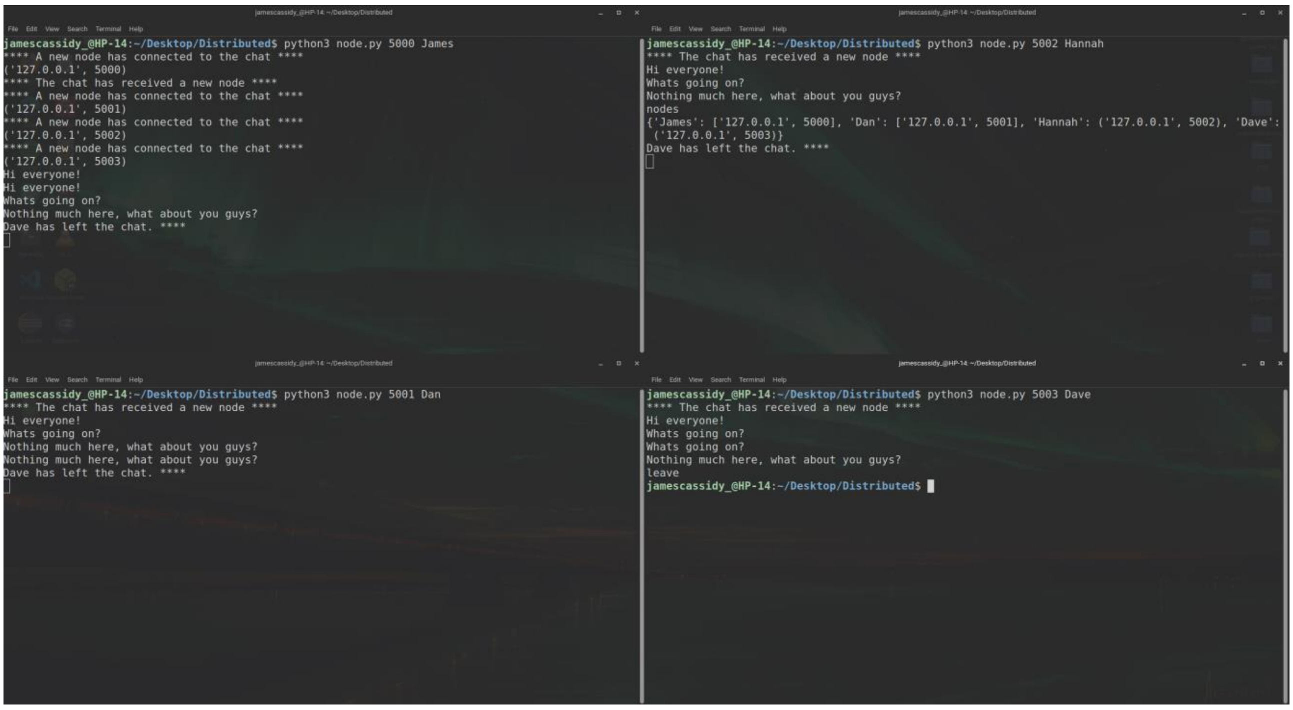
For this assignment, a distributed chat system has been developed that distributes data and processing between connected nodes of the system in a peer to peer-to-peer architecture.

My distributed chat system consists of two files and was developed using Python 3. Several modules including JSON, sockets, and threading were imported that aided in the completion of this assignment.

To start the program, a user must have a terminal window open, and their directory set to the folder in which the two files exist. From here they can start running the peer-to-peer chat. To initially execute the chat, the initial node must use the port number ‘5000’. For example, the initial node will execute this command to begin the peer-to-peer chat:

- python3 node.py 5000 James

The first argument will take the name of the node while the second argument will take the port number. Once executed, the peer-to peer chat will begin to run, allowing other nodes to connect, this time using a different name and port number. Multiple nodes can join the chat, communicate with one another, write specific commands to view details and leave gracefully without having to close the terminal itself. New nodes that join will be unable to see other node messages before they joined. Nodes will still be able to chat to one another once the initial node has left however new nodes will be unable to join.


jamescassidy_@HP-14: ~/Desktop/Distributed
File Edit View Search Terminal Help

```
jamescassidy_@HP-14:~/Desktop/Distributed$ python3 node.py 5000 James
**** A new node has connected to the chat ****
('127.0.0.1', 5000)
**** The chat has received a new node ****
**** A new node has connected to the chat ****
('127.0.0.1', 5001)
**** A new node has connected to the chat ****
('127.0.0.1', 5002)
**** A new node has connected to the chat ****
('127.0.0.1', 5003)
Hi everyone!
Hi everyone!
Whats going on?
Nothing much here, what about you guys?
Dave has left the chat. ****
```

```
jamescassidy_@HP-14:~/Desktop/Distributed$ python3 node.py 5002 Hannah
**** The chat has received a new node ****
Hi everyone!
Whats going on?
Nothing much here, what about you guys?
nodes
{'James': ['127.0.0.1', 5000], 'Dan': ['127.0.0.1', 5001], 'Hannah': ('127.0.0.1', 5002), 'Dave':
 ('127.0.0.1', 5003)}
Dave has left the chat. ****
```

```
jamescassidy_@HP-14:~/Desktop/Distributed$ python3 node.py 5001 Dan
**** The chat has received a new node ****
Hi everyone!
Whats going on?
Nothing much here, what about you guys?
Nothing much here, what about you guys?
Dave has left the chat. ****
```

```
jamescassidy_@HP-14:~/Desktop/Distributed$ python3 node.py 5003 Dave
**** The chat has received a new node ****
Hi everyone!
Whats going on?
Whats going on?
Nothing much here, what about you guys?
leave
jamescassidy_@HP-14:~/Desktop/Distributed$
```

Two files were developed for this peer-to-peer chat system, including
- messagingFunctions.py
- node.py

The messagingFucntions.py file uses several methods that are executed in the main node file that will be executed by the user.

Messaging Functions File

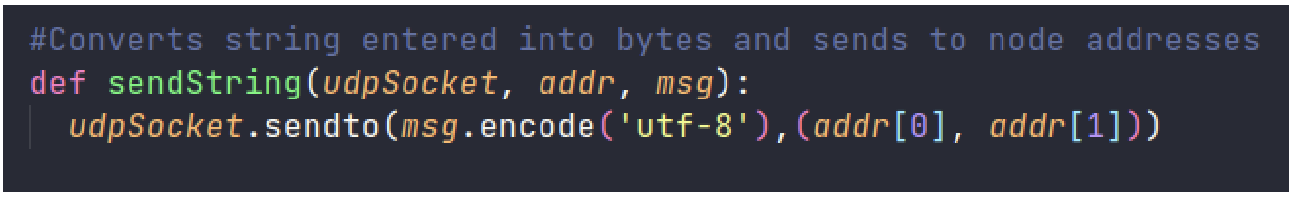
In this python file, four functions have been defined that will be used in the main node file. They include:

- sendString
- receiveBytes
- convertToJSON
- jsonBroadcast

The JSON module has been imported into this file to allow for the encoding and decoding of JSON data that will be handled throughout the peer-to-peer chat.

sendString()

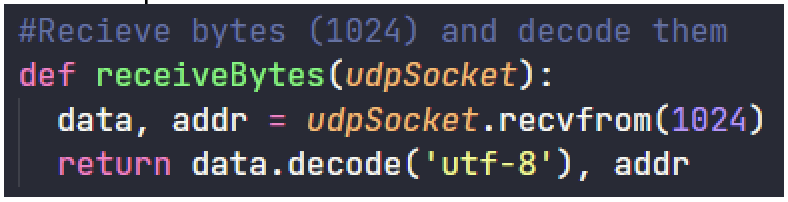
This function takes the message a node has entered and converts that into bytes. It takes the parameters of the UDP Socket, the address and port number of the node, and the message that has been sent.



```python
#Converts string entered into bytes and sends to node addresses
def sendString(udpSocket, addr, msg):
  udpSocket.sendto(msg.encode('utf-8'),(addr[0], addr[1]))
```

receiveBytes()

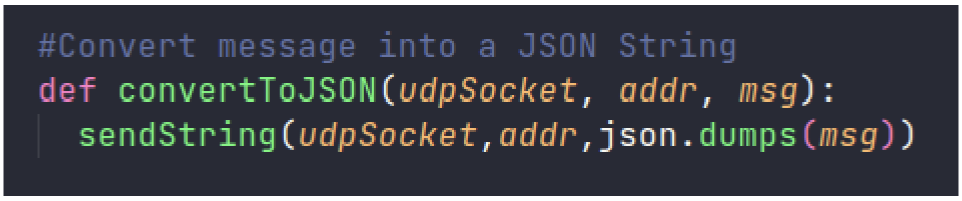
This function receives the 1024 bytes from the socket and will decode them. This function takes the parameters of the UDP Socket.



```python
#Recieve bytes (1024) and decode them
def receiveBytes(udpSocket):
  data, addr = udpSocket.recvfrom(1024)
  return data.decode('utf-8'), addr
```

convertToJSON()

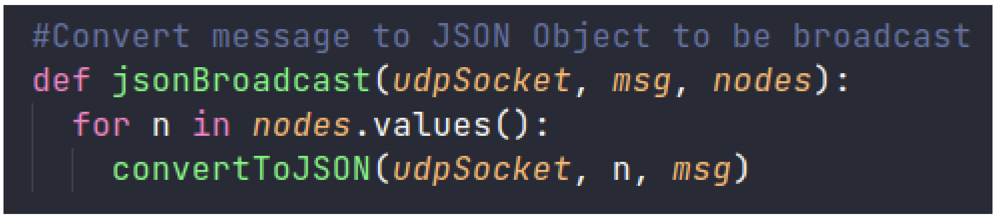
This function takes the message received by the sendString function and convert that message to a JSON string. This takes the same parameters as the sendString function including the UDP Socket, address of the node and message received. The dumps function will serialize the message to a JSON formatted string.



```python
#Convert message into a JSON String
def convertToJSON(udpSocket, addr, msg):
  sendString(udpSocket,addr,json.dumps(msg))
```

jsonBroadcast()

This function broadcasts the converted JSON string and broadcasts this to each node that exists in the chat. This takes the parameters of the UDP Socket, message received and the nodes in the chat.



```python
#Convert message to JSON Object to be broadcast
def jsonBroadcast(udpSocket, msg, nodes):
    for n in nodes.values():
        convertToJSON(udpSocket, n, msg)
```

Node File

This is the file that will be executed by each node that wishes to connect to other nodes. This file has three specific functions that handle the initial setup of the first node, sending chats to other nodes and receiving chats from other nodes. These include:

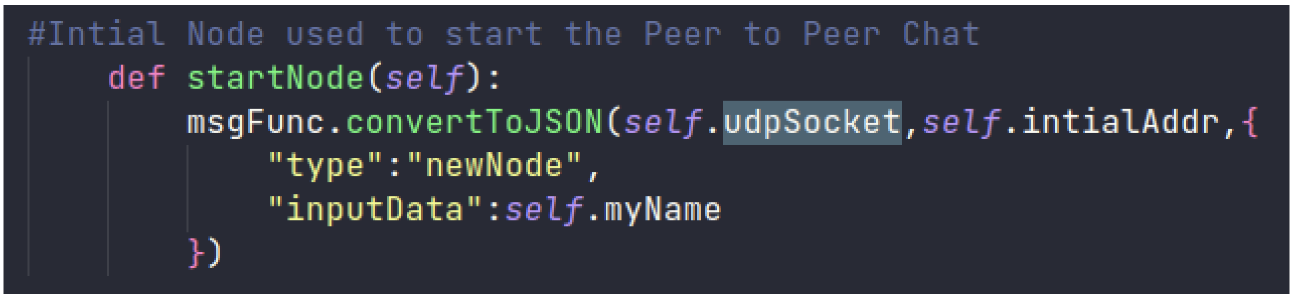
- startNode
- receiveChat
- sendChat

Several other modules have also been imported into the main node file including JSON again. These new modules will provide a BSD socket interface, threading, system arguments, and time related functions.

Several variables have been defined within the Node class. The intialAddr will hold the IP address and port number for the first initial node. Nodes is defined as an empty array that will hold all nodes that are currently in the chat. The myName variable holds the string for the name the node assigns itself when running the node file and udpSocket is an array that will hold the socket information of the node.

startNode()

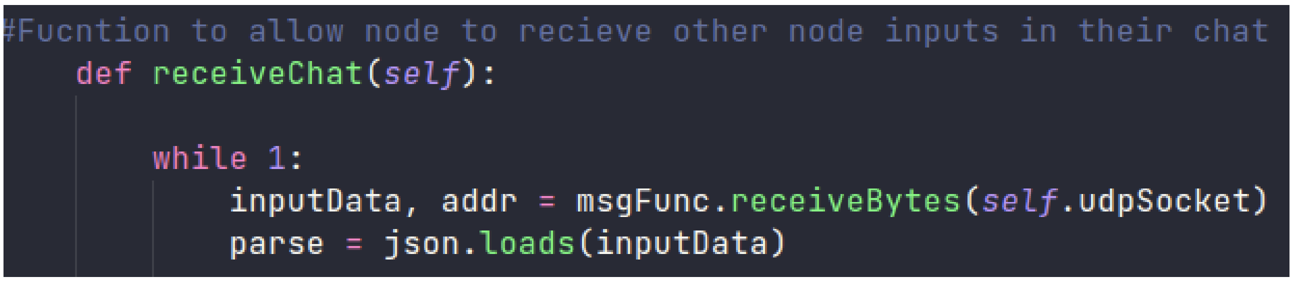
This function is used for the first node that starts the peer-to-peer chat. It uses the convertToJSON function defined in the messagingFunctions to take the UDP socket binded to the IP Address and port number (127.0.0.1 and 5000). The name of the node will be converted to JSON format.



```python
#Intial Node used to start the Peer to Peer Chat
    def startNode(self):
        msgFunc.convertToJSON(self.udpSocket,self.intialAddr,{
            "type":"newNode",
            "inputData":self.myName
        })
```

receiveChats()

This function utilises several if statements for several different inputs a node might receive. While the peer-to-peer chat is active this function will keep running. The data received and the address of the node will be utilised by the receiveBtyes function defined in messagingFunctions. Parse will also take the data inputted by a node and load that as a JSON object to be able to be read by the several JSON functions created in the previous file.

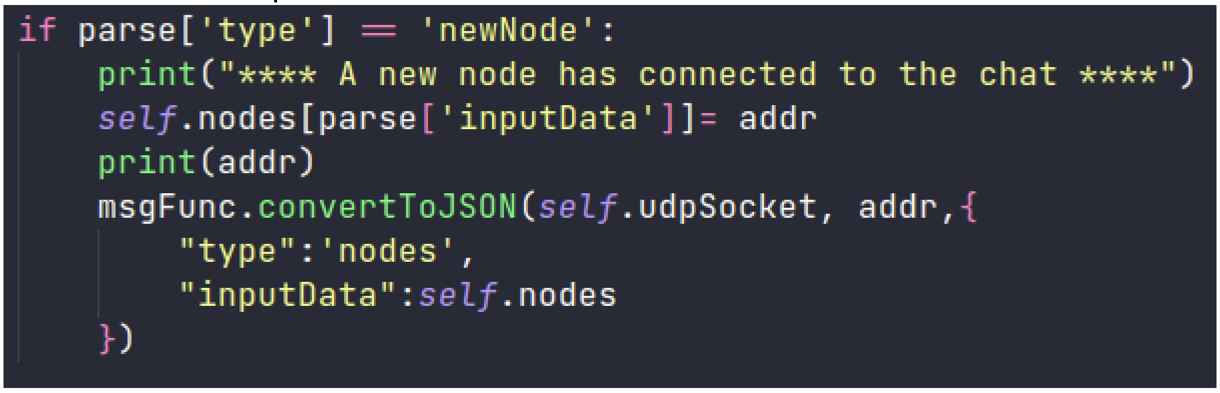


```
#Fucntion to allow node to recieve other node inputs in their chat
    def receiveChat(self):

        while 1:
            inputData, addr = msgFunc.receiveBytes(self.udpSocket)
            parse = json.loads(inputData)
```

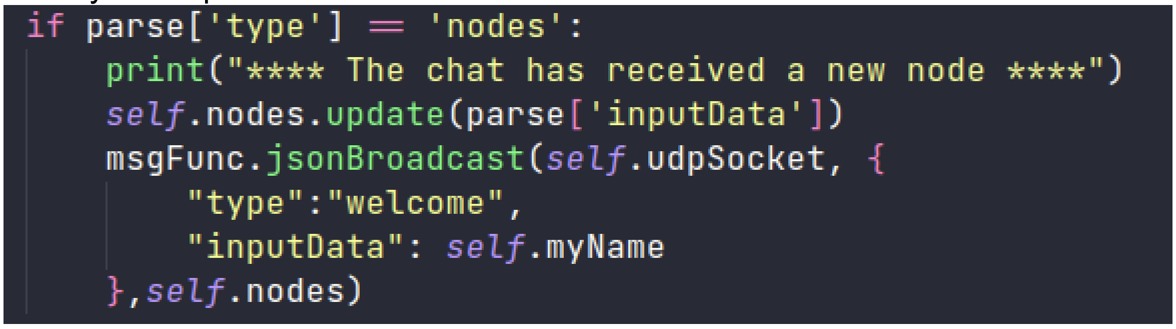
Five if statements exist within this function.

The first if statement will execute when a new node is connected to the chat network. The initial node that started the chat will be the only node to receive this message alongside the IP address and port number of the new node connected.



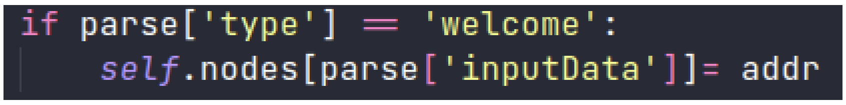
```
if parse['type'] == 'newNode':
    print("**** A new node has connected to the chat ****")
    self.nodes[parse['inputData']]= addr
    print(addr)
    msgFunc.convertToJSON(self.udpSocket, addr,{
        "type":'nodes',
        "inputData":self.nodes
    })
```

The next if statement will alert all nodes, including the initial node that started the chat has received a new node. The jsonBroadcast function used in this statement is responsible for broadcasting this to all nodes. It also parses in “type”:”welcome”, in JSON format that directly correspond to the next if statement.



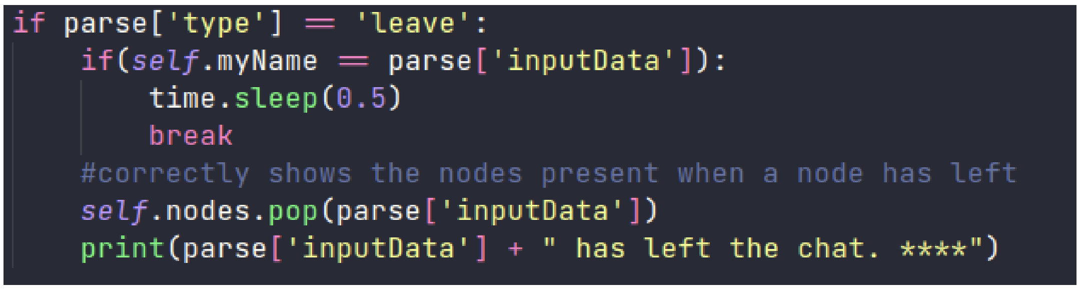
```
if parse['type'] == 'nodes':
    print("**** The chat has received a new node ****")
    self.nodes.update(parse['inputData'])
    msgFunc.jsonBroadcast(self.udpSocket, {
        "type":"welcome",
        "inputData": self.myName
    },self.nodes)
```

The if statement that follows states that if the type parsed is equal to ‘welcome’, the address of that new node will be parsed in the nodes array defined in the Nodes class.



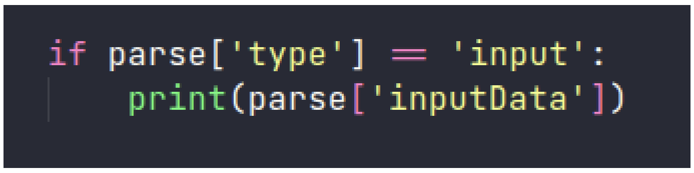
```
if parse['type'] == 'welcome':
    self.nodes[parse['inputData']]= addr
```

The next if statement utilised will allow a node to leave the chat by typing ‘leave’. It will wait for 0.5 seconds before finally allowing the node to gracefully exit the chat. The pop function will remove that node including their name, IP address and port number from the nodes array before finally sending a message to all other nodes on the chat that the name of the Node ‘has left the chat ****’



```
if parse['type'] == 'leave':
    if(self.myName == parse['inputData']):
        time.sleep(0.5)
        break
    #correctly shows the nodes present when a node has left
    self.nodes.pop(parse['inputData'])
    print(parse['inputData'] + " has left the chat. ****")
```

The final if statement states that if the input type parsed in is equal to ‘input’, the input from other nodes will be printed onto the current and all other active nodes in the terminal.

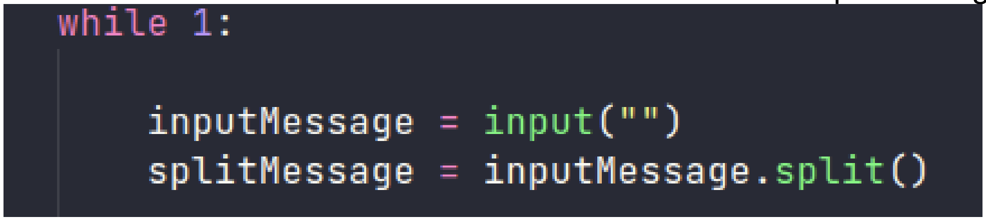


```
if parse['type'] == 'input':
    print(parse['inputData'])
```

sendChat()

The purpose of this function in the node file is to allow a node to send files to other nodes that are currently available in the peer-to-peer chat. Only while the peer-to-peer functionality has been achieved will this function work

Two variables have been defined in the while statement: inputMessage and splitMessage.



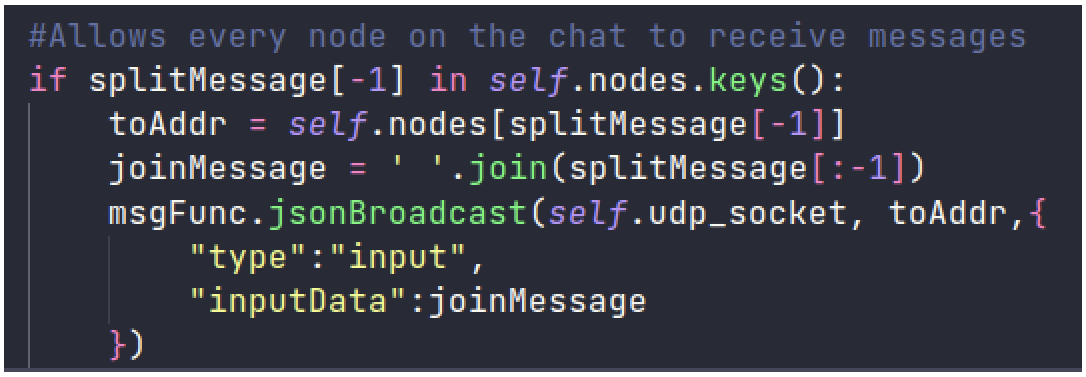
```
while 1:

    inputMessage = input("")
    splitMessage = inputMessage.split()
```

InputMessage is an empty string variable. This is used when the node file has been run in the terminal. SplitMessage will take the input string and split that into a list that will be used for the first if statement in the chat.

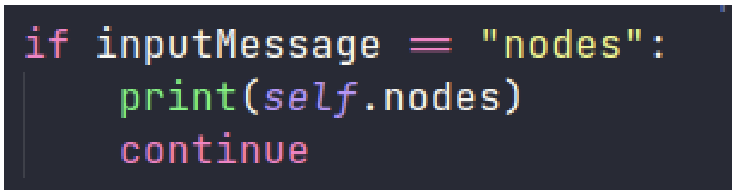
Like the previous receiveChat function, it also contains several if statements that determine what happens when a node types in random or specific text.

The first if statement in this function syncs all nodes currently in the chat and allows for all nodes to receive messages in sync. The joinMessage variable will join the split message separated by a ‘ ’ (space) to resemble the message again and broadcast that to other users in the peer-to-peer chat. Without this if statement, new nodes joining will not receive messages from the nodes that joined before it.

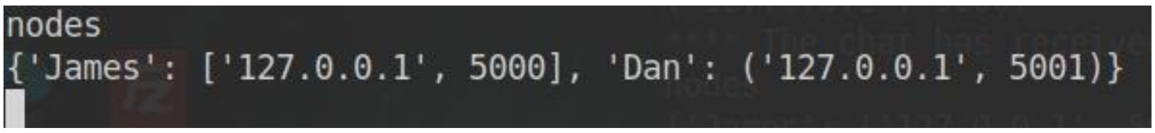


```
#Allows every node on the chat to receive messages
if splitMessage[-1] in self.nodes.keys():
    toAddr = self.nodes[splitMessage[-1]]
    joinMessage = ' '.join(splitMessage[:-1])
    msgFunc.jsonBroadcast(self.udp_socket, toAddr,{
        "type":"input",
        "inputData":joinMessage
    })
```

The following if statement states that if the inputMessage from the user is equal to ‘nodes’, each user will be displayed alongside their name, IP Address and Port Number. This will only be displayed in the node’s terminal that wished to request it. For example, if there are only two nodes present in the entire chat, the name, IP address and port number will be displayed:



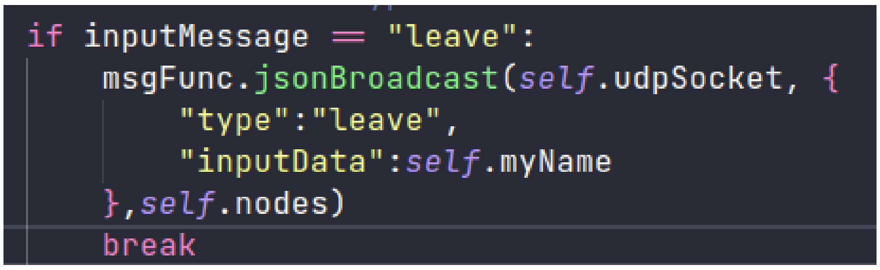
```
if inputMessage == "nodes":
    print(self.nodes)
    continue
```



```
nodes
{'James': ['127.0.0.1', 5000], 'Dan': ('127.0.0.1', 5001)}
```

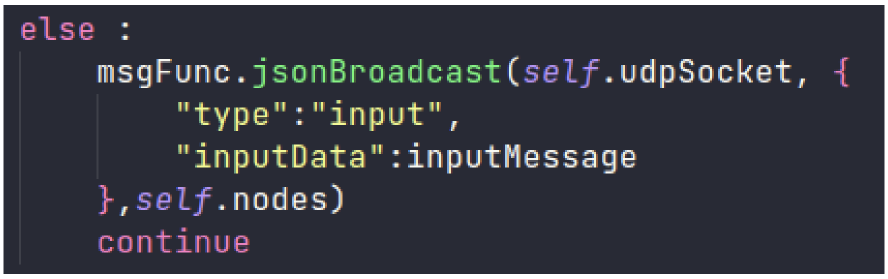
The continue will allow sendChat function to keep running while waiting for other input from the current node or others in the peer-to-peer chat.

Next, if the user wishes to leave the chat, they simply input the message ‘leave’ and they will be gracefully removed from the program. All other users on the peer-to-peer chat will receive the message of the Name of the node ‘has left the chat ****’. The break statement will terminate the loop allowing the user out of the node file.



```
if inputMessage == "leave":
    msgFunc.jsonBroadcast(self.udpSocket, {
        "type":"leave",
        "inputData":self.myName
    },self.nodes)
    break
```

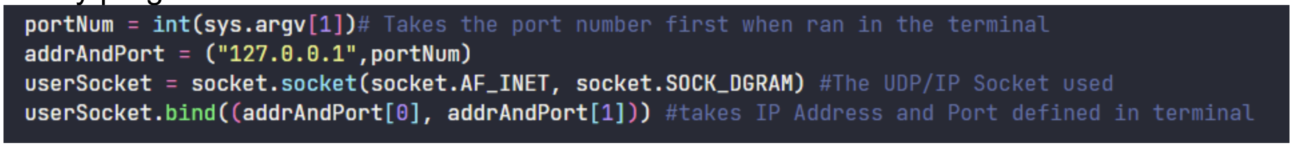
Finally, an else statement is used for any other input the node may have, it will be sent to other users on the peer-to-peer chat. The continue will allow sendChat function to keep running while waiting for other input from the current node or others in the peer-to-peer chat.



```
else :
    msgFunc.jsonBroadcast(self.udpSocket, {
        "type":"input",
        "inputData":inputMessage
    },self.nodes)
    continue
```

Main Method

With all functions defined, the main method must be programmed. This is the starting point of any program.

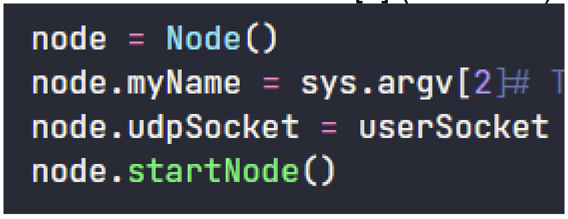


```
portNum = int(sys.argv[1])# Takes the port number first when ran in the terminal
addrAndPort = ("127.0.0.1",portNum)
userSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #The UDP/IP Socket used
userSocket.bind((addrAndPort[0], addrAndPort[1])) #takes IP Address and Port defined in terminal
```

At the beginning of the method, the port number is defined at the by sys.argv[1]. This means this will take the first argument from the user when they run the file. The addrAndPort variable has been defined with a predefined IP Address and the port number the user enters when starting the program.

Next, the udpSocket equals socket.socket(socket.AF_INET, socket.SOCK_DGRAM). This is taken from the socket module that has been imported. AF_INET states the address family that the socket has been desigintated with. In this case it will be working with an IPv4 address (127.0.0.1). SOCK_DGRAM creates a socket that will use datagrams (UDP). SOCK_STREAM will allow me to use byte stream instead (TCP).

Next the bind method will assign an IP address and a port number to the socket instance. In this case addrAndPort[0] (127.0.0.1) and addrAndPort[1] (portNum).



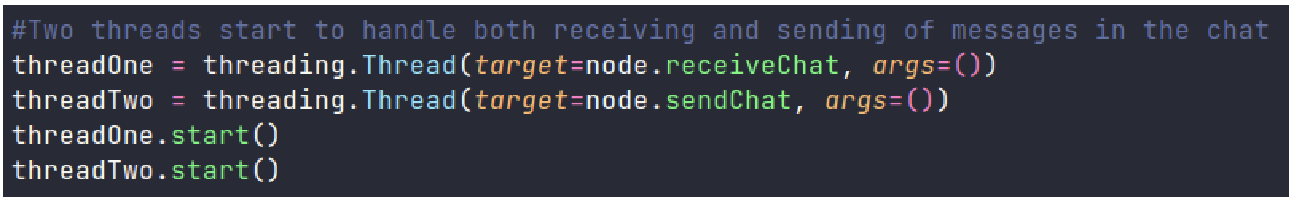
```
node = Node()
node.myName = sys.argv[2]# T
node.udpSocket = userSocket
node.startNode()
```

In the main function, node will call the class ‘Node’. This will allow the next three lines to utilise functions created in that class. The variable myName that was defined in the ‘Node’ class will equal the second system argument from the terminal.

Next, the userSocket defined in the main method will equal the udpSocket defined in the 'Node' class, allowing for other nodes to enter their name and port number. The following line, node.startNode() will take the initial node that executes the peer-to-peer chat. For example, to start the program will the terminal is active in the directory of the files, they must type:

- python3 node.py James 5000

This will successfully start the peer-to-peer program and allow other nodes to join in a similar way. Other nodes will be unable to use port number 5000, however.



```
#Two threads start to handle both receiving and sending of messages in the chat
threadOne = threading.Thread(target=node.receiveChat, args=())
threadTwo = threading.Thread(target=node.sendChat, args=())
threadOne.start()
threadTwo.start()
```

The final four lines of the program will utilise the threading module imported. Threading is appropriate in this case as it will allow the node python script to run multiple I/O bound tasks simultaneously. In this case, it will allow the script to run both the recieveChat function and sendChat function on threadOne and threadTwo respectively.