

# CSE 489/589

## Programming Assignment 1 Report

### Text Chat Application

#### Notes: (IMPORTANT)

- One of your group members select <File> - <Make a copy> to make a copy of this report for your group, and share that Google Doc copy with your teammates so that they can also edit it.
- Report your work in each section. Describe the method you used, the obstacles you met, how you solved them, and the results. You can take screenshots at key points. There are NO hard requirements for your description.
- For a certain command/event, if you successfully implemented it, **take a screenshot of the result from the grader (required)**. You will get full points if it can pass the corresponding test case of the automated grader.
- For a certain command/event, if you tried but failed to implement it, properly describe your work. We will partially grade it based on the work you did.
- **Do NOT claim anything you didn't implement.** If you didn't try on a certain command or event, leave that section blank. We will randomly check your code, and if it does not match the work you claimed, you and your group won't get any partial grade score for this WHOLE assignment.
- There will be 10 bonus points for this report. Grading will be based on the organization, presentation, and layout of your report.
- After you finish, export this report as a PDF file and submit it on the UBLearns. For each group, only one member needs to make the submission.

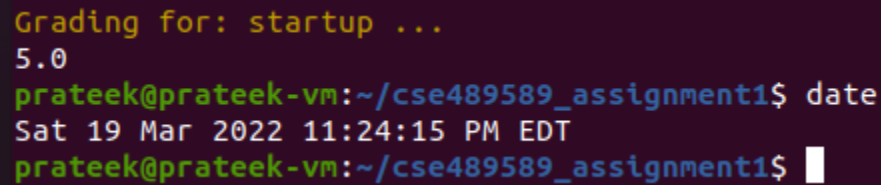
## 1 - Group and Contributions

- Name of member 1:
  - UBITName: kratisha
  - Contributions: Client Side Dev
- Name of member 2:
  - UBITName: prateekb
  - Contributions: Server Side Dev

## 2 - SHELL Functionality

### [5.0] Application Startup

(Describe how to start your application as a server/client, and how your application accept and process SHELL commands)

A terminal window with a dark purple background. The text is as follows:

```
Grading for: startup ...
5.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sat 19 Mar 2022 11:24:15 PM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

To start the server

```
./assignment1 s 4321
```

To start the client

```
./assignment1 c 5555
```

```
./assignment1 c 6666
```

If we start as a server, the server first binds a new tcp socket to port 4321 supplied and starts listening for new connection requests on the same.

If we start as a client, the client binds a new socket to the supplied port and does nothing until a Login command is supplied through shell.

On client, we create a new fd\_set and put STDIN and client socket id there to watch for. So that, the select function gets notified of the fd on which data is available. If user entered some command on the shell, then FD\_ISSET(STDIN, watchlist) will return true meaning that data is available from standard input, shell here. The client can then process the command from shell. If the client received msg from server on client socket, then FD\_ISSET(client\_socket\_fd) will be TRUE triggering the client to process the msg from server. In our case, we simply log RECEIVED msg event in that case.

On server, we do something similar. The difference here is that, we loop over all the sockets that we are watching and check whether data is available on that socket including the STDIN. If data is available, say in case the client sent a msg, then server processes that msg.

Since, server is listening on say server\_fd 2 and it receives a new client connection request via accept method, then a copy of the server\_fd, say 4 is created. So, the head\_count is now set to 4 in select method, since we now have to loop over 1, 2... 4 to check if this newly connected client socket to 4 sends any msg in future.

### 3 - Command for Server and Client

#### [0.0] AUTHOR

```
Grading for: author ...  
TRUE  
prateek@prateek-vm:~/cse489589_assignment1$ date  
Sat 19 Mar 2022 11:25:54 PM EDT  
prateek@prateek-vm:~/cse489589_assignment1$
```

#### [5.0] IP

```
Grading for: ip ...  
5.0  
prateek@prateek-vm:~/cse489589_assignment1$ date  
Sun 20 Mar 2022 01:25:08 AM EDT  
prateek@prateek-vm:~/cse489589_assignment1$
```

As per direction, we created a UDP socket and connected it to Google's DNS 8.8.8.8  
This made our local DNS to resolve our external IP address.

#### [2.5] PORT

```
Grading for: port ...  
2.5  
prateek@prateek-vm:~/cse489589_assignment1$ date  
Sun 20 Mar 2022 01:59:32 AM EDT  
prateek@prateek-vm:~/cse489589_assignment1$
```

both client and server ports are the ones supplied as command line args during startup.  
Both client and server bind a new TCP socket to this port. We explicitly made our clients to bind to this port. However, this was not necessary as a random available port could also be assigned automatically. For this same reason, we had to use the following line

```
setsockopt(client_fd, SOL_SOCKET, SO_REUSEADDR, &option, sizeof(option));
```

in the `get_new_binding()` method call at client side.

This configuration allows the socket to be rebound to the same port. This is important because when the client logs out and logs in again, the old binding is not quickly lost and new socket can't be bound already in-use port if we don't use the above config.

## [10.0] LIST

```
Grading for: _list ...
10.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 02:01:55 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

The server maintains a list of all the client ips that have connected to it since startup. This list is used to relay msgs to clients. This list also internally stores file descriptors of each client to identify their connection sockets later. This is hidden in the Client struct and not displayed.

## 4 - Command/Event for Server

### [5.0] STATISTICS

```
Grading for: statistics ...
5.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 02:03:39 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

This is a simple command to display the list of all clients until now, the number of msgs they have sent and received. Works fine.

### [7.0] BLOCKED <client-ip> + Exception Handling

```
Grading for: blocked ...
5.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 02:04:50 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

This is a server side command to list all the blocked users by a particular user. The server maintains a list of {blocker, blocked} pairs on its side as a global vector object. The command makes server filter through this list and display results.

#### Exception

IP is validated using the `is_valid_ip()` method. If `inet_pton()` is able to convert this ip to binary network format, we say that the ip is valid.

```
Starting grading server ...
OK

Grading for: exception_blocked ...
2.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 02:06:11 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

## [EVENT]: Message Relayed

The server prints/logs the details of a msg on its side wherever it is successful in relaying an incoming msg.

## 5 - Command/Event for Client

### [17.0] LOGIN <server-ip> <server-port> + Exception Handling

Combination of `_list` and `buffer` test cases sum up to 15 points for this feature

```
Grading for: _list ...
10.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 02:01:55 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

```
Grading for: buffer ...
5.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 02:09:36 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

The client receives the server ip and port number as part of the input parameters of the LOGIN command. It then tries to connect to this ip and port using the `client_socket`. If the connection is successful, it can then `send()` or `recv()` bytes over this `client_socket`. The server on receiving the new connection request using the `accept()` method accepts the connection and stores this client details in its `client_list` global variable. It then sends back an integer of 4 bytes using the `send_int_over_socket()` method. This integer represents the number of pending msgs for this client. The client on the other hand is waiting on receiving this number using the `recv()` blocking call. It is important for the server to send 0 even if there are no pending msgs, because `recv()` is

a blocking call and the client will keep on waiting forever(since we didn't put a timeout), if server doesn't let it know. Once the client has received the number/integer of pending msgs. It then loops n times to recv() these msgs one by one. Again, it is expecting the server to send the data one by one. The server on its side loops n times and send() these pending msgs one by one. The client and server are both happy now and go their own way.

#### Exception Handling

```
OK
Grading for: exception_login ...
2.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 02:10:42 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

We check for the validity of ip using the inet\_pton() as mentioned earlier. We also check where port number lies within the range of port numbers, etc

#### [5.0] REFRESH

```
Grading for: refresh ...
5.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 02:37:06 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

Refresh simply pulls the global client\_list from server to client. The server encodes the list in a fashion where each line is separated by :::: and each value is separated by \$\$\$. The client decodes this msg on its side and creates a new c\_client\_list to use later.

#### [17.0] SEND <client-ip> <msg> + Exception Handling

```
Grading for: send ...
15.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 02:40:00 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

The client sends a msg to server stating the receiver ip address. It encodes the msg in the same fashion described earlier. COMMAND:::::to\_ip\$\$\$message\_body

The server receives the msg and decodes it. It check whether the receiver is in the block list. If he is blocked by sender client, then it ignores the msgs and msg is lost. Otherwise, if client is offline, then it adds it to a pending list of messages to be delivered when the intended receiver LOGINs again. If receiver is online, then server immediately relays it to him.

Note:

Since the msg body and encoded string fit in one packet in our case, there is no issue. Otherwise, this can break as recv() and send() are not gauranteed to send or receive the complete msg in one go. Broken msg can't be decoded properly and might lead to abnormal program behaviour because unread part will start acting as a new command when read. To fix this, there are methods out of scope of this assignment.

### Exception Handling

```
Grading for: exception_send ...
2.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 02:41:28 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

Before sending, the client checks its c\_client\_list whether it knows about the receiver ip prehand or not. Otherwise, error. It also checks whether ip is valid.

### [10.0] BROADCAST <msg>

```
Grading for: broadcast ...
10.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 03:02:23 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

This is very similar to send. One client sends one msg to server with an encoded command BROADCAST. The server understands that the client is asking for a broadcast, and it again checks the block list. Then it relays to the clients who are online and not blocked. Otherwise, it saves it in pending list to be sent later.

### [7.0] BLOCK <client-ip> + Exception Handling

```
Grading for: block ...
5.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 03:04:27 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

Both client and server maintain a `c_block_list` and `block_list` vectors respectively. Whenever, the client issues a command to block an ip, it adds it to its own local list as well as the server adds to its global `block_list`.

Exception

```
OK

Grading for: exception_block ...
2.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 03:05:39 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

Like valid ip, already blocked, etc are checked.

## [4.5] UNBLOCK <client-ip> + Exception Handling

```
Starting grading server ...
OK

Grading for: unblock ...
2.5
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 03:06:56 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

Exception

```
Starting grading server ...
OK

Grading for: exception_unblock ...
2.0
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 03:08:17 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

Similar to above. Just that this time, value is removed from the list rather than adding.

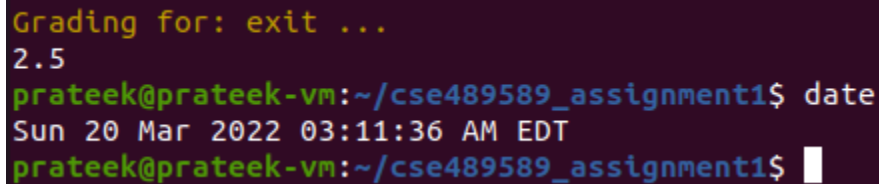
## [2.5] LOGOUT

```
Grading for: logout ...
2.5
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 03:09:52 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```



To terminate the connection gracefully, the client side socket sends a FIN segment to the server. When the server receives this, it closes the `socket_fd` associated with that client and sends an acknowledgment to the client. The client then closes its socket. The next time LOGIN is happening on the client, the client will create a new TCP socket binding and request a `connect()` to server.

## [2.5] EXIT



```
Grading for: exit ...
2.5
prateek@prateek-vm:~/cse489589_assignment1$ date
Sun 20 Mar 2022 03:11:36 AM EDT
prateek@prateek-vm:~/cse489589_assignment1$
```

To represent a hard exit, the client first sends a MSG to server saying that it wants to exit, the server then clears all data associated with this client like pending msgs, `block_list`, etc and then closes the socket associated with this client. Client also closes then on its side.

## [EVENT]: Message Received

The client socket `select()` method tracks the `client_socket` as well. So, when it is ready-to-read, it reads from this socket and prints/logs the msg received and from whom it was received. This is an encoded msg from the server containing `from_ip`, and `message_body`.

## 6 - BONUS: Peer-to-peer (P2P) file transfer

### [20.0] SENDFILE <client-ip> <file>

Not done. opted out.