Individual Portfolio Assignment 1 Socket Bots

Piotr Pajchel

ID - s338864

DATA2410

Contents

Introduction

This is a brief documentation of *Individual Portfolio Assignment 1 Socket Bots*. The assignment consists of three elements. A Bot-class, client-program and server-program. The TCP protocol is used as the socket protocol for communicating between server and client. As shown, in Figure 1 from Kurose and Ross (2022, p. 192). In addition to the TCP protocol, a simple protocol has been added in the application layer to handle communication in the context of a chatroom with automated bots and human users. The code base for this assignment build on code examples given as part of OsloMet lectures DATA2410 spring 2022 by Aws Naser Jaber Alzarqawee on the topic of socket programming in Python. As well as NeuralNine's guide (*Simple TCP Chat Room in Python*, 2020). Both code sources have been rewritten to meet the given requirements of the assignment.

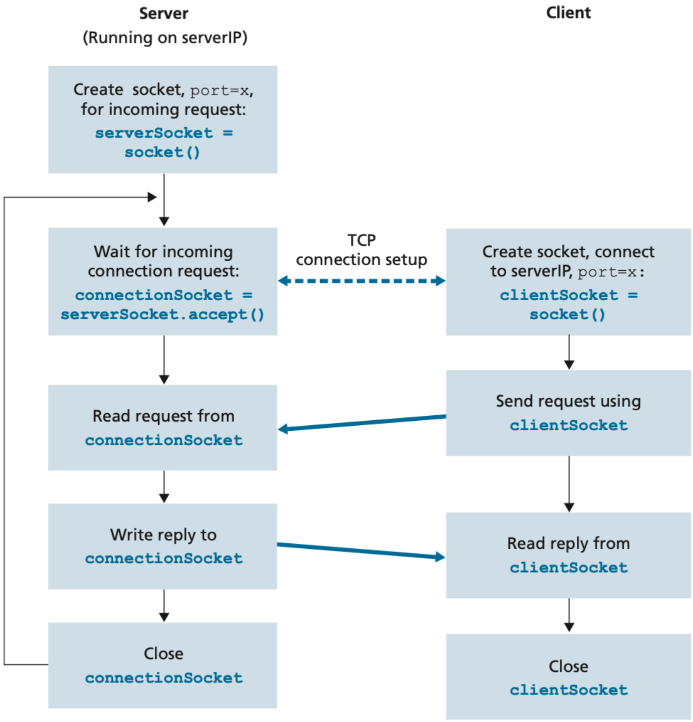

Server
(Running on serverIP)
Client
Create socket, port=x, for incoming request:
serverSocket = socket()
Wait for incoming connection request:
connectionSocket = serverSocket.accept()
TCP connection setup
Create socket, connect to serverIP, port=x:
clientSocket = socket()
Read request from connectionSocket
Send request using clientSocket
Write reply to connectionSocket
Read reply from clientSocket
Close connectionSocket
Close clientSocket

Figure 1

The client-server application using TCP

Bots

The Bot.py class consist of the functions that are needed to run the chat Bots. The Bots have different names and different personalities. The following words; play, eat, cry, sleep, fight will trigger a response from the bots. The *def find_word(string):* function is scanning sentences other users are typing in the chat. When a valid word is found, it becomes input (a) for function *def response(bot_type, a, b=None):* that generates a written bot response. To avoid an feedback loop of bots responding to other bots, the function *def name_check(string):* ignores responses written by bots. The Bots logic are used as given in the example code form lecture, however the colcatinating method has been rewritten to use f-string. The f string method is more readable, concise, and less prone to error than older solutions (Jablonski, 2021). For further documentation of the Bot.py class see p. 9 in Appendix section.

Client

The program client in my solution has two modes of operation. 1) As a chat client for human users, allowing for communication with other users and chat bots. 2) Client program as a Bot responding to trigger words picked up from chat dialog. When starting a client 4 command lines arguments are required; IP address of server, Port number, Mode and Name. Example of command for starting a user client:

```
python client.py 127.0.0.1 55556 user Neo
```

The client uses Python argparse module together with logical value checks to ensure that the client program has valid arguments before attempting to connect to server. This form for input check eliminates some "Broken Pipe errors" on the server and also makes the client program essayer to use correctly. The Client / Server programs share a simple protocol in addition to the TCP protocol. Each user sent message starts with a name with a ":" appended to the name followed by a message. Example of user message in chat:

Neo: Hello chat! Should I really take the red pill?

This name tag is generated from the client and used for routing and checking of message communication between server and clients. An example of this is the Bot function *def name_check(string):* that uses the name tag in the message to avoid feedback response between Bots. System messages form server to client are single string commands with capital letter without the appended ":". System messages are part of this simple protocol and governs situation and checks between client and server of systemic nature, like asking for an nickname with the 'NICK' command and replying with 'NICK_INVALID' if the nickname name is already registered on the server. The Server section of this project documentation further illustrates the relation ship between client and server and how this protocol scheme resolves different situations.

The Bot and the User code shares many similarity when it comes to establishing a connection with the server. Once a connection is established they operate differently. In bot mode the client is running a single thread handling both input and output in a self contained system. User mode has two threads. One thread for handling input from the user when they type a message on the keyboard. Another thread is listening for incoming messages and printing them in the terminal as they come inn. For further documentation of client.py see source code on p.11 in Appendix section.

Server

The main purpose of the server in this assignment is to distribute messages among the connected clients. The server i the central point in the chat server topology. In order to do this a client has to go trough system checks provided by the server protocol before they are allowed to connected to the server. For full protocol overview see Figure 2 in Appendix on p 17. Once a client is connected a separate thread is created by the *def receive():* function for that client. A multi threaded setup allows each client to send messages to the server at any point in time. The client connection is stored in the *clients = [] list* with a corresponding index list *nicknames = []* containing the name of the client. These two list serve as a database and are used by the server when it looks to disconnect a client or want to check if a given nickname is unique.

Once a client thread is running the *def handel(client):* function listens for messages from connected client. A incoming message containing data will be put in the broadcast queue for further distribution. Empty messages with zero byte indicates that the client have disconnected. If a empty message is detected the *def handel(client):* fuction will removed the client from server. An client is allowed to be inactive, but the *def set_keepalive* function will check if the socket connection is alive and after 5 failed pings attempts the client will be removed form server

Messages from connected clients are put in a thread safe queue by the *def broadcast_q():* function. The queue functionality of the server provides a unified model view of the message order. Without a queue in a multi threader client system the chat messages order displayed per client would not have had the same chronology. The *def broadcast_q():* also filters out the sender of a message so the message is not sent back to the sender only to other clients connected to the server. This operation is done by identifying sender by name tag (part of the "name:" protocol) and generating a new *send_list* used for sending a specific message. A new *send_list* is generated for each message sent from the server. If a name tag is not present the server sends out the message to all clients as system message. A typical system massage from server would be: "A user have disconnected".

While the server is running, a basic command line interface (CLI) is available for a system administrator. Typing the command *list* will list the name of all connected clients. If a user or bot have misbehaved they can be removed from the server by typing *kick* followed by a prompt asking for the name of the bot or user to be kicked. For further documentation of the server.py see p. 14 in the Appendix

Conclusion

During the process of programming this assignment one issue proved very difficult to resolve. When a client in user mode receives messages from the server, the print() function doesn't parse the line breaks correctly, resulting in irregular line break output in the client terminal window. To mitigate this problem a temporary fix was implemented in the *def broadcast_q():* at line 70 and 77 in server.py. By using time.sleep(0.001), the rate of message sent from server is staggered so the client print() can keep up without skipping line breaks. However this is by no means a good solution. Because it would not scale well with more users, since the message output to would be delayed too much as a result of the amount of users connected to the server. Also it is probably mask an underling problem in the client code that concerns how the input and output user threads are calling the print() function. I've tried thread locking the print function and also looked into using sys.stdout instead of print() but none of these attempts resolved the problem.

Another problem that surprised me during this project was the "[Errno 32] Broken pipe" error and how easily a client can crash the server. To my knowledge all of the critical server crash errors have been resolved, but more testing is needed to confirm this.

If we look at topics, beyond the scope of this assignment, it would be interesting to test out the code base in a live environment. To see how it performs with a large user base (granted that the print issue is resolved) and also see how stable the server code is overtime. I would also like to develop a proper interface for the application and implement encryption for the messages sent from user to user.

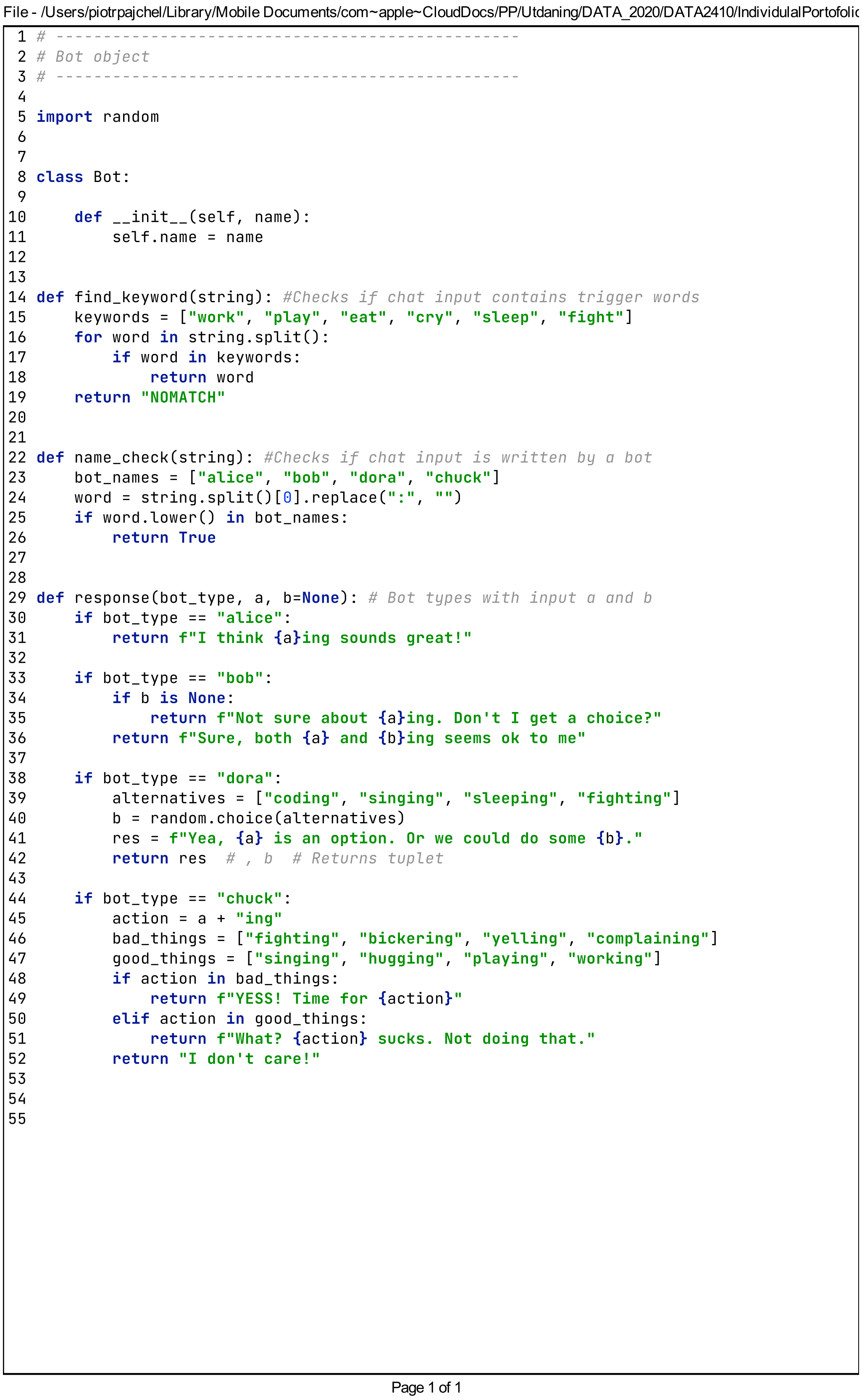
References

Jablonski, J. (2021). Python 3's f-strings: An improved string formatting syntax (guide). https://realpython.com/python-f-strings/

Kurose, J. F., & Ross, K. W. (2022). 2.7.2 socket programing with tcp. *Computer networking: A top-down approach* (Eight Edition, pp. 192–193). Pearson.

Simple tcp chat room in python. (2020). YouTube. https://www.youtube.com/watch?v=3UOyky9sEQY&amp;t=1188s

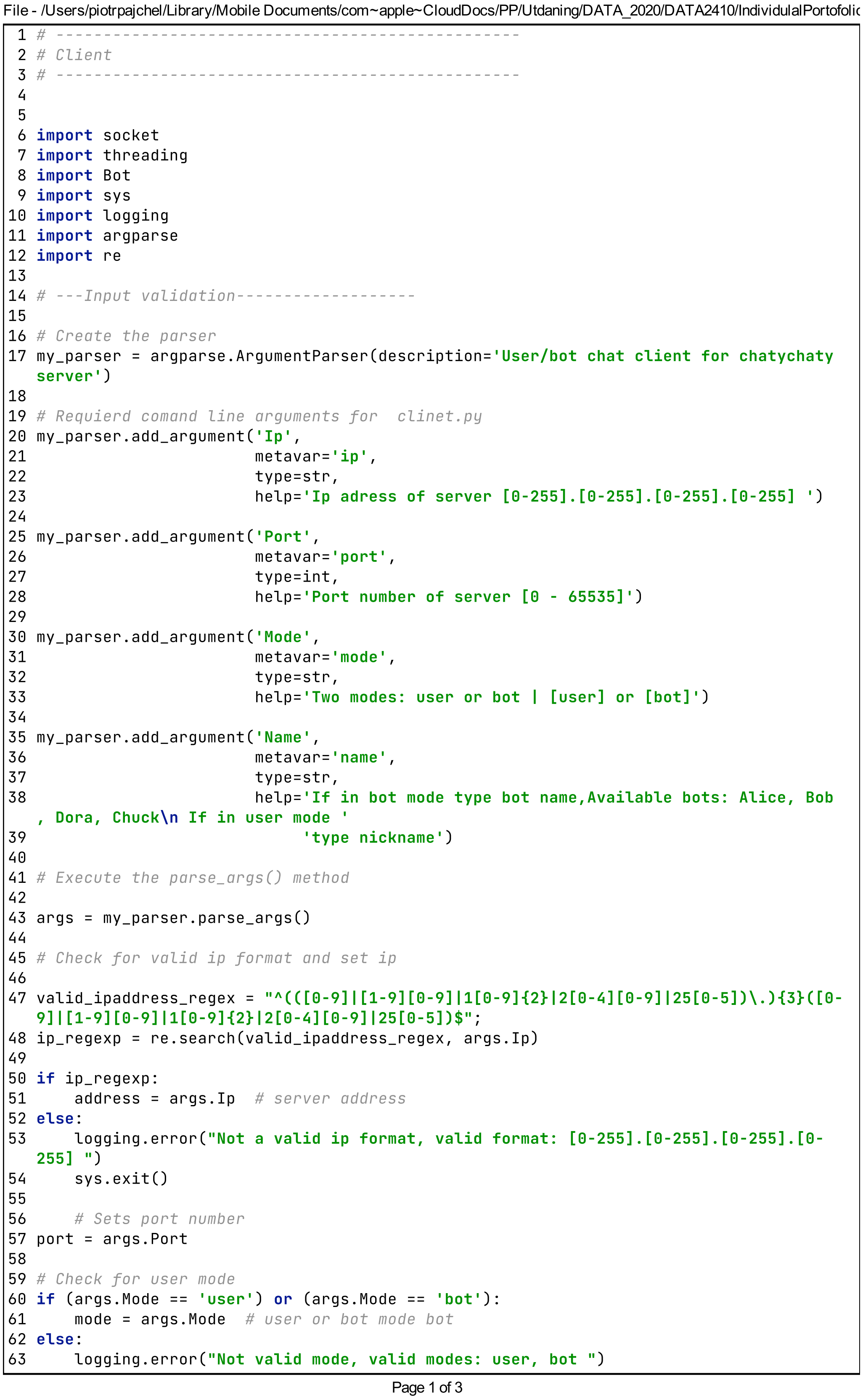
Appendix



File - /Users/piotrpajchel/Library/Mobile Documents/com~apple~CloudDocs/PP/Utdaning/DATA_2020/DATA2410/IndividulalPortofoli

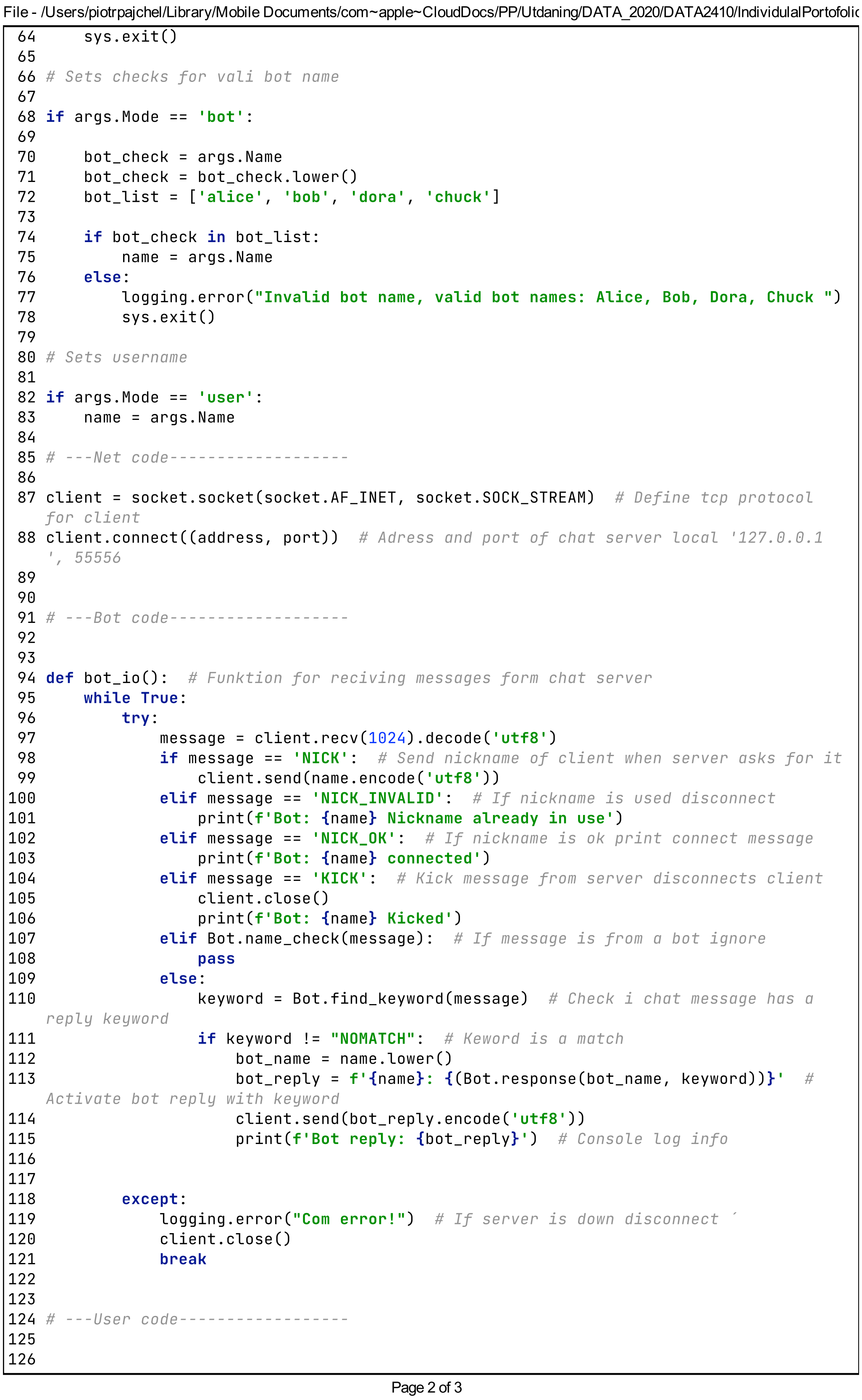
```python
# -------------------------------------------------
# Bot object
# -------------------------------------------------

import random


class Bot:

    def __init__(self, name):
        self.name = name


def find_keyword(string): #Checks if chat input contains trigger words
    keywords = ["work", "play", "eat", "cry", "sleep", "fight"]
    for word in string.split():
        if word in keywords:
            return word
    return "NOMATCH"


def name_check(string): #Checks if chat input is written by a bot
    bot_names = ["alice", "bob", "dora", "chuck"]
    word = string.split()[0].replace(":", "")
    if word.lower() in bot_names:
        return True


def response(bot_type, a, b=None): # Bot types with input a and b
    if bot_type == "alice":
        return f"I think {a}ing sounds great!"

    if bot_type == "bob":
        if b is None:
            return f"Not sure about {a}ing. Don't I get a choice?"
        return f"Sure, both {a} and {b}ing seems ok to me"

    if bot_type == "dora":
        alternatives = ["coding", "singing", "sleeping", "fighting"]
        b = random.choice(alternatives)
        res = f"Yea, {a} is an option. Or we could do some {b}."
        return res  # , b  # Returns tuplet

    if bot_type == "chuck":
        action = a + "ing"
        bad_things = ["fighting", "bickering", "yelling", "complaining"]
        good_things = ["singing", "hugging", "playing", "working"]
        if action in bad_things:
            return f"YESS! Time for {action}"
        elif action in good_things:
            return f"What? {action} sucks. Not doing that."
        return "I don't care!"
```

Page 1 of 1



File - /Users/piotrpajchel/Library/Mobile Documents/com~apple~CloudDocs/PP/Utdaning/DATA_2020/DATA2410/IndividulalPortofolic

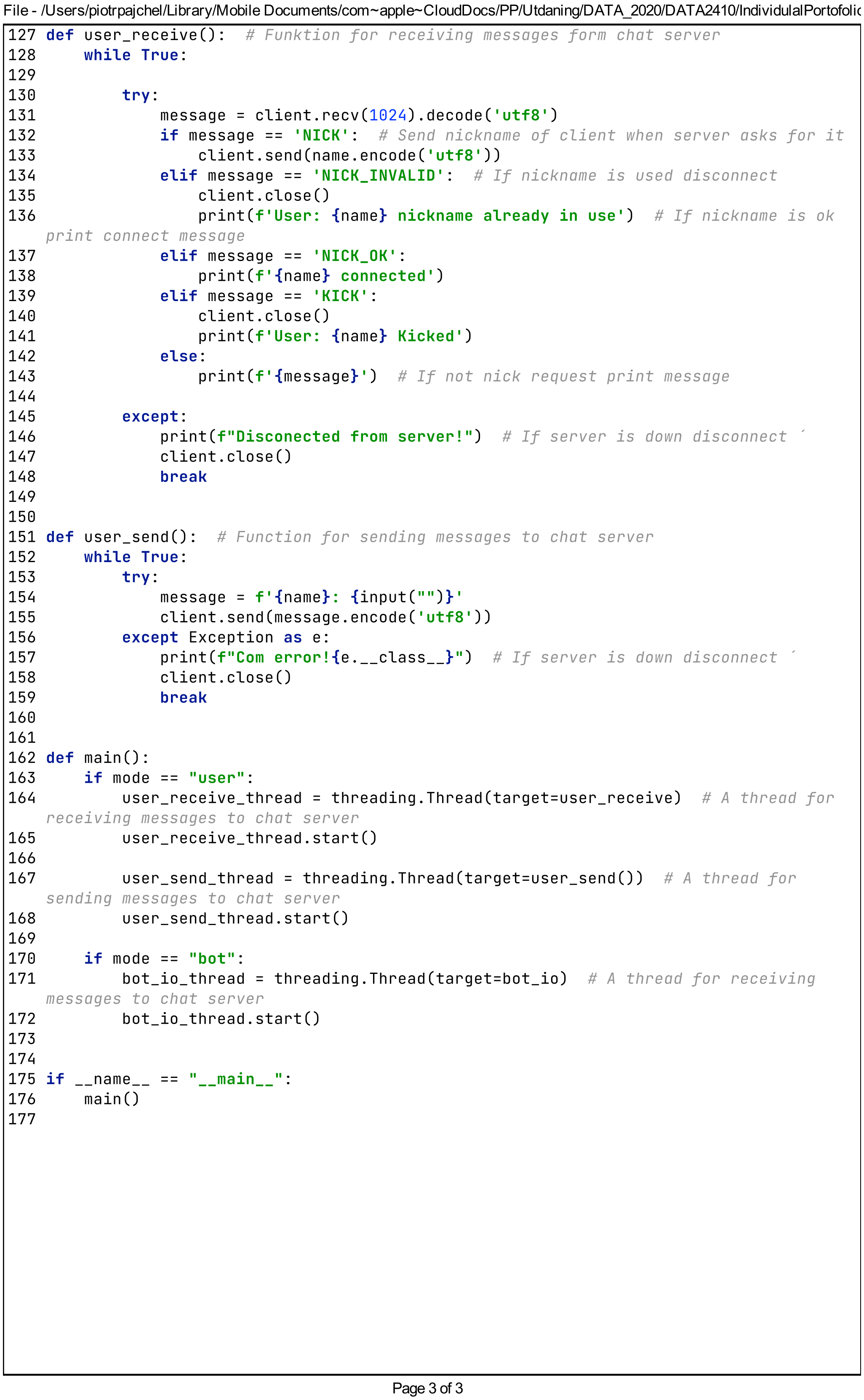
```python
# -------------------------------------------------
# Client
# -------------------------------------------------


import socket
import threading
import Bot
import sys
import logging
import argparse
import re

# ---Input validation-------------------

# Create the parser
my_parser = argparse.ArgumentParser(description='User/bot chat client for chatychaty server')

# Requierd comand line arguments for  clinet.py
my_parser.add_argument('Ip',
                       metavar='ip',
                       type=str,
                       help='Ip adress of server [0-255].[0-255].[0-255].[0-255] ')

my_parser.add_argument('Port',
                       metavar='port',
                       type=int,
                       help='Port number of server [0 - 65535]')

my_parser.add_argument('Mode',
                       metavar='mode',
                       type=str,
                       help='Two modes: user or bot | [user] or [bot]')

my_parser.add_argument('Name',
                       metavar='name',
                       type=str,
                       help='If in bot mode type bot name,Available bots: Alice, Bob, Dora, Chuck\n If in user mode '
                            'type nickname')

# Execute the parse_args() method

args = my_parser.parse_args()

# Check for valid ip format and set ip

valid_ipaddress_regex = "^(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\.){3}([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])$";
ip_regexp = re.search(valid_ipaddress_regex, args.Ip)

if ip_regexp:
    address = args.Ip  # server address
else:
    logging.error("Not a valid ip format, valid format: [0-255].[0-255].[0-255].[0-255] ")
    sys.exit()

    # Sets port number
port = args.Port

# Check for user mode
if (args.Mode == 'user') or (args.Mode == 'bot'):
    mode = args.Mode  # user or bot mode bot
else:
    logging.error("Not valid mode, valid modes: user, bot ")
```



File - /Users/piotrpajchel/Library/Mobile Documents/com~apple~CloudDocs/PP/Utdaning/DATA_2020/DATA2410/IndividulalPortofoli

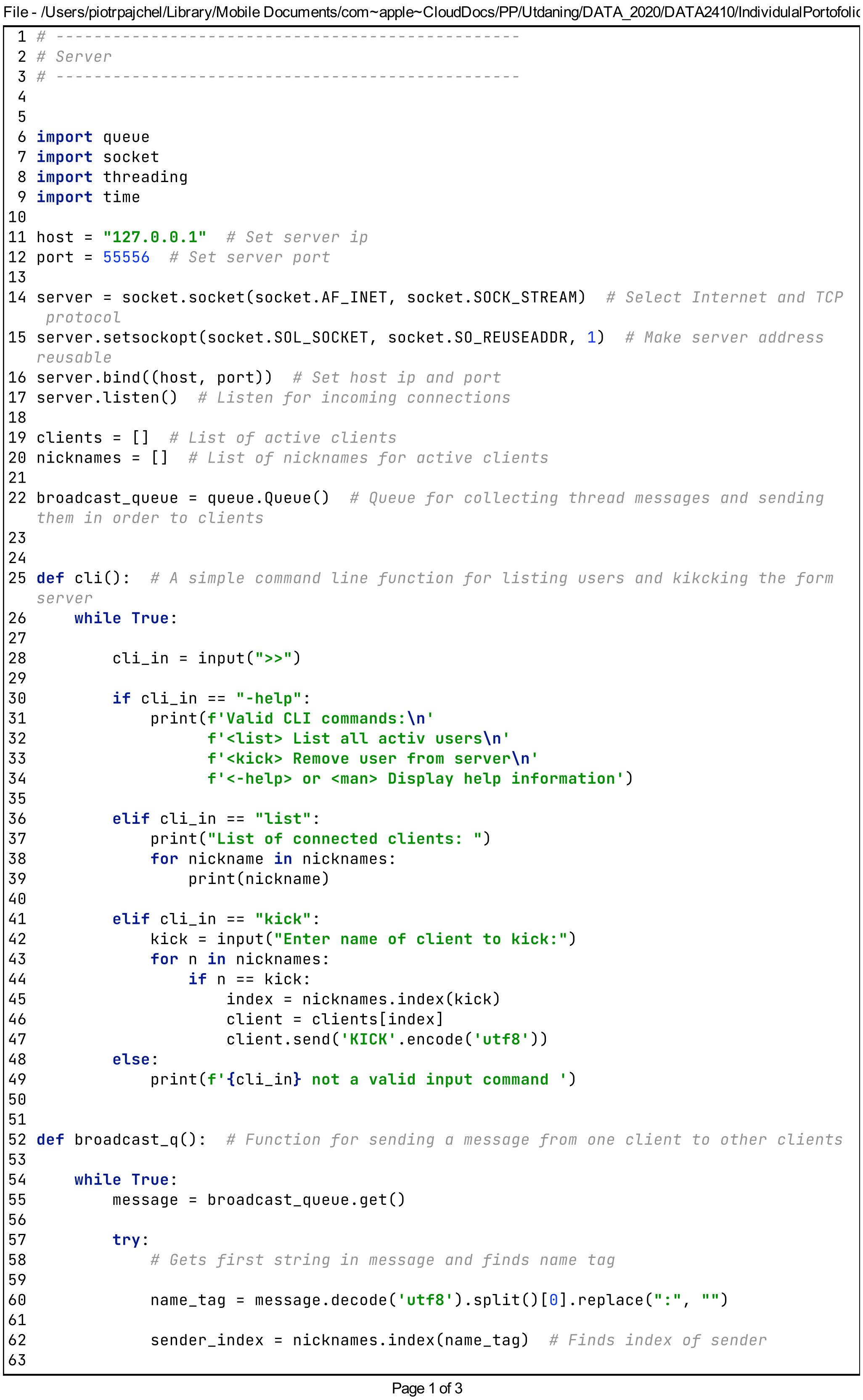
```python
    sys.exit()

# Sets checks for vali bot name

if args.Mode == 'bot':

    bot_check = args.Name
    bot_check = bot_check.lower()
    bot_list = ['alice', 'bob', 'dora', 'chuck']

    if bot_check in bot_list:
        name = args.Name
    else:
        logging.error("Invalid bot name, valid bot names: Alice, Bob, Dora, Chuck ")
        sys.exit()

# Sets username

if args.Mode == 'user':
    name = args.Name

# ---Net code-------------------

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Define tcp protocol for client
client.connect((address, port))  # Adress and port of chat server local '127.0.0.1', 55556


# ---Bot code-------------------


def bot_io():  # Funktion for reciving messages form chat server
    while True:
        try:
            message = client.recv(1024).decode('utf8')
            if message == 'NICK':  # Send nickname of client when server asks for it
                client.send(name.encode('utf8'))
            elif message == 'NICK_INVALID':  # If nickname is used disconnect
                print(f'Bot: {name} Nickname already in use')
            elif message == 'NICK_OK':  # If nickname is ok print connect message
                print(f'Bot: {name} connected')
            elif message == 'KICK':  # Kick message from server disconnects client
                client.close()
                print(f'Bot: {name} Kicked')
            elif Bot.name_check(message):  # If message is from a bot ignore
                pass
            else:
                keyword = Bot.find_keyword(message)  # Check i chat message has a reply keyword
                if keyword != "NOMATCH":  # Keword is a match
                    bot_name = name.lower()
                    bot_reply = f'{name}: {(Bot.response(bot_name, keyword))}'  # Activate bot reply with keyword
                    client.send(bot_reply.encode('utf8'))
                    print(f'Bot reply: {bot_reply}')  # Console log info


        except:
            logging.error("Com error!")  # If server is down disconnect ´
            client.close()
            break


# ---User code------------------


```

Page 2 of 3



File - /Users/piotrpajchel/Library/Mobile Documents/com~apple~CloudDocs/PP/Utdaning/DATA_2020/DATA2410/IndividulalPortofolic

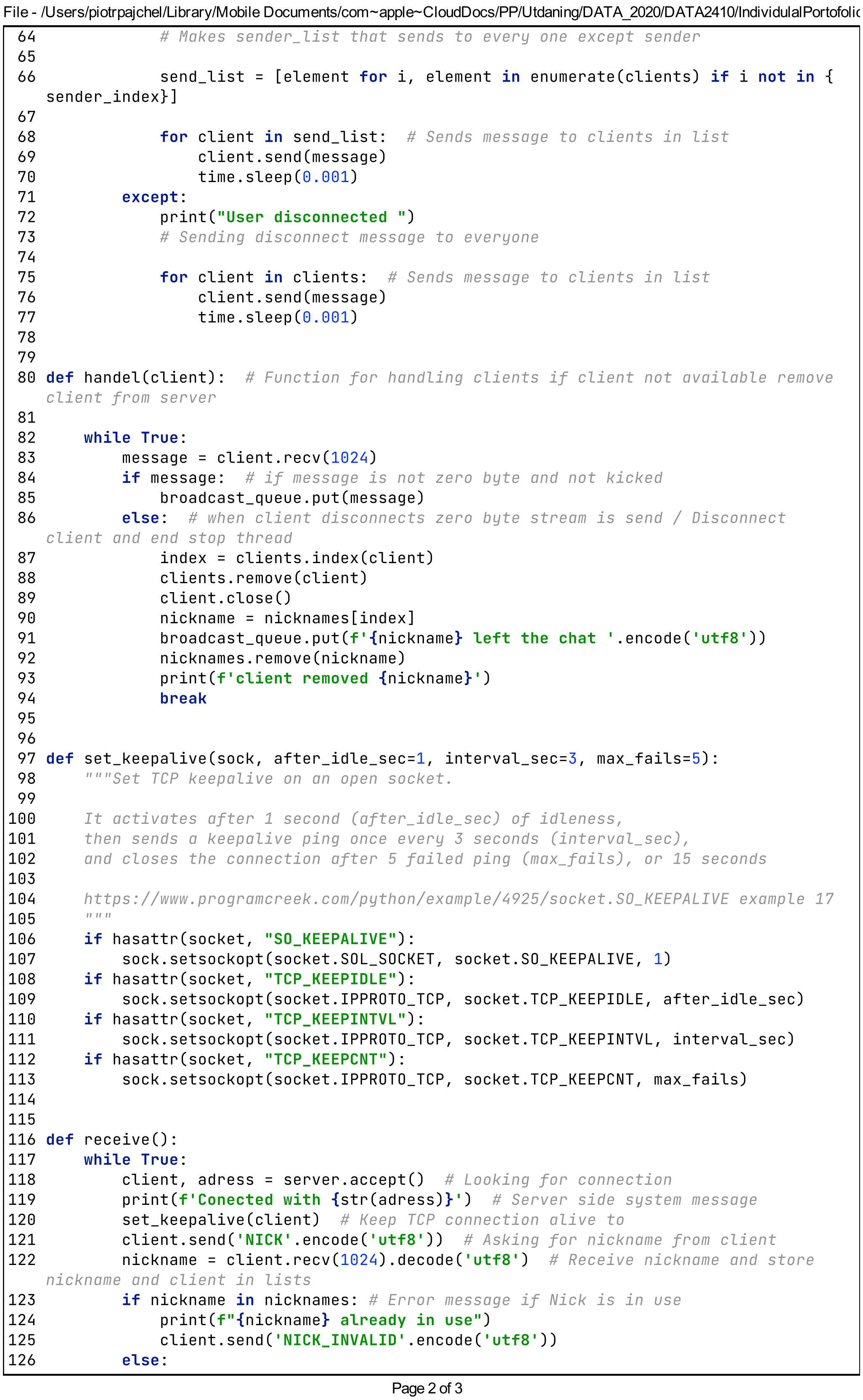
```python
def user_receive():  # Funktion for receiving messages form chat server
    while True:

        try:
            message = client.recv(1024).decode('utf8')
            if message == 'NICK':  # Send nickname of client when server asks for it
                client.send(name.encode('utf8'))
            elif message == 'NICK_INVALID':  # If nickname is used disconnect
                client.close()
                print(f'User: {name} nickname already in use')  # If nickname is ok print connect message
            elif message == 'NICK_OK':
                print(f'{name} connected')
            elif message == 'KICK':
                client.close()
                print(f'User: {name} Kicked')
            else:
                print(f'{message}')  # If not nick request print message

        except:
            print(f"Disconected from server!")  # If server is down disconnect ´
            client.close()
            break


def user_send():  # Function for sending messages to chat server
    while True:
        try:
            message = f'{name}: {input("")}'
            client.send(message.encode('utf8'))
        except Exception as e:
            print(f"Com error!{e.__class__}")  # If server is down disconnect ´
            client.close()
            break


def main():
    if mode == "user":
        user_receive_thread = threading.Thread(target=user_receive)  # A thread for receiving messages to chat server
        user_receive_thread.start()

        user_send_thread = threading.Thread(target=user_send())  # A thread for sending messages to chat server
        user_send_thread.start()

    if mode == "bot":
        bot_io_thread = threading.Thread(target=bot_io)  # A thread for receiving messages to chat server
        bot_io_thread.start()


if __name__ == "__main__":
    main()

```



File - /Users/piotrpajchel/Library/Mobile Documents/com~apple~CloudDocs/PP/Utdaning/DATA_2020/DATA2410/IndividulalPortofolic

```python
# -------------------------------------------------
# Server
# -------------------------------------------------


import queue
import socket
import threading
import time

host = "127.0.0.1"  # Set server ip
port = 55556  # Set server port

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # Select Internet and TCP protocol
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)  # Make server address reusable
server.bind((host, port))  # Set host ip and port
server.listen()  # Listen for incoming connections

clients = []  # List of active clients
nicknames = []  # List of nicknames for active clients

broadcast_queue = queue.Queue()  # Queue for collecting thread messages and sending them in order to clients


def cli():  # A simple command line function for listing users and kikcking the form server
    while True:

        cli_in = input(">>")

        if cli_in == "-help":
            print(f'Valid CLI commands:\n'
                  f'<list> List all activ users\n'
                  f'<kick> Remove user from server\n'
                  f'<-help> or <man> Display help information')

        elif cli_in == "list":
            print("List of connected clients: ")
            for nickname in nicknames:
                print(nickname)

        elif cli_in == "kick":
            kick = input("Enter name of client to kick:")
            for n in nicknames:
                if n == kick:
                    index = nicknames.index(kick)
                    client = clients[index]
                    client.send('KICK'.encode('utf8'))
        else:
            print(f'{cli_in} not a valid input command ')


def broadcast_q():  # Function for sending a message from one client to other clients

    while True:
        message = broadcast_queue.get()

        try:
            # Gets first string in message and finds name tag

            name_tag = message.decode('utf8').split()[0].replace(":", "")

            sender_index = nicknames.index(name_tag)  # Finds index of sender

```

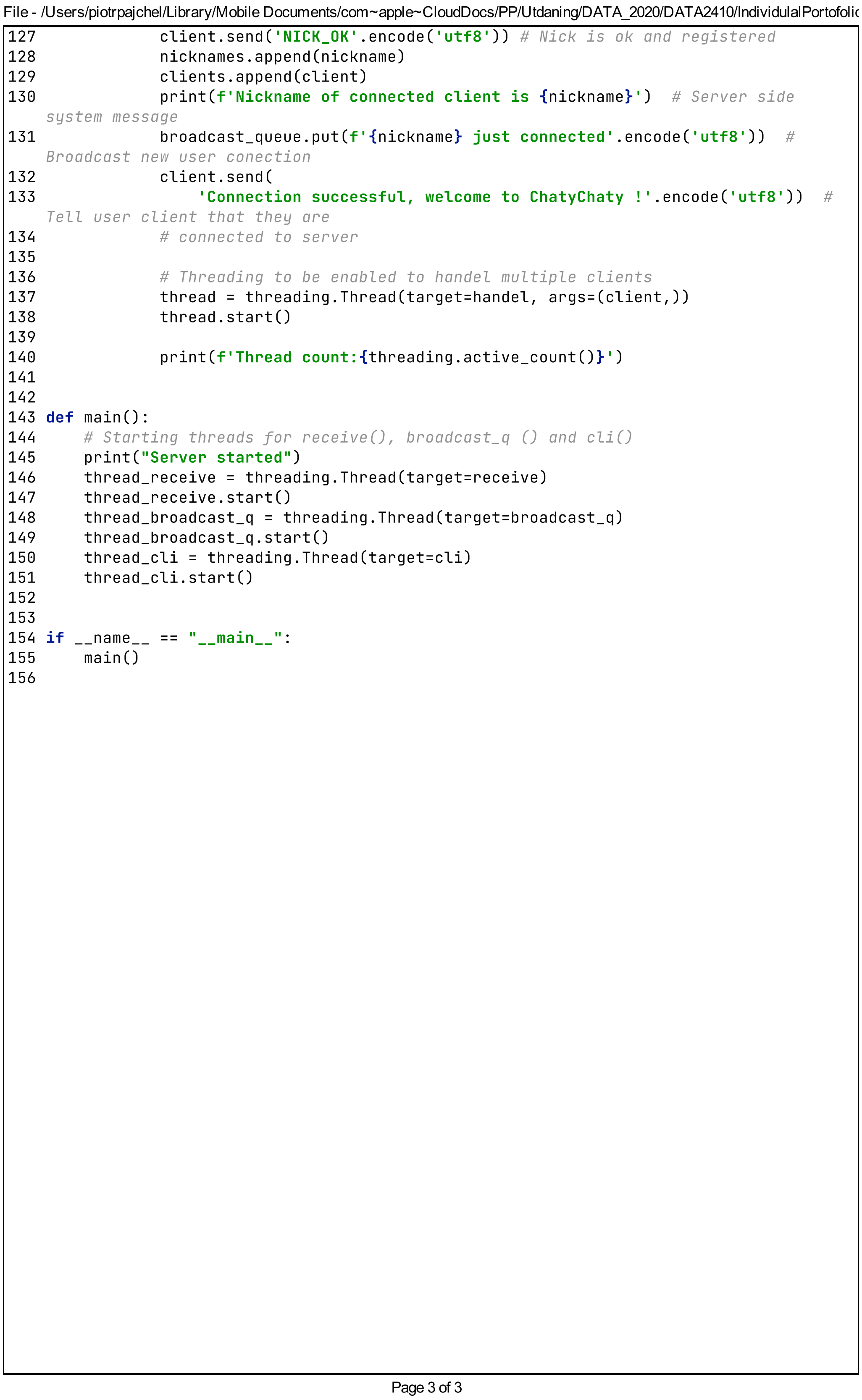
Page 1 of 3



File - /Users/piotrpajchel/Library/Mobile Documents/com~apple~CloudDocs/PP/Utdaning/DATA_2020/DATA2410/IndividulalPortofolic

```python
            # Makes sender_list that sends to every one except sender

            send_list = [element for i, element in enumerate(clients) if i not in {sender_index}]

            for client in send_list:  # Sends message to clients in list
                client.send(message)
                time.sleep(0.001)
        except:
            print("User disconnected ")
            # Sending disconnect message to everyone

            for client in clients:  # Sends message to clients in list
                client.send(message)
                time.sleep(0.001)


def handel(client):  # Function for handling clients if client not available remove client from server

    while True:
        message = client.recv(1024)
        if message:  # if message is not zero byte and not kicked
            broadcast_queue.put(message)
        else:  # when client disconnects zero byte stream is send / Disconnect client and end stop thread
            index = clients.index(client)
            clients.remove(client)
            client.close()
            nickname = nicknames[index]
            broadcast_queue.put(f'{nickname} left the chat '.encode('utf8'))
            nicknames.remove(nickname)
            print(f'client removed {nickname}')
            break


def set_keepalive(sock, after_idle_sec=1, interval_sec=3, max_fails=5):
    """Set TCP keepalive on an open socket.

    It activates after 1 second (after_idle_sec) of idleness,
    then sends a keepalive ping once every 3 seconds (interval_sec),
    and closes the connection after 5 failed ping (max_fails), or 15 seconds

    https://www.programcreek.com/python/example/4925/socket.SO_KEEPALIVE example 17
    """
    if hasattr(socket, "SO_KEEPALIVE"):
        sock.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1)
    if hasattr(socket, "TCP_KEEPIDLE"):
        sock.setsockopt(socket.IPPROTO_TCP, socket.TCP_KEEPIDLE, after_idle_sec)
    if hasattr(socket, "TCP_KEEPINTVL"):
        sock.setsockopt(socket.IPPROTO_TCP, socket.TCP_KEEPINTVL, interval_sec)
    if hasattr(socket, "TCP_KEEPCNT"):
        sock.setsockopt(socket.IPPROTO_TCP, socket.TCP_KEEPCNT, max_fails)


def receive():
    while True:
        client, adress = server.accept()  # Looking for connection
        print(f'Conected with {str(adress)}')  # Server side system message
        set_keepalive(client)  # Keep TCP connection alive to
        client.send('NICK'.encode('utf8'))  # Asking for nickname from client
        nickname = client.recv(1024).decode('utf8')  # Receive nickname and store nickname and client in lists
        if nickname in nicknames: # Error message if Nick is in use
            print(f"{nickname} already in use")
            client.send('NICK_INVALID'.encode('utf8'))
        else:
```

Page 2 of 3



File - /Users/piotrpajchel/Library/Mobile Documents/com~apple~CloudDocs/PP/Utdaning/DATA_2020/DATA2410/IndividulalPortofolic

```python
            client.send('NICK_OK'.encode('utf8')) # Nick is ok and registered
            nicknames.append(nickname)
            clients.append(client)
            print(f'Nickname of connected client is {nickname}')  # Server side system message
            broadcast_queue.put(f'{nickname} just connected'.encode('utf8'))  # Broadcast new user conection
            client.send(
                'Connection successful, welcome to ChatyChaty !'.encode('utf8'))  # Tell user client that they are
            # connected to server

            # Threading to be enabled to handel multiple clients
            thread = threading.Thread(target=handel, args=(client,))
            thread.start()

            print(f'Thread count:{threading.active_count()}')


def main():
    # Starting threads for receive(), broadcast_q () and cli()
    print("Server started")
    thread_receive = threading.Thread(target=receive)
    thread_receive.start()
    thread_broadcast_q = threading.Thread(target=broadcast_q)
    thread_broadcast_q.start()
    thread_cli = threading.Thread(target=cli)
    thread_cli.start()


if __name__ == "__main__":
    main()

```

Page 3 of 3

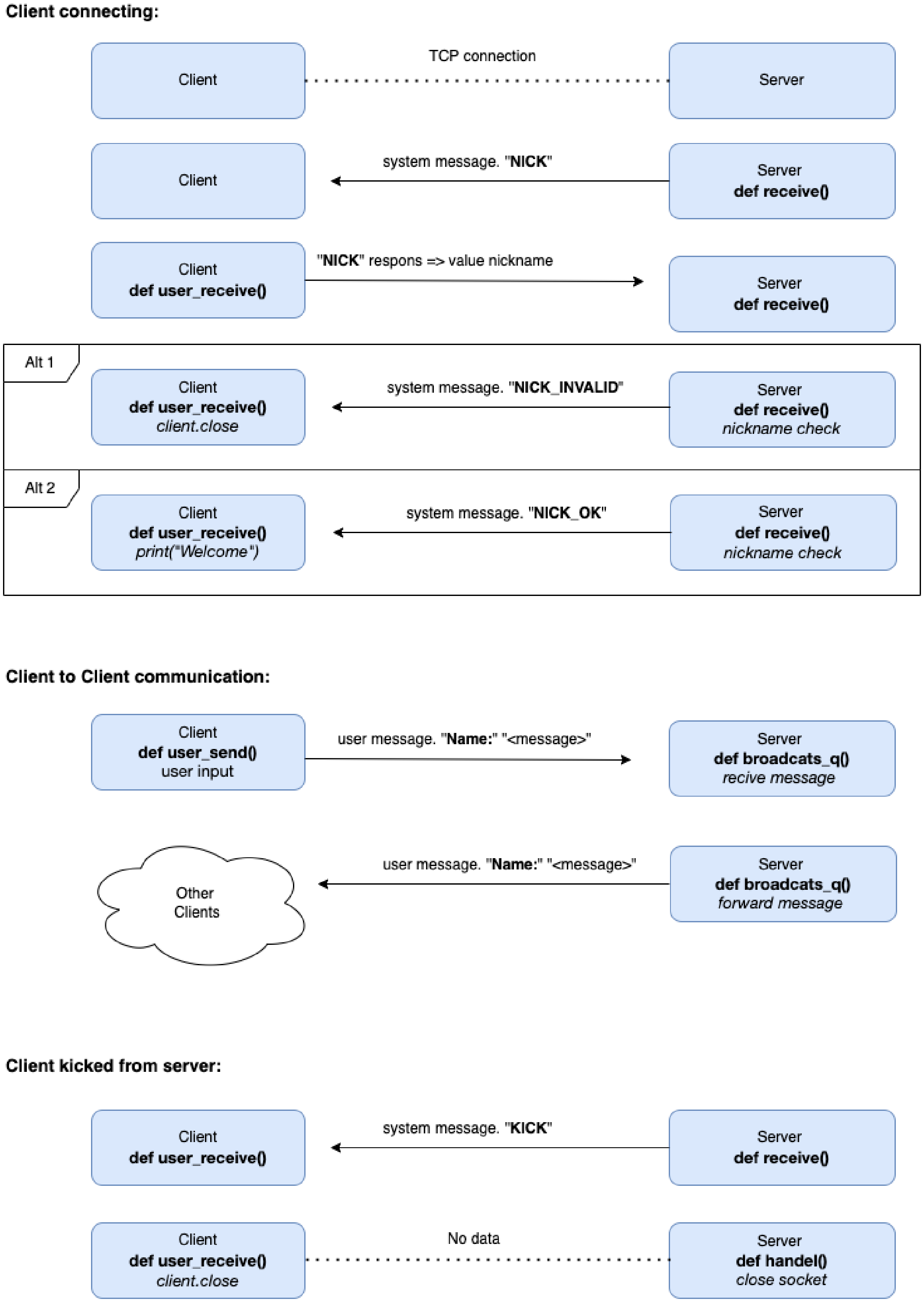


Figure 2

Overview of the client/server protocol commands (P.Pajchel 2022)