



OnlineChat: Chat Application for Multiple Clients

CSCE 513- Principles of Communication and Networks
Lab 2

Connor Rawls
C00211180
12/10/22

I. Introduction

Online chatting applications are a very popular method for people to communicate with other users. Such applications allow users to connect with others, share ideas, or simply learn how peoples' days are going. Creating an online chatting application provides a good exercise for learning networking protocols and how to use them in a practical manner. In this paper, the implementation for a simple online chat application is explained and discussed.

II. Project Requirements

The scenario for this chat application is under the guise of a classroom with a teacher and multiple students. It is intended that the teacher and students are enabled to talk amongst themselves through the application for various topics related to the class. As such, the requirements for this project are as follows:

- The application server manages clients in a robust manner.
- Clients can talk amongst themselves in public (broadcasting).
- Clients can talk amongst themselves in private.
- Clients do not communicate with each other directly, but rather communication is facilitated through the application server.

To accomplish the requirements for the project requirements, the application is partitioned into two separate programs. The first program is the server portion of the application. This portion allows possible clients to connect to the chatting services that it provides and handle the client actions appropriately. The second program is the client side of the application. The client program is what potential users will interact with to reach the services that the chat application offers. Specifically, the execution of the application is as follows:

1. Launch the application server.
2. Launch the client application.
3. Interact with the chat server through the client application.

III. Server Implementation

Socket Establishment

The server-side of the application is implemented in the file *Server.py*. To start this service, an administrator would simply call the command *python3 Server.py* from a terminal window. Once the application is launched, the server first establishes the socket that it is willing to offer to incoming clients to connect to. This operation can be viewed in Fig. 1.

```
110     # Create socket with TCP protocol
111     server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
112     server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
113
114     # Bind socket
115     server_sock.bind((ADDR, PORT))
116
117     # Listen for client connection
118     server_sock.listen(1)
119
120     PRINT_LOCK.acquire()
121     print('Server is online.')
122     PRINT_LOCK.release()
```

Figure 1: Establishing the Server Socket for Incoming Clients

Initialize Incoming Connections

Once the server is listening to the socket, it enters its forever loop to continually accept incoming client connections, as observed in Fig. 2. One can observe that the transport protocol followed in this implementation is TCP. As observed in *Line 130*, the server sends an acknowledgement to the client, informing them that they have properly established the connection to the server. From here, the server requests various identification information from the client, such as the requested username. After the client information is confirmed, the server then instantiates a new thread to handle the client for the rest of its lifespan, *Line 147*.

```
124     # Listen for new clients entering chat
125     while True:
126         conn, addr = server_sock.accept()
127         CONNS.append(conn)
128
129         # Send acknowledgement
130         sendMessage(conn, 'ACK')
131
132         # Request username (TODO needs own thread)
133         username = fetchMessage(conn)
134         username = resolveDupUser(username)
135
136         # Confirm username
137         USERNAMES[username] = conn
138         sendMessage(conn, username)
139
140         PRINT_LOCK.acquire()
141         print(f'{str(conn.getpeername())} {username} is now online.')
142         PRINT_LOCK.release()
143
144         broadcast(conn, f'\r{username} is now online.\n')
145
146         # Create new thread to handle client from here on out
147         _thread.start_new_thread(Client, (conn, username))
148
149         # Send welcome message
150         sendMessage(conn, WELCOME_MSG)
151
152         if TERMINATOR.leave(): break
153
154     for conn in CONNS: conn.close()
155     server_sock.close()
156
157     print('Closing server.')
```

Figure 2: Accepting and Initializing Incoming Client Connections

Client Handling

A class object is used to handle client actions once the main server thread relinquishes control. The class definition can be viewed in Fig. 3. Once a client connection has been binded to the *Client* object, the object enters its forever loop. The loop can be considered of two parts: client request and action conduction. A client interacts with the server through their terminal window in the form of string texts. The server receives all of these messages with *fetchMessage* (Line 46). By default, client messages are broadcasted for all of the chat users to see. However, the server will attempt to parse the message first to see if the client wishes to execute any specific

commands (*Line 47*). If a valid command is found, *Client* will execute the command utilizing the optional arguments the client has passed along with the command. The various commands presented to clients can be viewed in *Lines 28-38* of Fig. 3. These commands include requesting a list of possible commands, messaging other users in private, and creating private groups.

```
19 # Handles all client actions after initial connection
20 class Client:
21     def __init__(self, conn, username):
22         global CLIENTS
23         CLIENTS[username] = self
24         self.conn = conn
25         self.username = username
26         self.groups = []
27         self.action = {
28             'BROADCAST' : broadcast,
29             'GROUPCHAT' : broadcast,
30             'CREATEGROUP' : createGroup,
31             'JOINGROUP' : joinGroup,
32             'LEAVEGROUP' : leaveGroup,
33             'LIST_ACTIVEGROUPS' : sendMessage,
34             'LIST_ALLGROUPS' : sendMessage,
35             'LIST_ONLINE' : sendMessage,
36             'PM' : sendMessage,
37             'HELP' : sendMessage,
38             'ERROR' : sendMessage
39         }
40         self.interact()
41
42     def interact(self):
43         while True:
44             try:
45                 # Determine what client wishes to do
46                 message = fetchMessage(self.conn)
47                 action, args = fetchAction(message, self)
48                 # Conduct action
49                 self.action[action](args)
50
51                 # Client has gone offline
52             except:
53                 PRINT_LOCK.acquire()
54                 print(f'{str(self.conn.getpeername())} {self.username} has gone offline')
55                 PRINT_LOCK.release()
56                 broadcast(self.conn, f'\r{self.username} has gone offline.\n')
57                 break
58
59     self.destruct()
```

Figure 3: Client Class for Handling Client Connections and Actions

Group Chats

To enable clients to form their own private groups, the *Group* class is defined. The class definition for *Group* can be viewed in Fig. 4. Once a client is a member of a specific group, they are allowed to send and receive messages that only the additional group members have access to.

```

79  # Clients can form their own private groups (channels?)
80  class Group:
81      def __init__(self, groupname, username):
82          self.groupname = groupname
83          self.owner = username
84          self.group_users = {}
85          self.group_conns = []
86          self.addUser(self.owner)
87
88      def addUser(self, username):
89          global CLIENTS
90          conn = USERNAMES[username]
91          self.group_users[username] = conn
92          self.group_conns.append(conn)
93          grp_message = f'{username} has joined group {self.groupname}.\n'
94          broadcast(conn, grp_message, self.group_conns)
95          CLIENTS[username].groups.append(self.groupname)
96
97      def rmvUser(self, username):
98          global CLIENTS
99          conn = self.group_users[username]
100         grp_message = f'{username} has just left group {self.groupname}.\n'
101         broadcast(conn, grp_message, self.group_conns)
102         del self.group_users[username]
103         self.group_conns.remove(conn)
104         CLIENTS[username].groups.remove(self.groupname)

```

Figure 4: Group Class for Enabling Client Group Formation/Interaction

IV. Client Implementation

Connecting to the Server

The client side of the application is implemented in the file *Client.py*. To execute this program, users simply use the command `python3 Client.py <username>` in a terminal window. Once this command has been executed, the program will attempt to connect to the chat server. The implementation for this operation can be viewed in Fig. 5. We can observe how connecting to the server mirrors the server socket establishment.

```
16     # Create socket with TCP protocol
17     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
18
19     # Connect with server
20     s.connect((ADDR, PORT))
21
22     # Wait for acknowledgement
23     while True:
24         ack = fetchMessage(s)
25         if ack == 'ACK':
26             print('Connection to server established.')
27             break
28         else:
29             print('Could not establish connection to server.')
30             sys.exit()
31
32     # Initialize username for server
33     sendMessage(s, USERNAME)
34     USERNAME = fetchMessage(s)
35
36     promptUser()
```

Figure 5: Establishing the Connection to the Chat Server

Incoming/Outgoing Messages

Once the client has successfully connected to the server, the program enters its own forever loop. This operation can be observed in Fig. 6. The purpose of this loop is to continuously check the I/O interfaces for the client's OS. If the server is forwarding a message to the client (through the established socket), the client will then fetch the message and display it to the user through the terminal (*Lines 46-48*). If the user begins providing input to the terminal, the program will utilize the entry as a message-to-be-sent to the server (*Lines 51-53*).

```
38     # Send message to server
39     while True:
40         SOCKS = [sys.stdin, s]
41
42         read_socks, _, _ = select.select(SOCKS, [], [])
43
44         for sock in read_socks:
45             # Incoming messages
46             if sock == s:
47                 message = fetchMessage(sock)
48                 sys.stdout.write(message)
49
50             # Outgoing message
51             else:
52                 message = sys.stdin.readline()
53                 sendMessage(s, message)
54
55             promptUser()
56
57     if TERMINATOR.leave(): break
```

Figure 6: Detecting Incoming/Outgoing Messages on the Client Side

V. Other Features

Additional to the aforementioned features of *Server.py* and *Client.py*, there are a number of various features implemented in this online chatting paradigm. It can be inferred that server and client messaging protocols are very similar on a low level. To simplify the messaging process and reduce the redundancy of the program, a utility file *messaging.py* was created. The implementation of the previously seen *fetchMessage* and *sendMessage* functions can be viewed in Figs. 7 & 8, respectively. Being as these two functions are the backbone of this application, it is important to explain them amply.


```
3  # Receive message
4  def fetchMessage(conn):
5      data = ''
6      size_ex = None
7
8      while True:
9          buff = conn.recv(1024)
10
11         if size_ex is None:
12             buff = buff.decode().split(',')
13             size_ex = int(buff[0]) # Extract expected message size
14             del buff[0] # Remove size indicator
15             if len(buff) > 1: buff = ','.join(buff).encode()
16             else: buff = ''.join(buff).encode() # Recode
17
18         data = data + buff.decode()
19
20         if len(data.encode()) >= size_ex: break
21
22     return data
```

Figure 7: Receive Message from Remote Process

```
24  # Send message
25  def sendMessage(*argv):
26      # Needs ability for arg overload due to Thread.action call
27      if len(argv) == 2:
28          s = argv[0]
29          message = argv[1]
30      else:
31          s = argv[0][0]
32          message = argv[0][1]
33
34      try: message = message.decode() # We need a str object to continue
35      except (UnicodeDecodeError, AttributeError): pass
36
37      size_ex = len(message.encode())
38      message = str(size_ex) + ',' + message
39      message = message.encode()
40      unsent_data = len(message)
41      sent_data = 0
42
43      while True:
44          buff = s.send(message[sent_data:])
45          if buff >= unsent_data: break
46          if sent_data >= unsent_data: break
47          else: sent_data += buff
```

Figure 8: Send Message to Remote Process

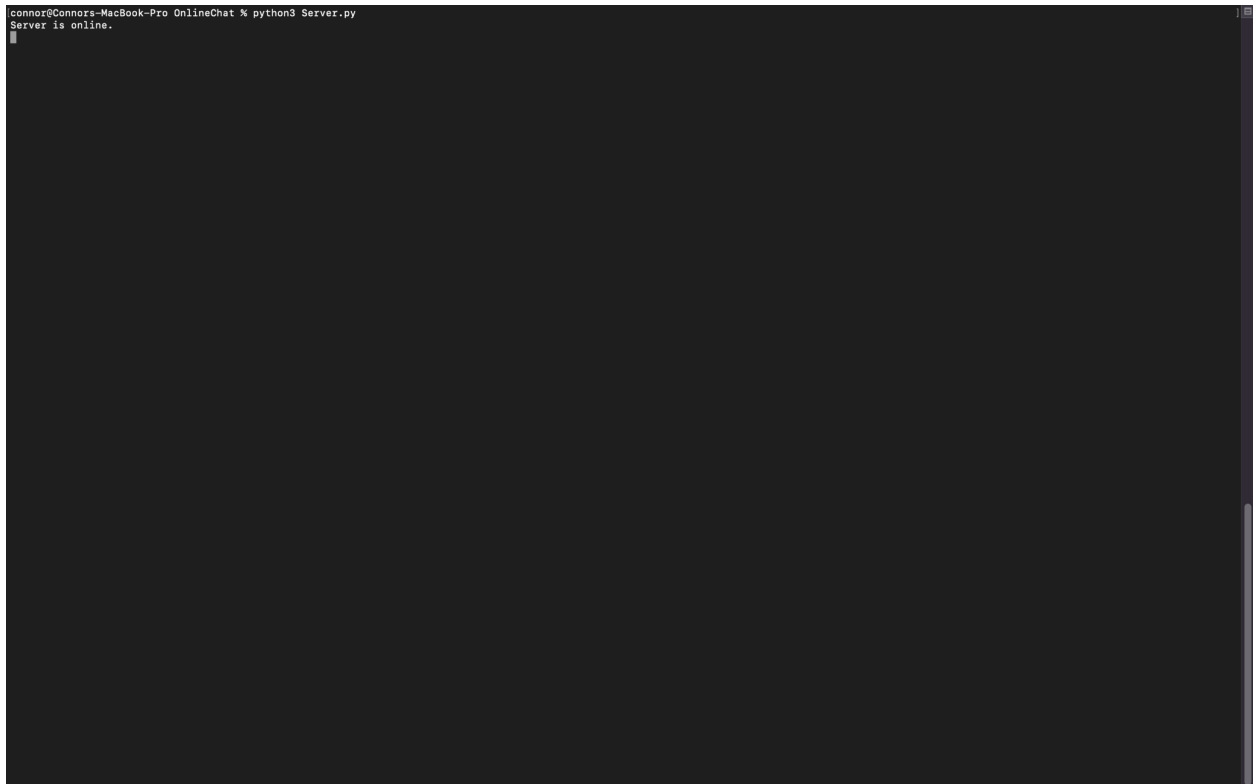
Receive Message

The main gist of how receiving a message works in this application is by iterating an unspecified amount of times until all of the message is received. *Line 9* of Fig. 7 is used to receive a given amount of data from a socket, with the maximum amount in a given loop iteration being 1024. The data of the current iteration is copied to a buffer and then appended to the overall data sequence received. However, recall that this operation is executed in a loop with an unspecified lifespan. In theory, the loop should be exited once all of the data has been received. How will the receiving end of a message know this information? It is a specified protocol in this application that all messages are prepended with their respective length. This information allows *fetchMessage* to parse the value out and determine what the size of the data sequence should be (*Lines 11-16* of Fig. 7). Once it is detected that the collective data sequence has reached the size that the sending application has specified, the loop will exit. From here, the message is finalized through decoding.

Send Message

On the opposing end of this application's communication, sending a message starts with the input string. *sendMessage* will first preprocess the string by determining its size. This value will be prepended to the string with a known delimiter to the application (*Lines 37-38* of Fig. 8). From here, a forever loop is run until all of the message has been sent to the receiver. Correspondingly, the relation between the forever loop, message size, and termination is similar to the same complex in *fetchMessage*. This loop can be viewed in *Lines 43-47* of Fig. 8.

VI. Demo

A terminal window with a dark background. The prompt is 'connor@Connors-MacBook-Pro OnlineChat %'. The command 'python3 Server.py' has been executed, and the output is 'Server is online.' followed by a cursor.

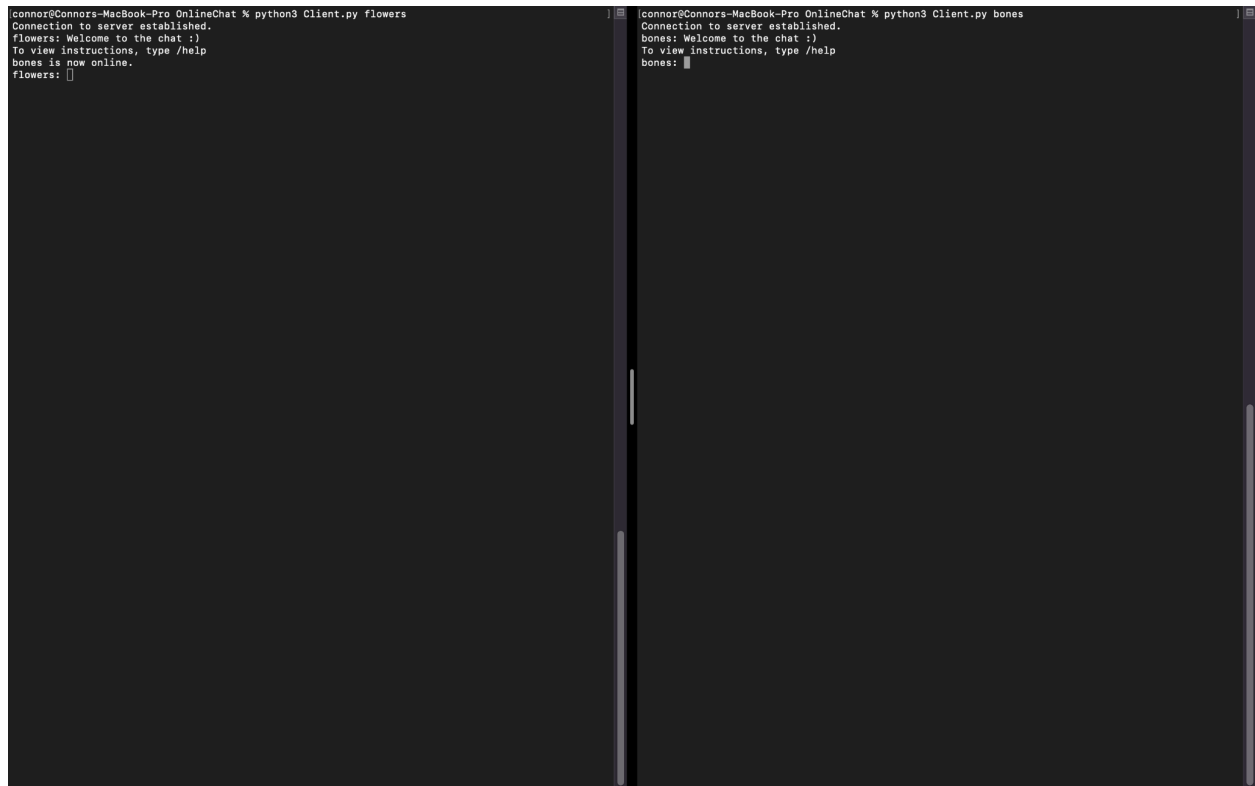
```
connor@Connors-MacBook-Pro OnlineChat % python3 Server.py
Server is online.
```

Figure 9: Server Initial Startup



```
connor@Connors-MacBook-Pro OnlineChat % python3 Client.py flowers
Connection to server established.
flowers: Welcome to the chat :)
To view instructions, type /help
flowers: 
```

Figure 10: First Client Joins (Left Side Terminal)



```
connor@Connors-MacBook-Pro OnlineChat % python3 Client.py flowers
Connection to server established.
flowers: Welcome to the chat :)
To view instructions, type /help
bones is now online.
flowers:

connor@Connors-MacBook-Pro OnlineChat % python3 Client.py bones
Connection to server established.
bones: Welcome to the chat :)
To view instructions, type /help
bones:
```

Figure 11: Second Client Joins (Right Side Terminal)

```

connor@Connors-MacBook-Pro OnlineChat % python3 Client.py flowers
Connection to server established.
flowers: Welcome to the chat :)
To view instructions, type /help
bones is now online.
flowers: /help
flowers:
* Messages are broadcasted by default. *
/help View command options.
/pm <username> <message> Send user a private message.
/gc <groupname> <message> Send message to private group.
/cg <groupname> Create a private group.
/jg <groupname> Join a private group.
/lg <groupname> Leave a private group.
/listg List groups you are a member of.
flowers: hello :)
bones: hi
flowers: what's up?
(PM) bones: nm wby
flowers: /cg Awesome
flowers: Group Awesome created.
bones has joined group Awesome.
flowers: /gc Awesome welcome to the group
flowers: A group with that name already exists.
flowers: /gc Awesome welcome to the group
(Awesome) bones: thanks
flowers: ^C^C
connor@Connors-MacBook-Pro OnlineChat %

connor@Connors-MacBook-Pro OnlineChat % python3 Client.py bones
Connection to server established.
bones: Welcome to the chat :)
To view instructions, type /help
flowers: hello :)
bones: hi
flowers: what's up?
bones: /pm flowers nm wby
bones: /jg Awesome
(Awesome) flowers: welcome to the group
bones: /gc Awesome thanks
flowers has gone offline.
bones: ^C^C
connor@Connors-MacBook-Pro OnlineChat %

```

Figure 12: Client Interaction

```

connor@Connors-MacBook-Pro OnlineChat % python3 Server.py
Server is online.
('127.0.0.1', 54565) flowers is now online.
('127.0.0.1', 54577) bones is now online.
('127.0.0.1', 54565) flowers: hello :)

('127.0.0.1', 54577) bones: hi
('127.0.0.1', 54565) flowers: what's up?

bones has joined Awesome.
"Awesome"
(Awesome) ('127.0.0.1', 54565) flowers: welcome to the group

"Awesome"
(Awesome) ('127.0.0.1', 54577) bones: thanks
('127.0.0.1', 54565) flowers has gone offline.

```

Figure 13: Server's Display of Interaction