CSE 489/589
Programming Assignment 1 Stage 2 Report
Text Chat Application

Notes: (IMPORTANT)

- ➔ Your submission will **NOT** be graded without submitting this report. It is required to use this report template. Do not add sections for Stage 1, as they will not be graded.
- ➔ 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 (optional). **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, **attach the 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. If you can get some points (not full) from the grader, also **attach the screenshot of the result from the grader (required).** 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 run your code**, and if it does not match the work you claimed largely, you and your group won't get any partial grade score for this WHOLE assignment.
- ➔ This report has 5 bonus points. Bonus points grading will be based on your **description** in each section, including the sections where you got full points from the grader.
- ➔ Maximum score for Stage 2: 58 + 5 = 63

1. Group and Contributions

- Name of member 1: Brecken McGeough
  - UBITName: breckenm
  - Contributions: All

2. Test results

[0.0] AUTHOR (author)

Your submission will not be graded if the AUTHOR command fails to work.
Grader screenshot:

Description:
(Describe your work here...)
Like the first stage, after the user inputs AUTHOR (either into client or server command prompt) logs the statement “I, breckenm, have read and understood the course academic integrity policy” to the console.

[15.0] SEND (send)

Grader screenshot:

Description:
(Describe your work here...)
When a client inputs “SEND <ip> <msg>” into the console, I make sure that the message length does not exceed 256 bytes before packaging the ip of the recipient client (client to receive the message) and msg into one string and sending it to the server alongside a special character that denotes that the info sent is to be sent to another client. When the server receives the information and sees that the special character for the SEND functionality is present, it parses the string and extracts the recipient ip and message to be send. Then, I input the message and ip into a string that follows the specified format (“msg from: <ip>\n[msg]:<msg>”). Then, I iterate through the linked list that is the clients list until it finds the recipient client with the matching ip. Then, it first iterates through the matching clients blocked linked list and checks if the ip of the client who sent the message appears in that list. If it doesn’t, that means they are not blocked. If the client who sent the message isn’t blocked, but the recipient client is offline (specified by a binary value stored in the clients structure: 1 for online, 0 for offline), the message is stored at the end of the recipient clients backlog linked list, which is a linked list (head pointer) in their structure which holds all of their backlog messages (“[RECEIVED:SUCCESS]\n” and “[RECEIVED:END]\n” are string concatenated onto the backlog message to preserver proper format specified in handout). Then, the message is not sent to the recipient, since they’re offline and the specified (Also, extra info like the recipient ip, sender ip, and message itself are stored separately within the backlog to display to the server the proper format RELAYED message event later on, instead of having to parse it from the backlog message). If the client who sent the message is not blocked and the recipient client is online logged into the server, then instead of storing the message in the backlog, the formatted message will be sent to the recipient client

(by storing their file descriptor inside their struct, can retrieve it and use it to send messages to them: fd are updated in a clients struct everytime they log off and log back in). The client then receives the message and formats it properly with “[RECEIVED:SUCCESS]\n [RECEIVED:END]\n.” in between the message and logs it to the console. This is possible because the client is always listening for new messages written to its fd using the select() function, much like the server uses the select function, except it doesnt loops through a bunch of file descriptors to check which one has new info written to it, it just continuously checks its own fd and STDOUT. When it detects new info is written to it using select(), and checking its fd with FD_SET, it recv()’s the message sent from the server and displays it. This offers the ability to receive messages in realtime and display them.

If a client is offline and receives messages from other clients, when they log back into the server, the server will check if the client who logs is has any messages in their backlog. If they do, the server will concatenate all the messages from the linked list (they are already formatted properly with each message having its own “[RECEIVED:SUCCESS]\n, [RECEIVED:END]\n attached to the message) into one string that the server will send to the client, and the client will receive and display. At the same time the backlog was being iterated through to construct the message to send, another string is being constructed which will contain all of the RELAYED EVENT messages for the server to display, all properly formatted where each backlog message had extra info stored like the recipient ip, sender ip, message itself, to construct each RELAYED message of the format “msg from:<sender ip>, msg to:<recipient ip>\n[msg]:<msg>. This string is then logged and printed to the servers console after sending the whole complete backlog message to the client to display.

Lastly, when a client successfully (not blocked) sends a message to another client, each client struct has two ints which store the number of message they sent and the number of messages they received, and the messages sent will be incremented by 1 for the sender and the received int will be incremented by 1 for the receiver.

[EVENT]: Message Relayed

Description:
(Describe your work here...)
Please see above description in SEND for details on how EVENT: message relayed is handled on the server side.

[EVENT]: Message Received

Description:
(Describe your work here...)
Please see above description in SEND for details on how EVENT: message relayed is handled on the client side (offline and online).

[10.0] BROADCAST (broadcast)

Grader screenshot:

Description:
(Describe your work here...)
When a client types in “BROADCAST <msg>”, and sends it to the server (alongside a special character to denote to the server that this message is to be broadcast), the server iterates through all of the clients structs and using each of their ips and file descriptors runs the code responsible for the SEND functionality on them. As it iterates through the linked list of clients, when it comes upon the client who broadcast the message, it will not send this client the message. So it basically just calls the same code for the SEND functionality on every client to send them the message except the one who broadcast the message. Then, it will print out just one RELAYED EVENT to the servers console where the recipient ip is 255.255.255.255

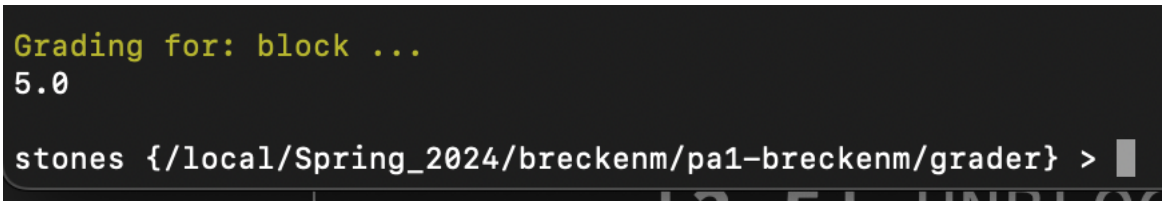
[5.0] STATISTICS (statistics)

Grader screenshot:

Description:
(Describe how you implement the LOGIN and LIST commands)
The STATISTICS command on the server side uses basically the same code as the LIST code, except it just changes the format to display the the number client they are (1st, 2nd, 3rd,…), hostname, number of messages sent, number of messages received, and their listening port, sorted by their listening ports.

[5.0] BLOCK (block)

Grader screenshot:



Description:
(Describe your work here...)
Every client struct stored a pointer to a linked list which hold ips of those they have blocked. When a client types in BLOCK <ip> and sends the ip to the server (with a special character), the server adds this ip to the end of their blocked linked list within their struct.

[5.0] BLOCKED (blocked)

Grader screenshot:

Description:
(Describe your work here...)
When the server types in BLOCKED <ip> into the prompt, it uses basically the same code as LIST and STATISTICS to sort all clients (except for this, its only the list of clients that appear in the specified ip clients blocked list) by their listening ports.

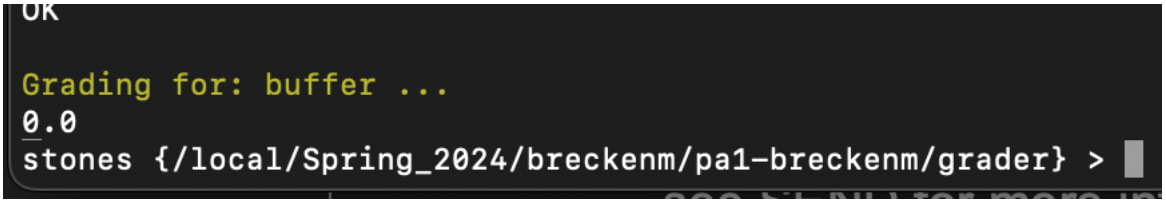
[2.5] UNBLOCK (unblock)

Grader screenshot:

Description:
(Describe your work here...)
If a client types UNBLOCK <ip> and sends the ip to the server (with a special character), the server iterates through their blocked linked list until it finds the ip that matches the ip they want to unblock, and then just sets all of the memory of that string to null terminators (so when that client sends a message to them, their ip is no longer in their blocked list and they are able to, see SEND for more info).

[5.0] (buffer)

Grader screenshot:



Description:
(Describe how your server stores/buffers messages)
Couldn't get full credit. I use a linked list to store backlog messages. Each client struct contains a pointer to another struct (head of a linked list) which contains a string and a pointer to another struct of the same kind. Buffered messages are stored at the back of the list and read from the head to preserver the order in which they were sent. For more info on how this works when clients log in, see SEND description. Also, I accept back to back messages (SEND <ip> <msg>\nSEND <ip2> <msg2>\n…) by parsing by \n in the string and then breaking each up into their own message and ip and using the SEND code on them in a while loop. I got it to (seemingly) work perfectly, but I still got 0 on the buffer. See my code for specifics including use of strtok(). Also, see SEND code description for details on how messages are stored in backlog and displayed for offline client that gets messages and then comes online.

[2.5] LOGOUT (logout)

Grader screenshot:

Description:
(Describe your work here...)
When the clien types LOGOUT, it sets the global online status to 0 (not online, which forbids LIST, REFRESH, SEND, BROADCAST, etc.) and sends their ip to the server. The server receiving the clients ip in a message alone means that they mean to log off, and so the server sets their status to offline (0), closes their socket and removes their fd from the watchlist. The client hasn't exit() though, so can still type in commands such as IP, LOGIN, AUTHOR, etc..

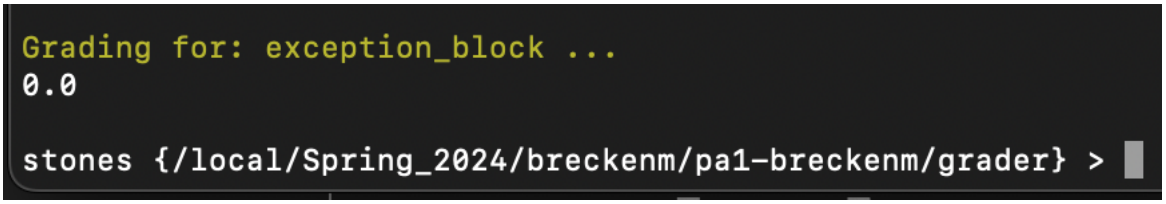
[2.0] SEND Exception Handling (exception_send)

Grader screenshot:

Description:
(Describe your work here...)
I couldn't get full credit but I made it so clients cant send messages to themselves by checking if the ip they are trying to send to is their own ip.

[2.0] BLOCK Exception Handling (exception_block)

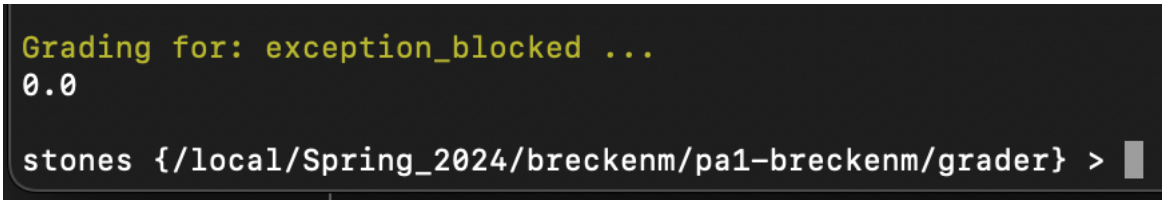
Grader screenshot:



Description:
(Describe your work here...)
Couldn't get full credit.

[2.0] BLOCKED Exception Handling (exception_blocked)

Grader screenshot:



Description:
(Describe your work here...)
Couldn't get full credit.

[2.0] UNBLOCK Exception Handling (exception_unblock)

Grader screenshot:

Description:
(Describe your work here...)
Couldn’t get full credit.