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.
- ➔ After you finish, export this report as a PDF file and submit it on the UBLearns along with your tarball. For each group, only one member needs to make the submission.
- ➔ Maximum score for Stage 2: 58 + 5 = 63

1. Group and Contributions

- Name of member 1: Kunal Chand
  - UBITName: kchand
  - Contributions: Server Side Implementation and Report
- Name of member 2:
  - UBITName: rahulyad
  - Contributions: Client Side Implementation and Report

Common Contributions:
print_functions.cpp

helper_functions.cpp
global.h

2. Test results

[0.0] AUTHOR (author)

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

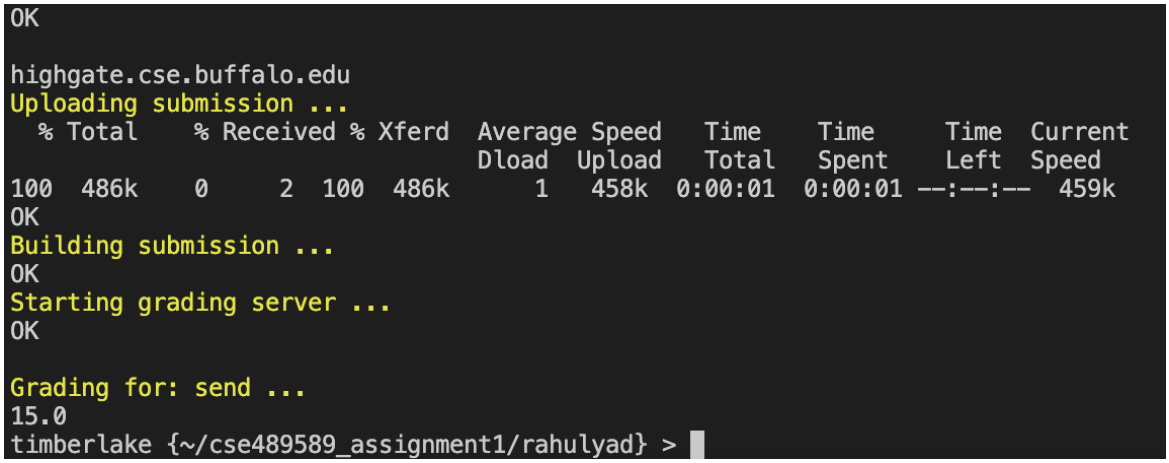
```
Building submission ...
OK
Starting grading server ...
OK

highgate.cse.buffalo.edu
Uploading submission ...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  486k    0     2  100  486k      1   458k  0:00:01  0:00:01 --:--:--  459k
OK
Building submission ...
OK
Starting grading server ...
OK

Grading for: author ...
TRUE
timberlake {~/cse489589_assignment1/rahulyad} >
```

Description:
(Describe your work here...)

- **Which shell executes this command?**
  Both Client and Server have this functionality.
- **Command Syntax**
  AUTHOR
- **Command Functionality / Expected Output**
  Upon executing this command, the code will execute:
  printf("I, %s, have read and understood the course academic integrity policy.\n", your_ubit_name);
  And will print "I, kchand, have read and understood the course academic integrity policy." or "I, rahulyad, have read and understood the course academic integrity policy." based on whatever ubit_name is passed to the print method.

[15.0] SEND (send)

Grader screenshot:



Description:
(Describe your work here...)

- **Which shell executes this command?**
  This command is executed from the client shell.

- **Command Syntax**
  SEND <client_ip> <message>

- **Command Functionality / Expected Output**
  As explained the requirement server is the central point of communication between all the logged-in clients, one logged client(sending client) can send messages to another client(receiving client), if the receiving client has not blocked the sender. The message first reaches the server and the server redirects the message to the respective receiving clients. If any of the receiving clients are logged out then the server writes the message into the buffer of receiving client.

[EVENT]: Message Relayed

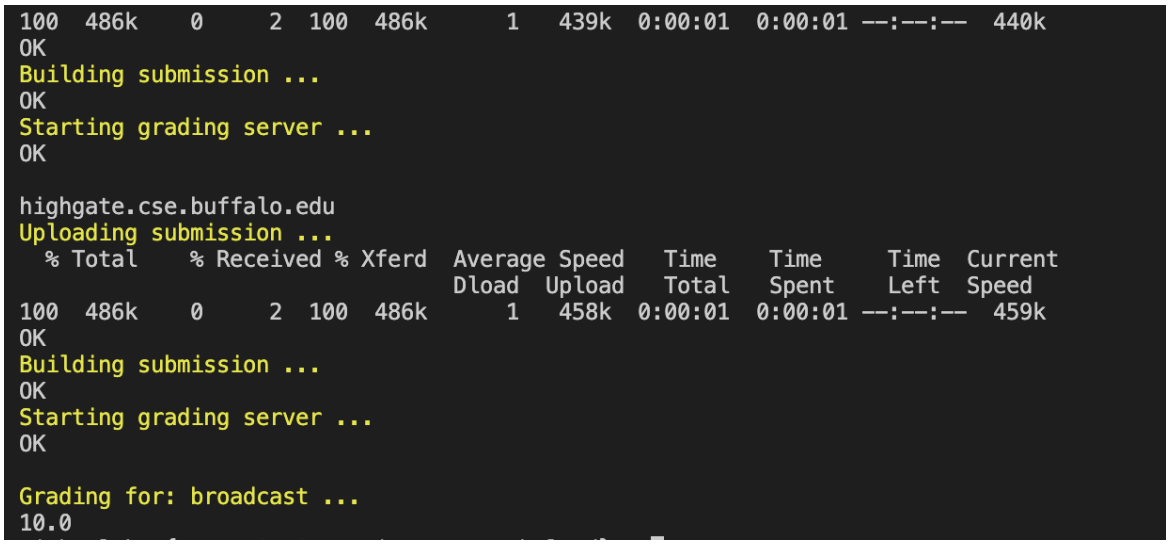
Description:
(Describe your work here...)
The message relayed is printed on the server side when the server sends the message directly to the logged-in client or when the client reads the message from the buffer when it logs back in. Message relay will have three pieces of information sender client IP, receiver client IP and actual message sent.

[EVENT]: Message Received

Description:
(Describe your work here...)
Upon receiving the message client prints logs as Message Received with the sender client's IP and actual message. This will happen in both of the cases when a message is received after client logs back in (received from the buffer) or when the message is received directly when the client was already logged in at the time of sent.

[10.0] BROADCAST (broadcast)

Grader screenshot:

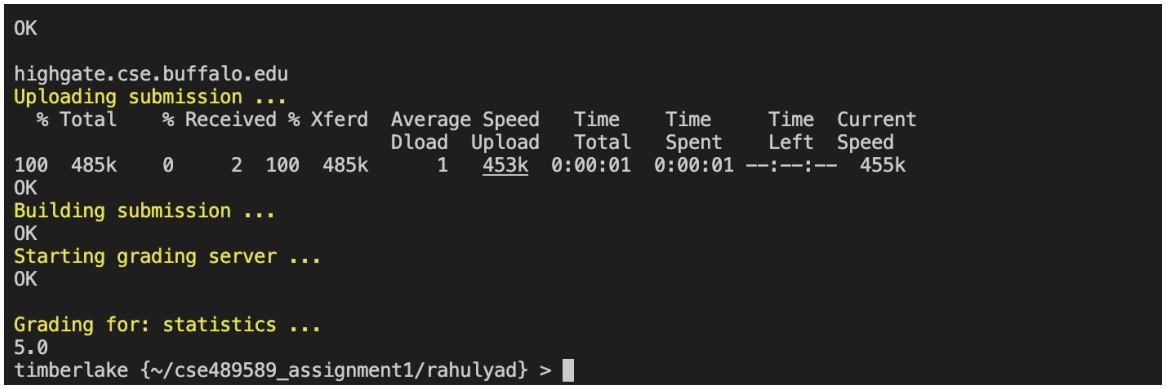


Description:

- **Which shell executes this command?**
  Only a Client can execute this command. The server doesn't have this functionality.
- **Command Syntax**
  BROADCAST <message>
- **Command Functionality / Expected Output**
  When a client executes this command, the message is broadcasted to all other clients. Internal working is as follows: The message is sent to server and server then sends this message to all the logged in clients except the client which is the source of the message. And if any client is logged out, the server stores the message in the respective buffer to send it later whenever the respective client logs back in. Our code also ensures that the message is not sent if the receiving client has already blocked the source client.

[5.0] STATISTICS (statistics)

Grader screenshot:



Description:

(Describe how you implement the LOGIN and LIST commands)

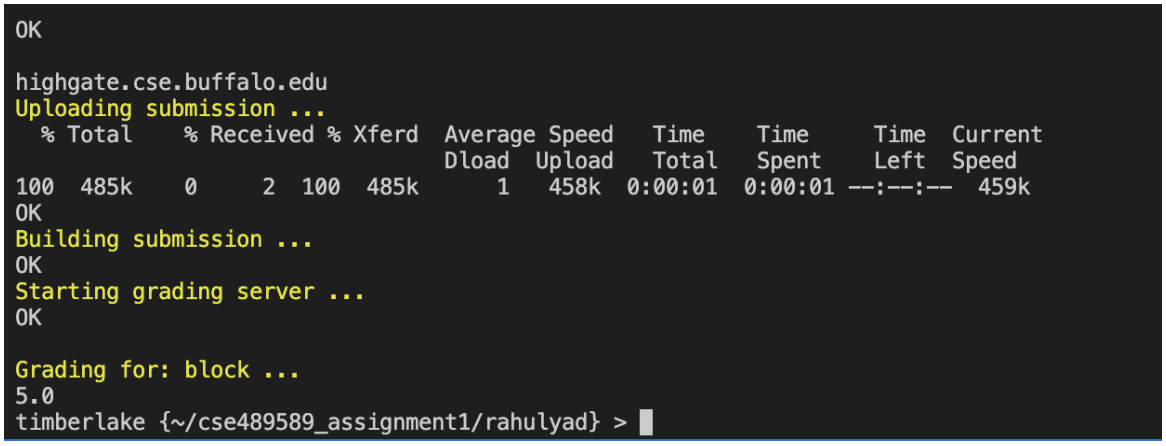
- **Which shell executes this command?**
  Only the Server can execute this command. Client doesn't have this functionality.
- **Command Syntax**
  STATISTICS
- **Command Functionality / Expected Output**

In our code, the server stores the information of the message transaction in a data structure that can be accessed through this command. Upon executing this command, a list of logged-in clients and logged-out clients is printed along with the number of messages received and sent by them. Our code iterates through the data structure and fetches information related to each client and prints it. Before iterating, a simple sort method is called to sort the list in the increasing order of PORT.

- **LOGIN Implementation Explanation**
  First step involves socket initialization via getaddrinfo(), socket() and bind() methods. The client then executes the connect() method to establish a connection. After that a login request is sent from the client side which is then received by the server. Upon receiving the request, server processes the request and sends an acknowledgement along with a list of currently logged in clients.
- **LIST Implementation Explanation**
  List is implemented on both server and client end. Working of both the commands are independent. No exchange of information between server and client happens. Basic logic is to iterate through its local list of logged in clients and print the information. Note that the local list on the client end might not be updated, and to update it, the client needs to execute the REFRESH command.

[5.0] BLOCK (block)

Grader screenshot:



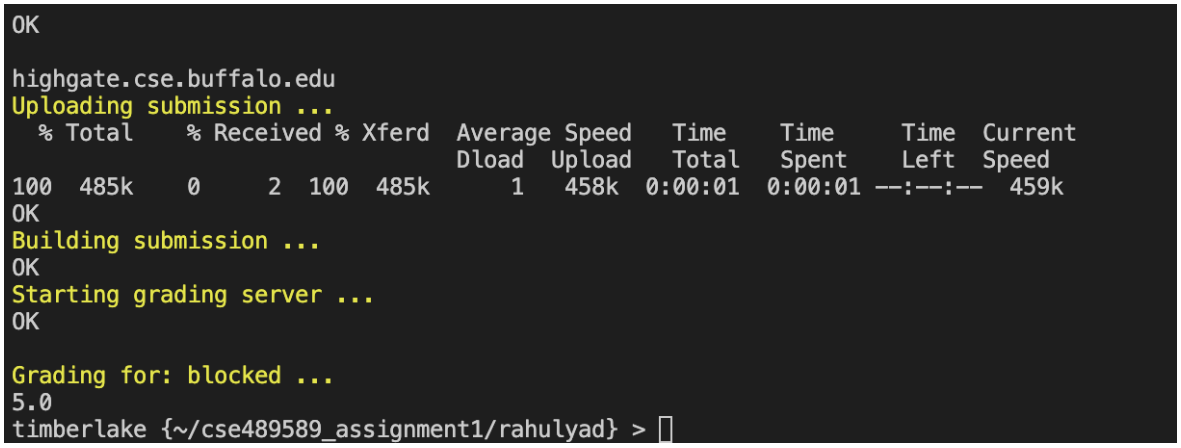
Description:

(Describe your work here...)

- **Which shell executes this command?**
  This command is executed on the client shell, when a client wants to block another client.
- **Command Syntax**
  BLOCK <client_ip>
- **Command Functionality / Expected Output**
  Block functionality allows a client to block incoming messages from another client. We implemented the functionality as per the requirement that if the sending client is blocked by the receiver then the message will not reach the receiving client but the sender will not be notified of the block i.e. the sender will print a success message. When a client is blocked by another client then the client IP is added in the vector (blockeduser) maintained in the struct object of the blocking client. The same information is stored in the client struct object list maintained at the server side. So the client sends the block message to the server so that the server is notified of the blocking update, using this the server add the client IP in the blockeduser vector maintained in the client struct object.

[5.0] BLOCKED (blocked)

Grader screenshot:



Description:

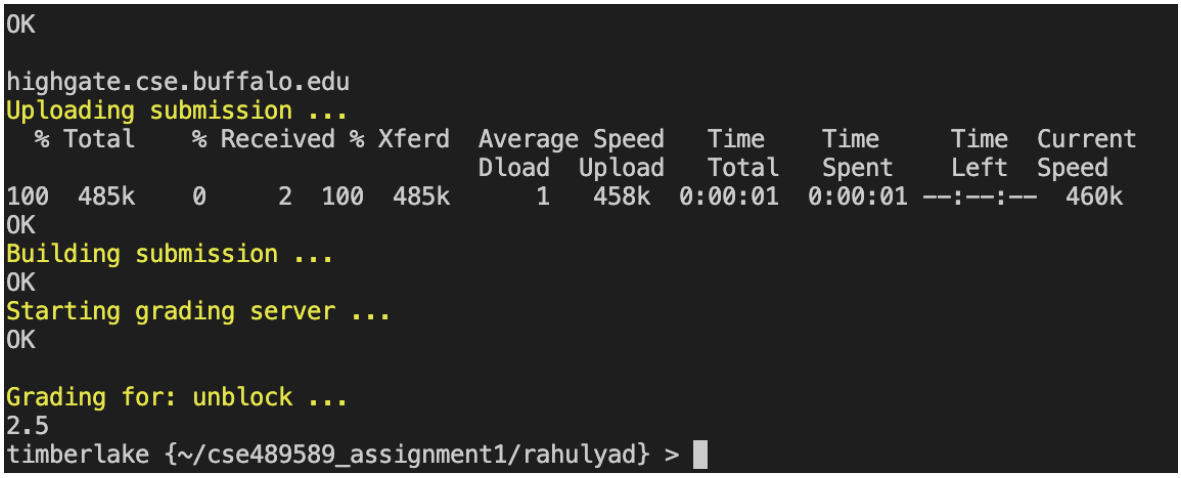
(Describe your work here...)

- **Which shell executes this command?**
  Only the Server can execute this command. Client doesn't have this functionality.
- **Command Syntax**
  BLOCKED <client_ip>
- **Command Functionality / Expected Output**

This command is used by the server to print the list of clients blocked by the provided client_ip. The significance of this feature is that the server blocks any incoming message if the source client is present in the blocked list. Implementation of this feature involves iterating through the data structure and storing the information in a map with key as the <port number> and value as the <SocketObject>. The map is then sorted in ascending order with respect to the key using a custom comparator. In case any one of the socket information can't be fetched, maybe due to the data getting corrupted, the code doesn't print a success message. SUCCESS message is only printed when socket information related to all the clients are fetched properly and server is ready to print.

[2.5] UNBLOCK (unblock)

Grader screenshot:



Description:
(Describe your work here...)

- **Which shell executes this command?**
  This command is executed on the client shell, when a client wants to unblock another client.
- **Command Syntax**
  UNBLOCK <client_ip>
- **Command Functionality / Expected Output**
  Unblock functionality allows incoming messages from another client. An earlier blocked client can be unblocked by using this functionality which will allow incoming messages to be received in the receiving client side. To unblock a blocked client the IP of this client is remove from the array(blockeduser) in the client struct object. The same information is to be updated on the server side list of struct objects, so the client sends the unblock message to the server so that the server is notified of the unblocking update, and removes the client ip from the blockeduser vector present in the client struct object.

[5.0] (buffer)

Grader screenshot:

```
100  486k    0     2  100  486k      1   438k  0:00:01  0:00:01 --:--:--  439k
OK
Building submission ...
OK
Starting grading server ...
OK

highgate.cse.buffalo.edu
Uploading submission ...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  486k    0     2  100  486k      1   457k  0:00:01  0:00:01 --:--:--  458k
OK
Building submission ...
OK
Starting grading server ...
OK

Grading for: buffer ...
0.0
```

Description:
(Describe your work here...)
Command Functionality / Expected Output
The basic functionality of the buffer is to store messages on the server if the destination client has logged out temporarily. The order of the messages to be stored is implemented using a queue like data structure. This functionality also allows to store multiple buffer messages for each client separately by storing information in the respective SocketObject.

Buffer functionality is executed in the below cases:

1) SEND : Client calls send process to send a message to another client but the other receiving list is logged-out and so can receive the message straight away. In this case the server adds this message into the buffer vector present in the receiver client object(struct data structure). This buffer vector will store all the messages for the logged out client except in the case when the sender client is blocked by the receiving client.
2) LOGIN : When a logged out client logs back in, the server knows that the client is already logged in by checking the check_login_status flag. In such cases the server iterates through the respective buffer and then through the list of messages to be sent. The order of messages to be sent is carefully coded to be the same as that of the order of messages stored in the buffer.
3) BROADCAST : Client calls broadcast process to send the same message to all the clients present in the server's client list. All the logged in clients which have not blocked the server will receiving the message straight away, whereas for all the logged out clients(except for the one who has blocked the sender) the server will add the messages to the respective buffers of these clients(maintained as a vector in the struct data structure of the client).

   When the client successfully reads the messages from the buffer, that message is erased and the client shell prints success received and the server at this moment prints a relay message on the server shell.

```
[RELAYED:SUCCESS]
msg from:128.205.36.46, to:128.205.36.33
[msg]:stones msg1
[RELAYED:END]
[RELAYED:SUCCESS]
msg from:128.205.36.46, to:128.205.36.33
[msg]:stones msg2
[RELAYED:END]
[RELAYED:SUCCESS]
msg from:128.205.36.34, to:128.205.36.33
[msg]:euston msg1
[RELAYED:END]
[RELAYED:SUCCESS]
msg from:128.205.36.36, to:128.205.36.33
[msg]:underground msg1
[RELAYED:END]
[RELAYED:SUCCESS]
msg from:128.205.36.34, to:128.205.36.33
[msg]:euston msg2
[RELAYED:END]
[RELAYED:SUCCESS]
msg from:128.205.36.36, to:128.205.36.33
[msg]:underground msg2
[RELAYED:END]
```

Above figure is the relay messages printed on the server side after all the buffer messages are sent to the client that logged in.

[2.5] LOGOUT (logout)

Grader screenshot:

```
100  486k    0     2  100  486k      1   450k  0:00:01  0:00:01 --:--:--  451k
OK
Building submission ...
OK
Starting grading server ...
OK

highgate.cse.buffalo.edu
Uploading submission ...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  486k    0     2  100  486k      1   458k  0:00:01  0:00:01 --:--:--  459k
OK
Building submission ...
OK
Starting grading server ...
OK

Grading for: logout ...
2.5
```

Description:

(Describe your work here...)

- **Which shell executes this command?**
  This command is called from the Client Shell only.
- **Command Syntax**
  LOGOUT <server_ip> <server_port>
- **Command Functionality / Expected Output**

This command is used by a client to log out of the server. In our implementation we are maintaining a string variable in the client struct object which shows the current log-in status of the client. There are two possibilities “logged-in” or “logged-out”, so in case when client is live and then the status is set to “logged-in” at the time of log out we just change the status of client to “logged-out”. In this stage the client is still connected by not active so won’t receive the messages from any other client directly whereas the messages will go in the buffer vector of the client struct object. This happens first on the client side then client sends the logout message to server which updates the list of client accordingly.

[2.0] SEND Exception Handling (exception_send)

Grader screenshot:

```
Starting grading server ...
OK

highgate.cse.buffalo.edu
Uploading submission ...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  486k    0     2  100  486k      1   458k  0:00:01  0:00:01 --:--:--  460k
OK
Building submission ...
OK
Starting grading server ...
OK

Grading for: exception_send ...
2.0
timberlake {~/cse489589_assignment1/rahulyad} >
```

Description:

(Describe your work here...)

Command Functionality / Expected Output

In the SEND process, we checked for two types of exceptions:

1) Invalid IP check:
   We checked if the destination IP address provided is valid or not. this is done by our implemented function ip_exception_check() which returns true for valid IP. Implementation involves checking the size and characters of the ip string by iterating through each character. Failing this check the system will throw an exception, letting the user know that the IP is invalid and they need to correct this.
2) IP address not existing in the local copy of logged in clients:
   Each client maintains a list of logged in clients as a list of struct objects provided by the server to the client at the time of login. This list might be outdated but as per the requirement we throw an expectation if the send process is attempted to any client that

is not present in this local list of struct objects. This is done by our implementation of client_find_object() which gives a struct object associated with the input client IP, if no struct object exists in the local list with the input IP then it returns NULL.

[2.0] BLOCK Exception Handling (exception_block)

Grader screenshot:

```
OK
Building submission ...
OK
Starting grading server ...
OK

highgate.cse.buffalo.edu
Uploading submission ...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  486k    0     2  100  486k      1   459k  0:00:01  0:00:01 --:--:--  460k
OK
Building submission ...
OK
Starting grading server ...
OK

Grading for: exception_block ...
2.0
timberlake {~/cse489589_assignment1/rahulyad} >
```

Description:
(Describe your work here...)
Command Functionality / Expected Output

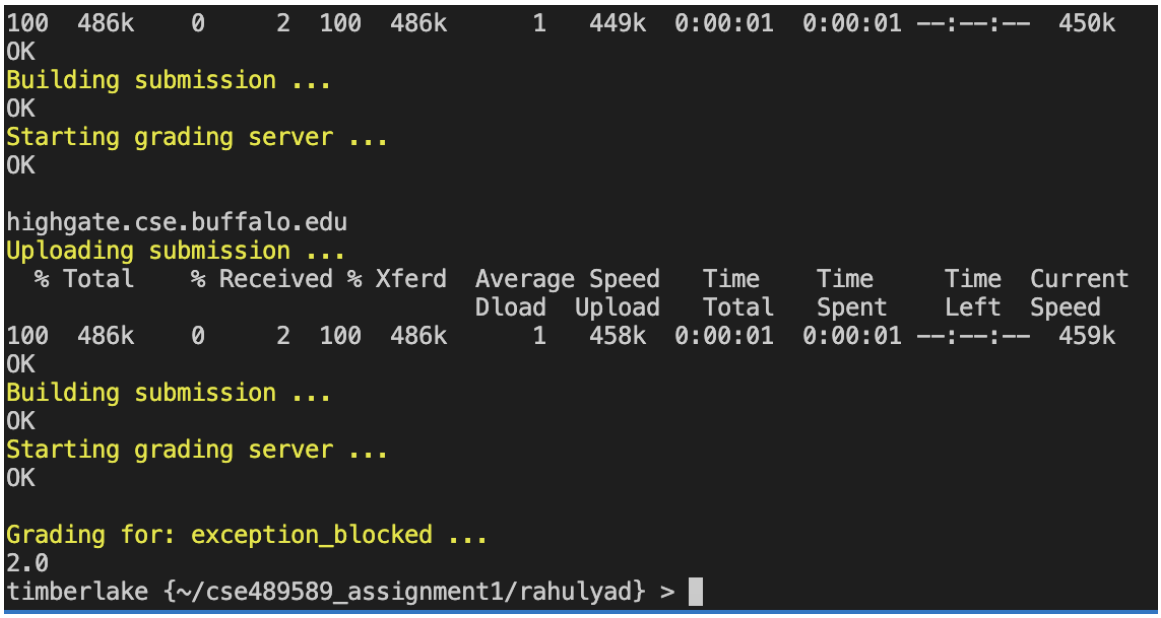
1) Invalid IP address:
   ip_exception_check() method validates the ip and returns true if the ip is valid and returns false if the ip is not valid. Implementation involves checking the size and characters of the ip string by iterating through each character.

2) Valid IP address which does not exist in the local copy of the list of logged-in clients:
   If the ip address doesn't exist in the local copy of the list, the code doesn't call the refresh functionality. Instead it throws an error and lets the user know that the client is unaware of the ip entered by the user.

3) Client with IP address: <client-ip> is already blocked:
   If a user is requesting to execute the BLOCK command multiple times for the same IP, a flag in the data structure will indicate if the client is already blocked or not. In case of an attempt to block an already blocked client, the code will simply throw an exception and let the user know about that.

[2.0] BLOCKED Exception Handling (exception_blocked)

Grader screenshot:



Description:

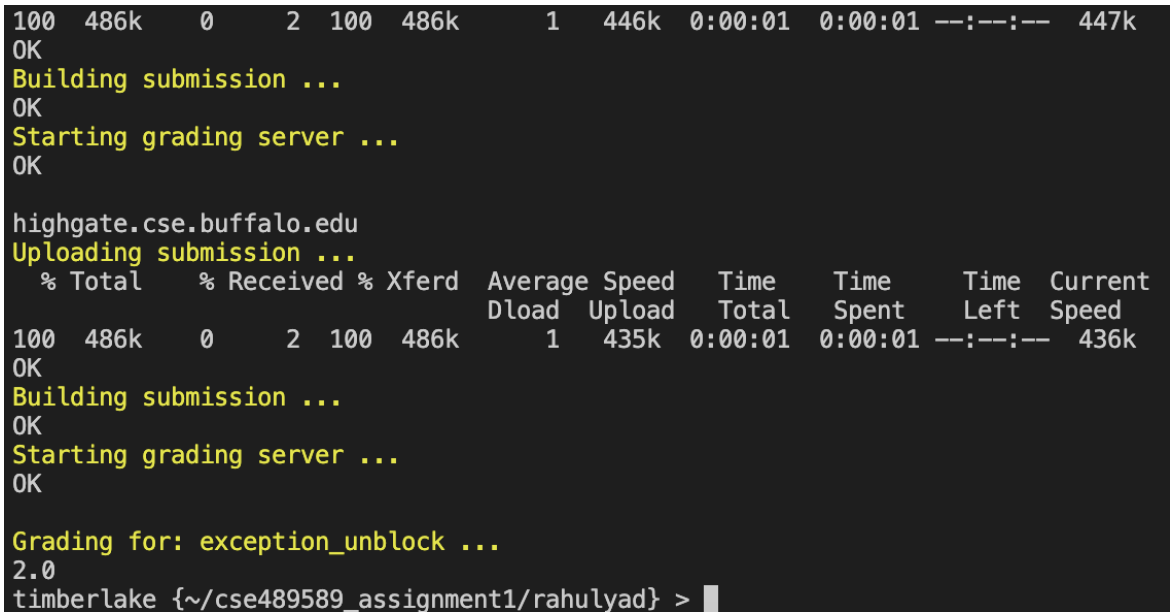
(Describe your work here...)

Command Functionality / Expected Output

1) Non-Existent IP:
   In case any one of the socket information can't be fetched, possibly when the client related to the requested ip has executed the EXIT command, then the server data structure doesn't have any information related to that client. In such a case our code doesn't print a success message. SUCCESS message is only printed when socket information related to all the clients are fetched properly and the server is ready to print.
2) Sort wrt Port Number:
   Implementation of this feature involves iterating through the data structure and storing the information in a map with key as the <port number> and value as the <SocketObject>. The map is then sorted in ascending order with respect to the key using a custom comparator.
3) Valid IP Check:
   ip_exception_check() method validates the ip and returns true if the ip is valid and returns false if the ip is not valid. Implementation involves checking the size and characters of the ip string by iterating through each character.

[2.0] UNBLOCK Exception Handling (exception_unblock)

Grader screenshot:



Description:

(Describe your work here...)

Command Functionality / Expected Output

For Unblock functionality we implemented three types of exceptions:

1) Invalid IP address:
   ip_exception_check() method validates the ip and returns true if the ip is valid and returns false if the ip is not valid. Implementation involves checking the size and characters of the ip string by iterating through each character.
2) Valid IP address which does not exist in the local copy of the list of logged-in clients:
   If the ip address doesn't exist in the local copy of the list, the code doesn't call the refresh functionality. Instead it throws an exception and lets the user know that the client is unaware of the ip entered by the user.
3) Client with IP address: <client-ip> is already blocked:
   If a user is requesting to execute the UNBLOCK command for an IP address which is not already blocked, a flag in the data structure(struct object) will indicate if the client is already unblocked. In case of an attempt to unblock an already unblocked client, the code will simply throw an exception and let the user know about that.