MiniProject Summary Report

Himanshu Banerji – 210905180

Implementing chat feature using python socket programming:

Working with client.py,

#Setting up the initial config,

```
import socket
import select
import errno

HEADER_LENGTH = 10

IP = "127.0.0.1"
PORT = 9989
my_username = input("Username: ")

# Create a socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect to a given ip and port
client_socket.connect((IP, PORT))
```

 # Set connection to non-blocking state, so ".recv()" call won't block, just return some exception we'll handle

```
client_socket.setblocking(False)
```

 # Prepare username and header and send them
 # We need to encode username to bytes, then count number of bytes and prepare header of fixed size, that we encode to bytes as well

```
username = my_username.encode('utf-8')
username_header = f"{len(username):<{HEADER_LENGTH}}".encode('utf-8')
client_socket.send(username_header + username)

while True:

    # Wait for user to input a message
    message = input(f'{my_username} > ')

    # If message is not empty - send it
    if message:
```

 # Encode message to bytes, prepare header and convert to bytes, like for username above, then send

```
        message = message.encode('utf-8')
        message_header = f"{len(message):<{HEADER_LENGTH}}".encode('utf-8')
        client_socket.send(message_header + message)
```

```
try:
```
 # we'd want to loop over received messages (there might be more than one) and print them

```
    while True:
```
 # Receive our "header" containing username length, it's size is defined and constant
```
        username_header = client_socket.recv(HEADER_LENGTH)
```

 # If we received no data, server will close the connection
```
        if not len(username_header):
            print('Connection closed by the server')
            sys.exit()
```

 # Convert header to int value
```
        username_length = int(username_header.decode('utf-8').strip())
```

 # Receive and decode username
```
        username = client_socket.recv(username_length).decode('utf-8')
```

 # Now do the same for message (as we received username, we received whole message, there's no need to check if it has any length)
```
        message_header = client_socket.recv(HEADER_LENGTH)
        message_length = int(message_header.decode('utf-8').strip())
        message = client_socket.recv(message_length).decode('utf-8')

        # Print message
        print(f'{username} > {message}')
```

 # This is normal on non blocking connections - when there are no incoming data error is going to be raised
 # We are going to check for both - if one of them - that's expected, means no incoming data, continue as normal

```
except IOError as e:
    if e.errno != errno.EAGAIN and e.errno != errno.EWOULDBLOCK:
        print('Reading error: {}'.format(str(e)))
        sys.exit()
```

 # We just did not receive anything
continue
```
except Exception as e:
```
 # Any other exception - something happened, exit
```
    print('Reading error: '.format(str(e)))
    sys.exit()
```

Working with server.py,

#Setting up initial config,

```
import socket
import select

HEADER_LENGTH = 10

IP = "127.0.0.1"
PORT = 9989

# Create a socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# SO_ - socket option
# SOL_ - socket option level
# Sets REUSEADDR (as a socket option) to 1 on socket
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Bind, so server informs operating system that it's going to use given IP and port
# For a server using 0.0.0.0 means to listen on all available interfaces, useful to connect locally to 127.0.0.1 and remotely to LAN interface IP
server_socket.bind((IP, PORT))
server_socket.listen()


# List of sockets for select.select()
sockets_list = [server_socket]

# List of connected clients - socket as a key, user header and name as data
clients = {}
print(f'Listening for connections on {IP}:{PORT}...')


# Handles message receiving
def receive_message(client_socket):
    try:

    # Receive our "header" containing message length, it's size is defined and constant
        message_header = client_socket.recv(HEADER_LENGTH)

    # If we received no data, server will close the connection
     if not len(message_header):
            return False

    # Convert header to int value
        message_length = int(message_header.decode('utf-8').strip())

     # Return an object of message header and message data
        return {'header': message_header, 'data': client_socket.recv(message_length)}

    except:

     # If client closed connection violently, for example by pressing ctrl+c on his script or just lost his connection
     # socket.close() also invokes socket.shutdown(socket.SHUT_RDWR) what sends information about closing the socket (shutdown read/write) and that's also a cause when we receive an empty message
        return False

while True:
```

```
# Calls Unix select() system call or Windows select() WinSock call with three parameters:
#   - rlist - sockets to be monitored for incoming data
#   - wlist - sockets for data to be send to (checks if for example buffers are not full and socket is ready to send some data)
#   - xlist - sockets to be monitored for exceptions (we want to monitor all sockets for errors, so we can use rlist)
# Returns lists:
#   - reading - sockets we received some data on (that way we don't have to check sockets manually)
#   - writing - sockets ready for data to be send thru them
#   - errors  - sockets with some exceptions
# This is a blocking call, code execution will "wait" here and "get" notified in case any action should be taken
   read_sockets, _, exception_sockets = select.select(sockets_list, [], sockets_list)


 # Iterate over notified sockets for notified_socket in read_sockets:
   # If notified socket is a server socket - new connection, accept it

        if notified_socket == server_socket:

      # Accept new connection
      # That gives us new socket - client socket, connected to this given client only, it's unique for that client
      # The other returned object is ip/port set
            client_socket, client_address = server_socket.accept()

      # Client should send his name right away, receive it
            user = receive_message(client_socket)

      # If False - client disconnected before he sent his name
            if user is False:
                continue

       # Add accepted socket to select.select() list
            sockets_list.append(client_socket)

       # Also save username and username header
            clients[client_socket] = user
            print('Accepted new connection from {}:{}, username: {}'.format(*client_address,
user['data'].decode('utf-8')))

    # Else existing socket is sending a message
       # Receive message
         else:
            message = receive_message(notified_socket)

      # If False, client disconnected, cleanup
            if message is False:
                print('Closed connection
from:{}'.format(clients[notified_socket]['data'].decode('utf-8')))

        # Remove from list for socket.socket()
          sockets_list.remove(notified_socket)

        # Remove from our list of users
          del clients[notified_socket]
                continue

       # Get user by notified socket, so we will know who sent the message
            user = clients[notified_socket]
             print(f'Received message from {user["data"].decode("utf-8")}:
{message["data"].decode("utf-8")}')

      # Iterate over connected clients and broadcast message
        # But don't sent it to sender

            for client_socket in clients:
                if client_socket != notified_socket:
```

```
            # Send user and message (both with their headers)
            # reusing message header sent by sender, and saved username header send by user when he connected
                        client_socket.send(user['header'] + user['data'] + message['header'] +
message['data'])

    # Remove from list for socket.socket()
     sockets_list.remove(notified_socket)

    # Remove from our list of users
     del clients[notified_socket]
```