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

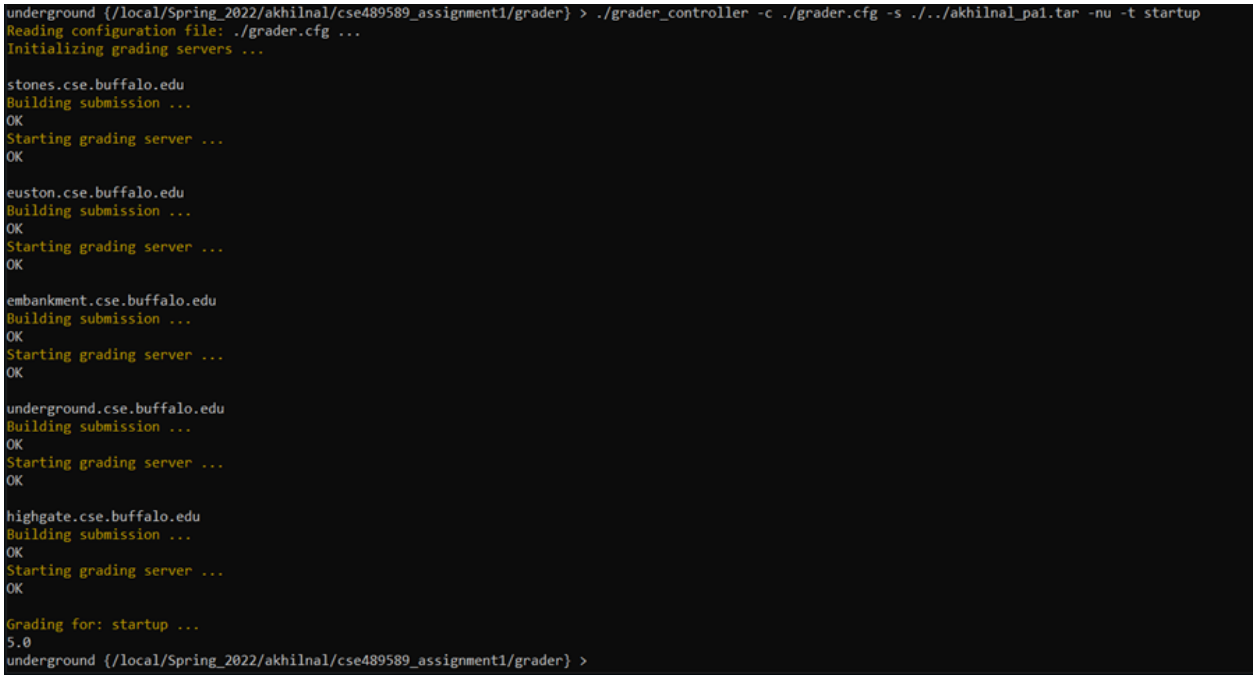
- Name of member 1:
  - UBITName: akhilnal
  - Contributions : Initial environment setup and discussion, implementation of - (startup, author, login, send, block, unblock, exit, statistics), testing and debugging ,report
- Name of member 2:
  - UBITName: spullagu
  - Contributions : Initial environment setup and discussion, implementation of - (ip, port, list, blocked, broadcast, logout, exceptions handling), testing and debugging, report

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)

- For application startup, we made the program accept two arguments – first for ‘s’/’c’ and the next for port number. (For example ./assignment1 s 2000 for server).
- We check to ensure that there are no more/less than 2 arguments and check that the program. We access the command line arguments using argv[index] and arguments using argc.first argument should be either ‘s’ or ‘c’. If either of these are not satisfied, we exit the
- If the input is satisfactory, we move further into the functions OperateInServerMode() / OperateInClientMode() based on the input.



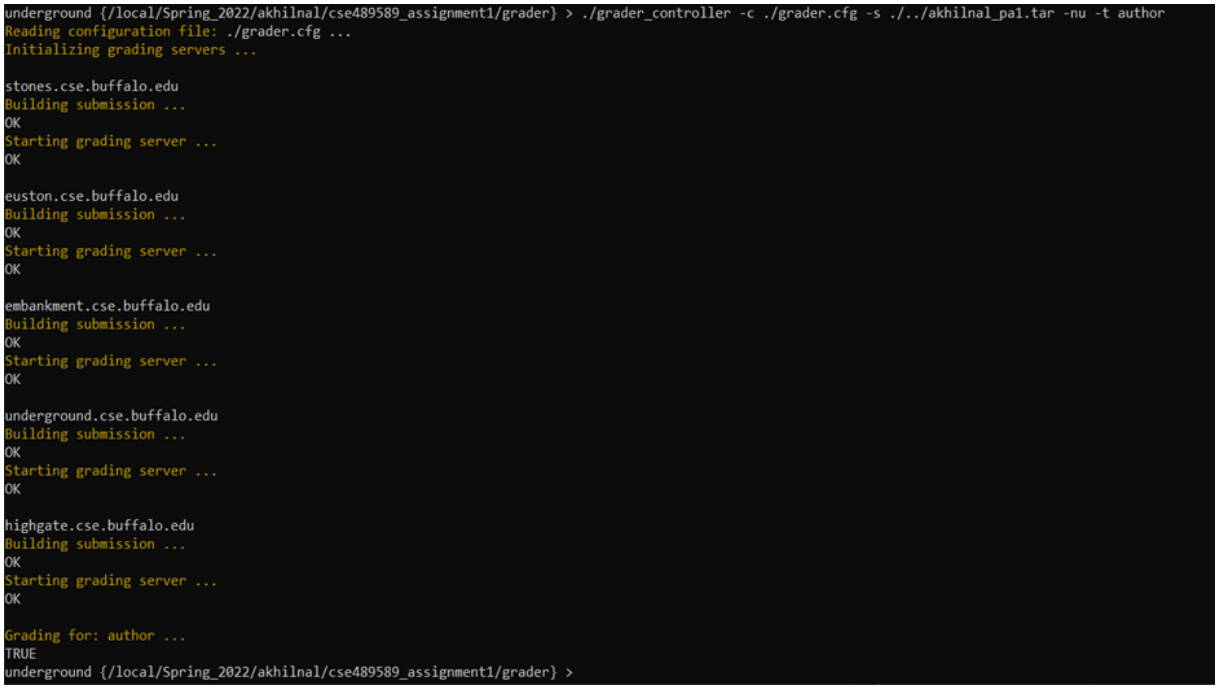
3 - Command for Server and Client

[0.0] AUTHOR

- We use the select() system call in a continuous while loop so that the program can simultaneously accept commands from STDIN as well as socket communication from

the other side (client/server) as given in the demo code. We distinguish it to be standard input/ communication from the other side based on the file descriptor set. If it's from STDIN, then it is a command that the user ran.

- In case of the author command, if the standard input contains the string “AUTHOR”, we display the below string. This command can be executed by both server and client.
  I, akhilnal, have read and understood the course academic integrity policy.



[5.0] IP

(Describe your work here...)

- This command can be executed by both server and client.
- To retrieve the IP address of that process, we create a UDP socket and connect to google’s dns server 8.8.8.8:53 as recommended in the PA1 description.
- Next, we create a connection to the above created socket using connect().
- Then.we retrieve the IP address of the socket with getsockname() and then inet_ntop().

```
underground {/local/Spring_2022/akhilnal/cse489589_assignment1/grader} > ./grader_controller -c ./grader.cfg -s ./../akhilnal_pa1.tar -nu -t ip
Reading configuration file: ./grader.cfg ...
Initializing grading servers ...

stones.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

euston.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

embankment.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

underground.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

highgate.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

Grading for: ip ...
5.0
```

[2.5] PORT

(Describe your work here...)

- This command can be executed by both server and client.
- If the standard input contains the string “PORT”, we print out the port number received from the standard input as a command line argument during startup. (argv[2]).

```
underground {/local/Spring_2022/akhilnal/cse489589_assignment1/grader} > ./grader_controller -c ./grader.cfg -s ./../akhilnal_pa1.tar -nu -t port
Reading configuration file: ./grader.cfg ...
Initializing grading servers ...

stones.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

euston.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

embankment.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

underground.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

highgate.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

Grading for: port ...
2.5
underground {/local/Spring_2022/akhilnal/cse489589_assignment1/grader} >
```

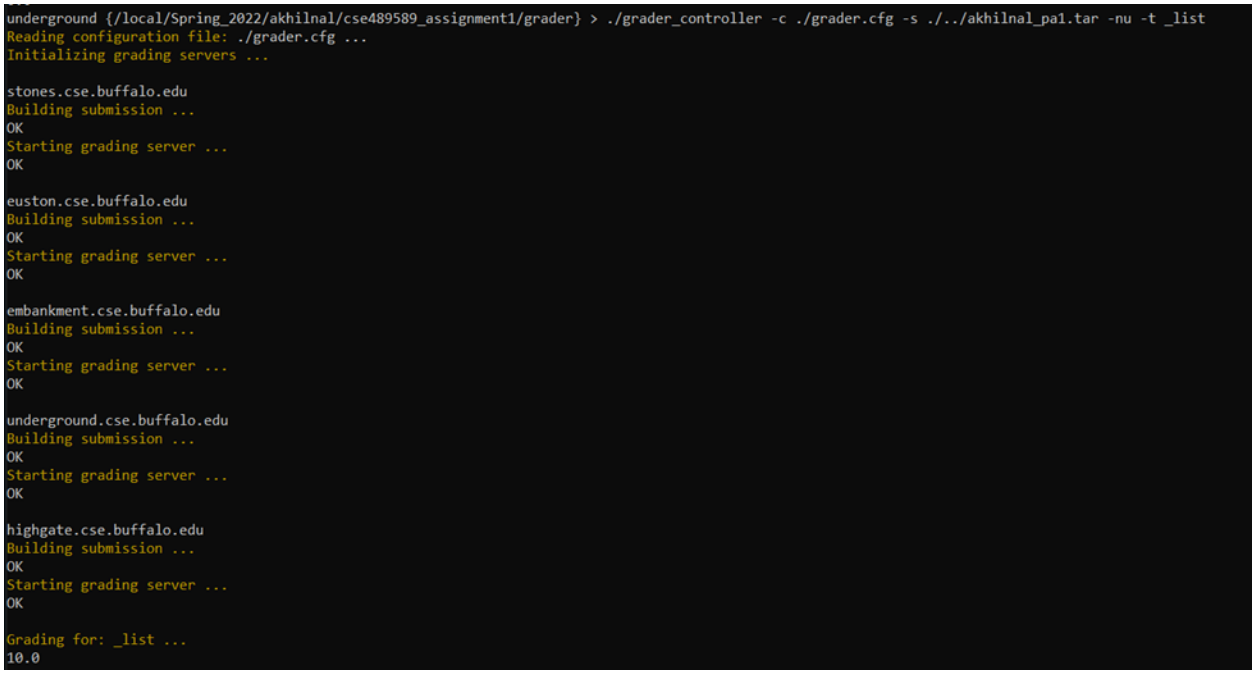
[10.0] LIST

(Describe your work here...)

- This command can be executed by both server and client.
- We maintain each host details in a struct called **hos**t similar to the structure recommended in PA1 recitation, whose declaration is shown below. The list of hosts is maintained in a linked list whose starting pointer is "**hostlisthead**", and a pointer to the next host is stored in the member variable **next_host**.

```
struct host
{
    char hostname[HOST_NAME_MAX_SIZE];
    char ip_addr[INET_ADDRSTRLEN];
    int port_num;
    int num_msg_sent;
    int num_msg_rcv;
    char status[20];
    int fd;
    char blocked[100];
    struct host *next_host;
    bool is_logged_in;
    bool is_server;
    struct message queued_messages[PENDING_QUEUE];
    int queue_size;
} *hostlisthead = NULL;
```

- Each time a client logs into the server, all the details of the server will be stored in the above mentioned linked list. So, if the user requests for the current list of clients, we iterate through the pointer till the last entry and for each entry, we display the details in the order and only the currently logged in clients (using the member **is_logged_in**): **list_id, hostname, ip_addr, port_num**. We defined a function **HandleListCommand()** to print the list.
- To make sure that they are displayed sorted by port number, we maintain the linked list in sorted order during insertion of any new node which we implemented in the function **InsertIntoHostLL().**
- Note that in the case of a client, LIST command can be executed only if the executing client is currently in logged state. (which we maintain on the client side using **this_client_logged_in** boolean variable.)

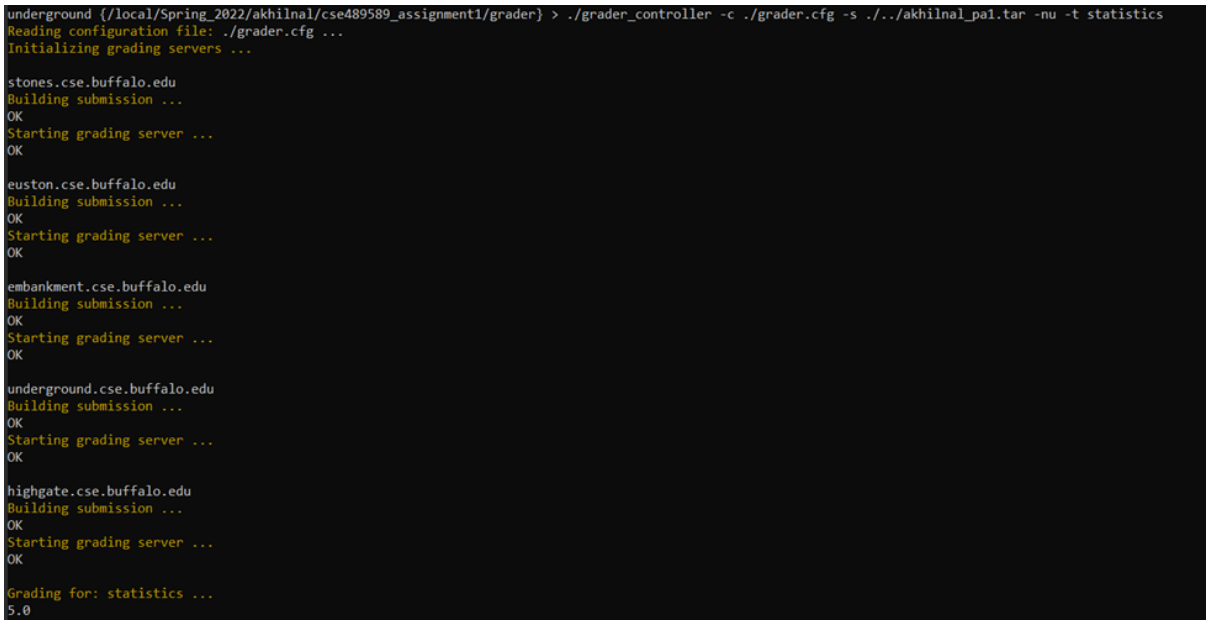


4 - Command/Event for Server

[5.0] STATISTICS

(Describe your work here...)

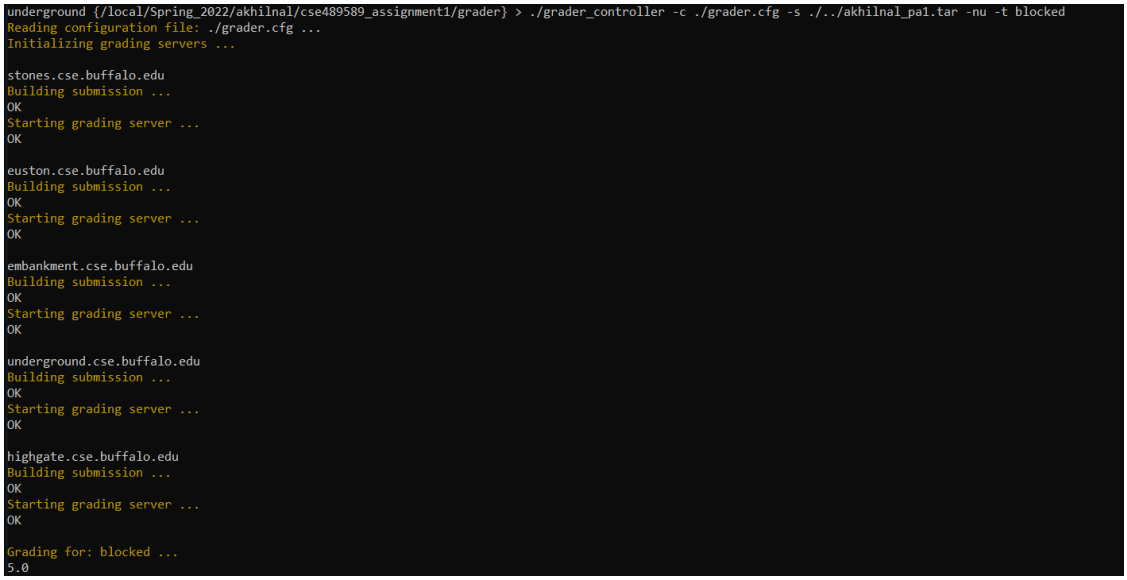
- We defined a function **HandleStatisticsCommand()** to print the statistics.
- Similar to the LIST command, we iterate through the linked list of **host**s maintained and print the details hostname, ip_addr, num_msg_sent, nuv_msg_rcv and status. Since they are already stored in sorted order, it outputs them sorted order by port.

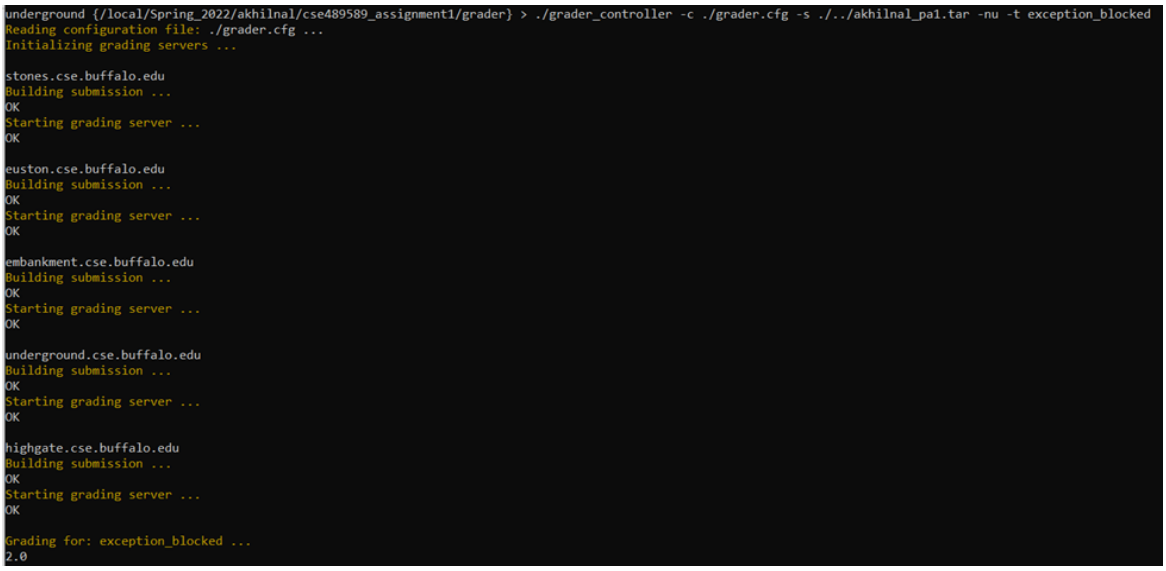


[7.0] BLOCKED <client-ip> + Exception Handling

(Describe your work here...)

- On the server side, to maintain the IP address blocked by each host, we store those IP addresses in a member variable **blocked** in the corresponding **host** struct object.
- They are stored in a continuous char array delimited by a space. For example if a client blocks two other clients with IP addresses A and B, the **blocked** member variable for this **host** object will contain a string 'A B'.
- So when the BLOCKED command is executed, we first handle exceptions by checking if the input IP is a valid IP string - using **ValidateIPaddress ()** and also check if the IP is in the maintained linked list of hosts - using **IPCurrentlyInClientList()** .
- Once the above checks are passed, we print in sorted order by port, the IP address stored in the **blocked** member variable.
- Below are the grader outputs for blocked and exception_blocked.





[EVENT]: Message Relayed

(Describe your work here...)

- This event is printed every time the server relays a message from one client to another both in cases of unicast (SEND command) and broadcast (BROADCAST command) messages.
- We implemented this in the functions SendCommandServerSide() and BroadcastCommandServerSide().
- We print the sender ip, receiver ip and message string. In case of broadcast message, we print 255.255.255.255 as the receiver ip.

5 - Command/Event for Client

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

(Describe your work here...)

- **On the client side** - First when the client executes the login command, we first extract the arguments i.e server-ip and server-port and handle exceptions using the functions ValidateIPaddress() and ValidatePortNumber().
- Then we connect to the server ip and port using socket() and connect() and add the obtained fd to the master list.
- We then send the client's port number to the server using send().
- Then we also communicate to the server that the client has executed the login command. We send all such internal communication from client to server using the structure **internalctos** whose definition is shown below.

```
struct internalctos
{
    int commandid;
    char ip[INET_ADDRSTRLEN];
    int port;
    char msg[BUFFER_SIZE];
};
```

- We fill the ip and port of this client in the struct and send the object to the server using send() system call.
- **On the server side** - while continuously looping through the sockets after select system call, if a new client is requesting connection , it understands that a client executed the login command, so it accepts the connection using accept() and gets an fd for this client's socket.
- Then we use recv() on the obtained file descriptor to receive the port number sent by the client. We also receive an object of type **internalctos** using recv() system call.
- We then extract the IP address and hostname of the client using inetntop() and getnameinfo().
- Now, we check in the server maintained linked list of hosts to check if the client is a new client or an old one and based on that we insert a new host or update the existing host's fd accordingly.
- Also, on the server side, we send some information back to the client using an internal communication structure **internalstoc** whose definition is shown below which is used for all internal communication from server to client.

```
struct internalstoc
{
    char message_type[20];
    listitem clientlist[5];
    message pending_msgs[PENDING_QUEUE];
    int queue_size;
    int clientcount;
    char message_text[BUFFER_SIZE];
    char from_ip[INET_ADDRSTRLEN];
};
```

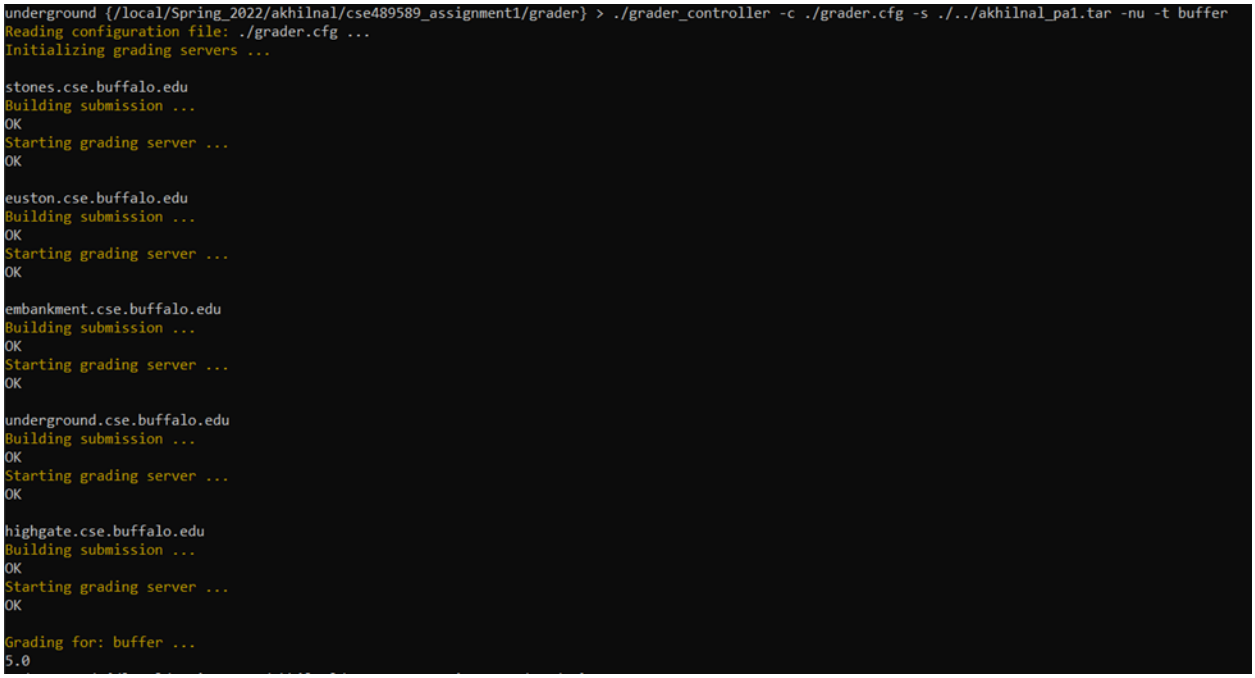
- In **internalstoc** object**,** we send the currently logged in list of hosts packed in **listitem** objects and buffered messages if any, for this client stored in the corresponding host object (packed in **message** objects) back to the client. Below are the structure definitions of **listitem** and **message .**
-

```
struct message
{
    char text[BUFFER_SIZE];
    char from_ip[INET_ADDRSTRLEN];
    bool is_broadcast;
};
```

```
struct listitem
{
    char hostname[HOST_NAME_MAX_SIZE];
    char ip[INET_ADDRSTRLEN];
    int port;
};
```

- **On the client side** - we receive this server sent object. From the object we extract the client list and update the client side maintained linked list of hosts. We also extract the buffered messages if any and execute RECEIVED events for all such messages.
- Below are screenshots of grader confirming the ” exception_login” and “buffer” cases.

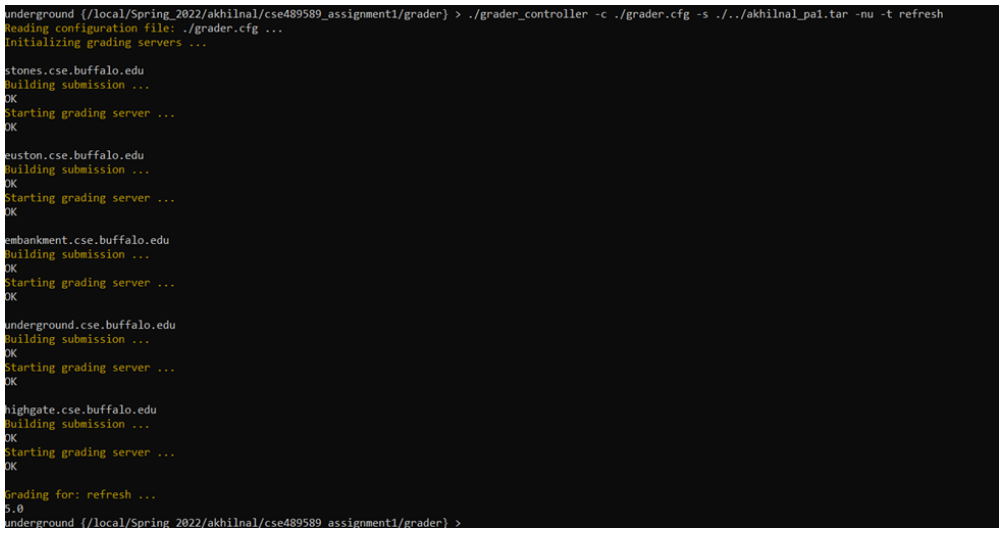
```
underground {/local/Spring_2022/akhilnal/cse489589_assignment1/grader} > ./grader_controller -c ./grader.cfg -s ./../akhilnal_pa1.tar -nu -t exception_login
Reading configuration file: ./grader.cfg ...
Initializing grading servers ...

stones.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

euston.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

embankment.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

underground.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

highgate.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

Grading for: exception_login ...
2.0
```



[5.0] REFRESH

(Describe your work here...)

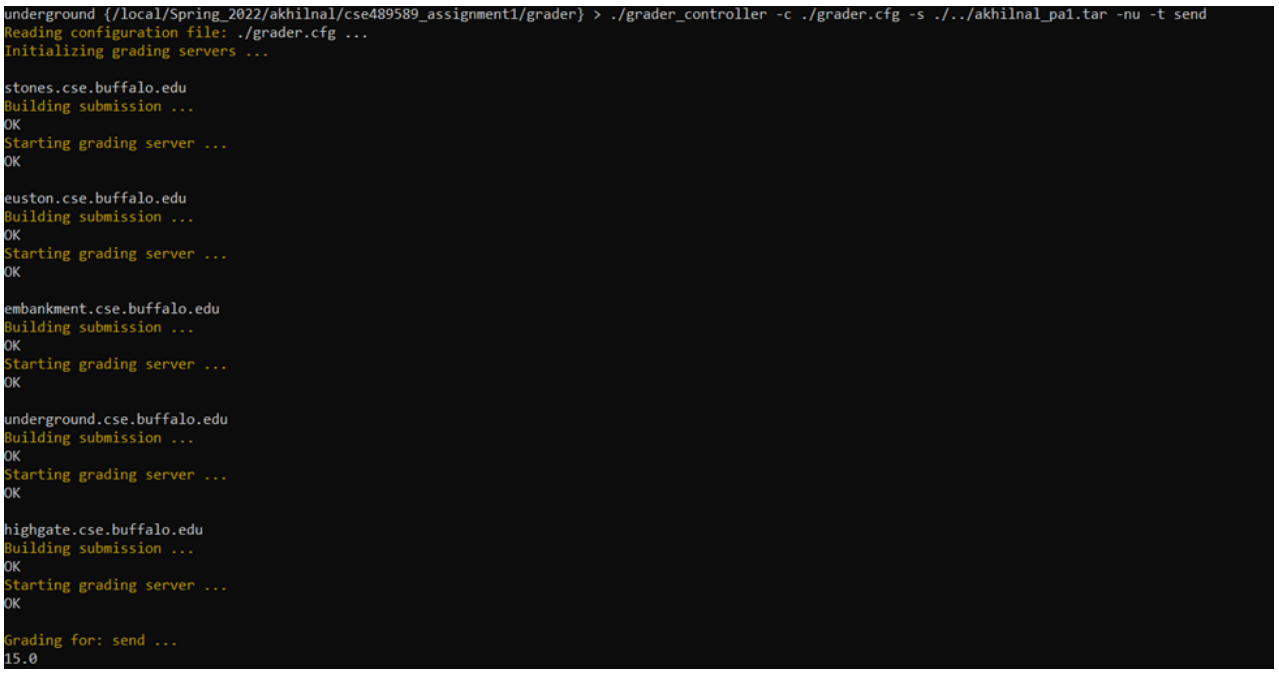
- To obtain the list of updated logged in list of hosts, we call a send() function with **internalctos** with command id 2, for which the server responds back with the updated list of clients in an **internalstoc** object.
- Then we first clear the client side maintained host list and update it with the list of hosts received from the server.
- Note the above steps works only if the server is currently in logged state.

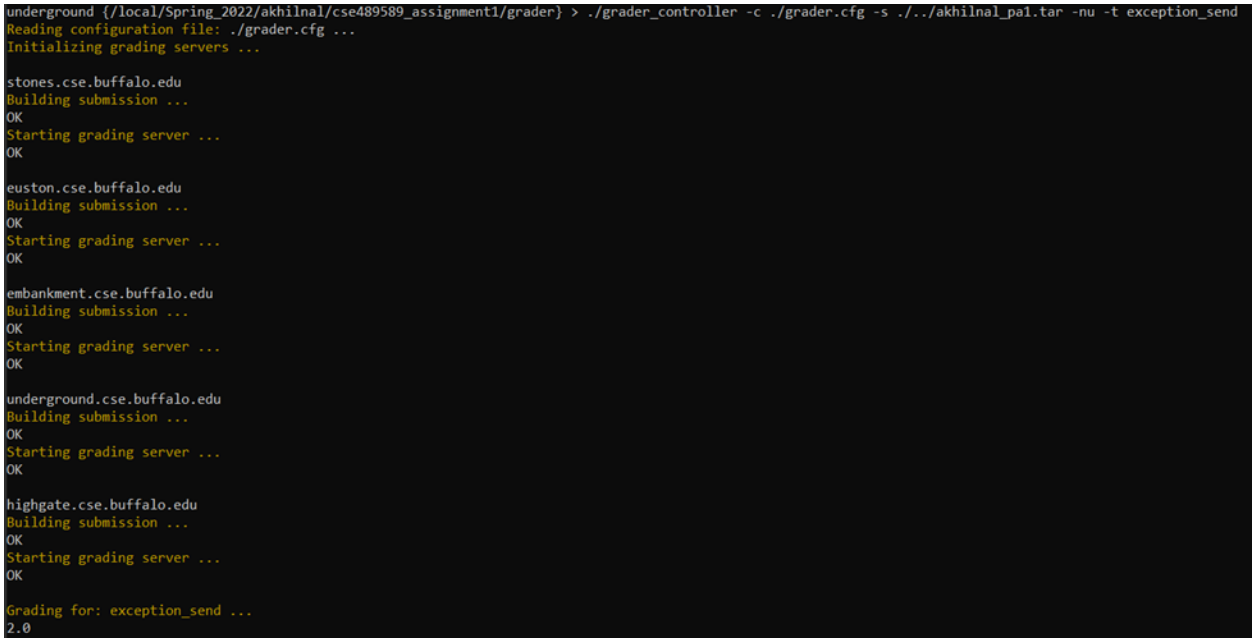


[17.0] SEND <client-ip> <msg> + Exception Handling

(Describe your work here...)

- **On the sender client side** -. We initially handle exceptions - we validate if the IP address is valid and also if the ip exists in the list of current logged in hosts
- Then, we call send() with the **internalctos** object containing destination IP address, message text and command id 1 indicating send command.
- **On the server side** - we locate the host object for the sender and receiver.
- We check if the receiver is blocked by the sender and if yes, we don't relay the message. We just increment num_msg_sent for the sender.
- If it's not blocked, we check if the receiver is logged in. If yes, we send an **internalstoc** object to the receiver client fd with the message text, sender IP and message_type = 'unicast-message', execute RELAYED event and increment num_msg_sent for sender and num_msg_rcv for receiver. If it's not logged in, we buffer the message in the **queued_messages** member variable of the receiver **host** object.
- Then we send a string 'send-complete' or 'send-failed' in case the send fails to the sender client.
- **On the sender client side -** Then we call recv() to receive the status of the send operation from the server. If the status message reads as "send_complete", then send operation is successful, else an error has occurred. It accordingly logs command success/failed.



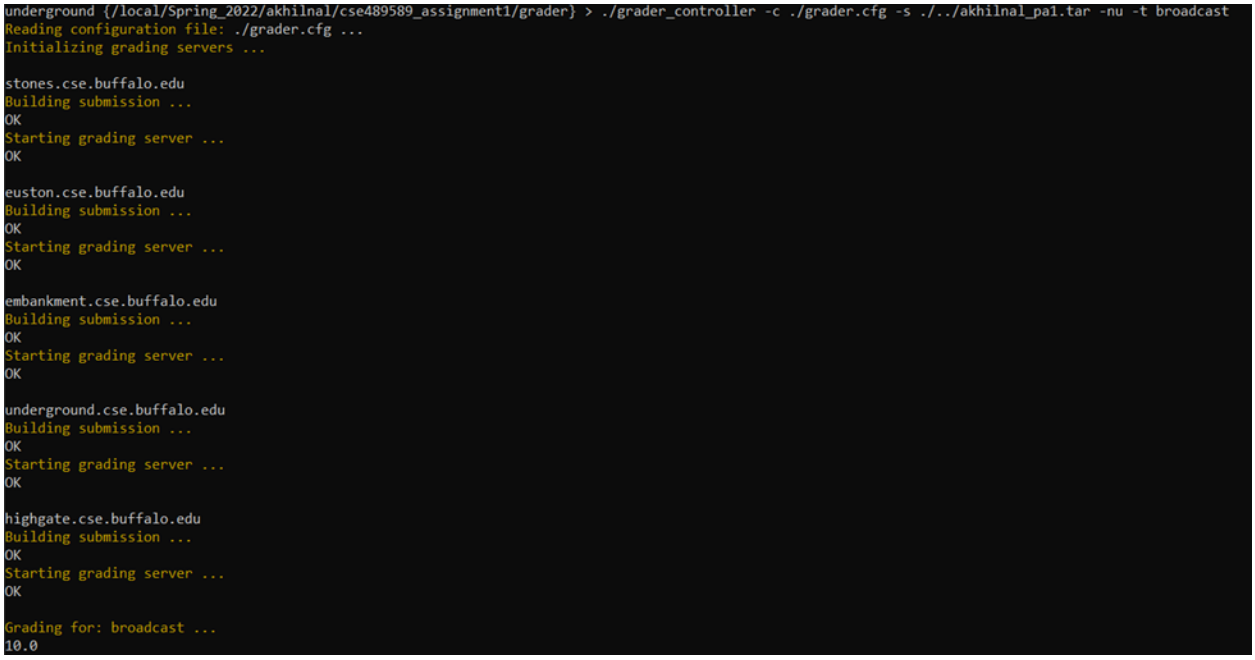


[10.0] BROADCAST <msg>

(Describe your work here...)

- A broadcast message request works in the same way as unicast messaging but here, a message is broadcasted to all existing hosts, except the client which initiated the broadcast. For this purpose, make use of the IP address **255.255.255.255**.
- **On the sender client side -** we call send() with the **internalctos** object containing message text and command id 3 indicating broadcast command.
- **On the server side -** the server will initially determine the sender host details.
- Next the server starts sending the message to each host. Note that each time the server checks if the current host is the sender itself and only if it is not a sender, then we move for the next steps to make sure the broadcast message is not sent back to the sender.
- Similar to unicast messaging, for each host, we verify if the sender is blocked or not by the receiver and if not blocked, we lookout whether the receiver is logged in currently or not. If logged in, the message will be sent to the receiver in an **internalstoc** object with message_type = broadcast-message to the receiver and also increases num_msg_rcv for each receiver .If not, message is stored in the buffer of each such receiver so that receiver can get them once it logs back.
- If all the hosts have been sent the broadcasted message, then the sender is confirmed with ‘bcast_complete’ status message. Else ‘bcast_failed’ message will be sent back to the receiver.
- **On the sender client side -** Then we call recv() to receive the status of the send operation from the server. If the status message reads as ‘bcast_complete’ , then the

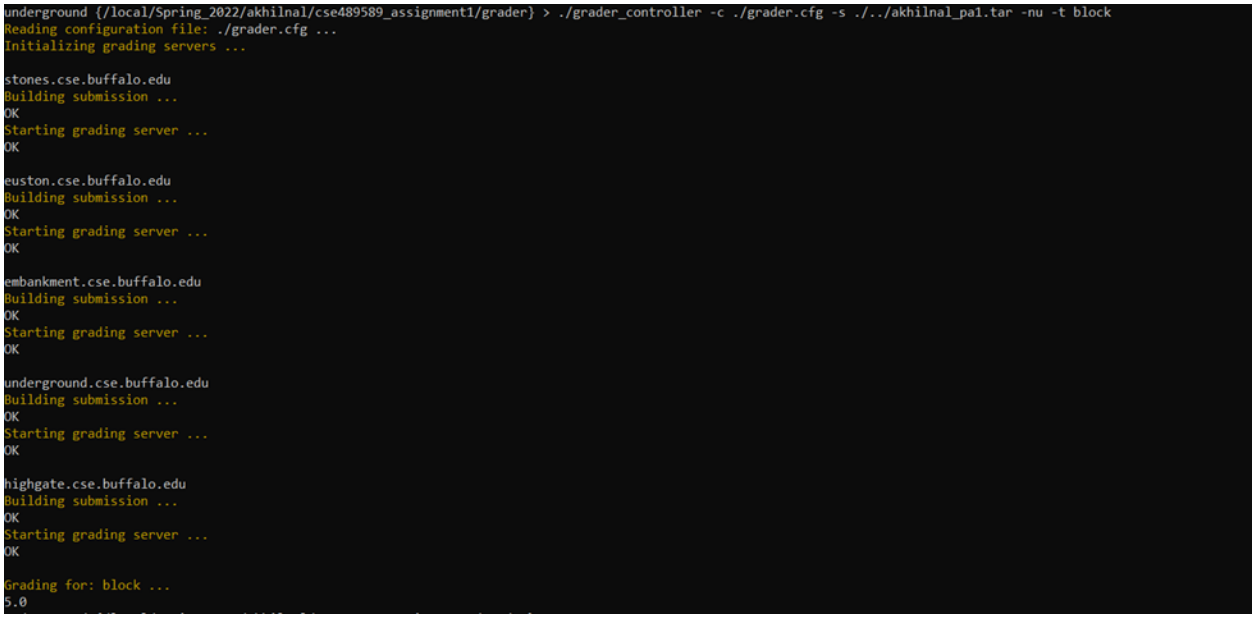
broadcast operation is successful, else an error has occurred. It accordingly logs command success/failed.

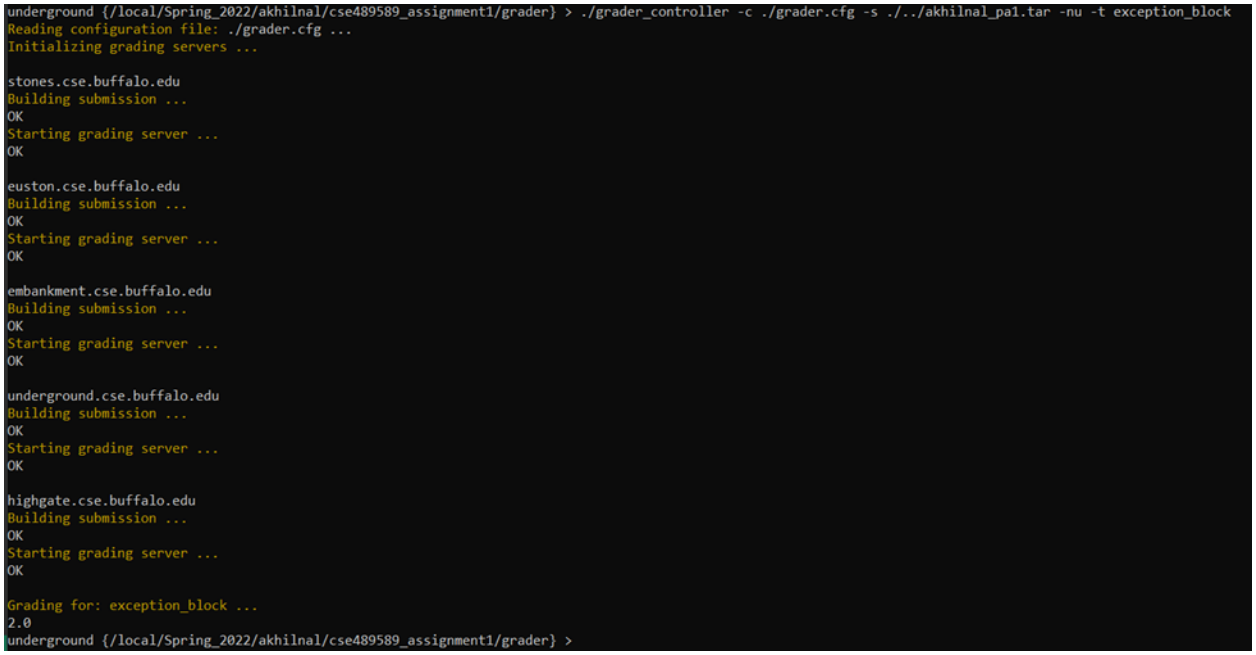


[7.0] BLOCK <client-ip> + Exception Handling

(Describe your work here...)

- **On the client side** - When the client requests to block another client, we initially validate if the IP address of the client to be blocked is valid or not. We also verify if the client-IP does exist in the current list of hosts. We also check if the client is NOT already blocked by the sender which we check in the client side maintained vector of blocked ips, **c_blocked_ips** . If all the preceding conditions satisfy, we move to the next steps.
- We send the block request to the server in an **internalctos** object with command id 4.
- **On the server side** - The server first locates the blocker's **host** object and appends the **blockee_ip** to the string **blocked** in the **host** object. Then it sends a 'success' status message to the client.
- **On the client side -** then the client receives back the status response from the server and logs command success/failed.





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

(Describe your work here...)

- **On the client side** - When the client requests to unblock another client, we initially validate if the IP address of the client to be blocked is valid or not. We also verify if the client-IP does exist in the current list of hosts. We also check if the client is already

blocked by the sender which we check in the client side maintained vector of blocked ips, **c_blocked_ips** . If all the preceding conditions satisfy, we move to the next steps.

- We send the unblock request to the server in an **internalctos** object with command id 5.
- **On the server side** - The server first locates the blocker's **host** object and removes the **blockee_ip** from the string **blocked** in the **host** object. Then it sends a status message to the client.
- **On the client side -** then the client receives back the status response from the server and logs command success/failed.

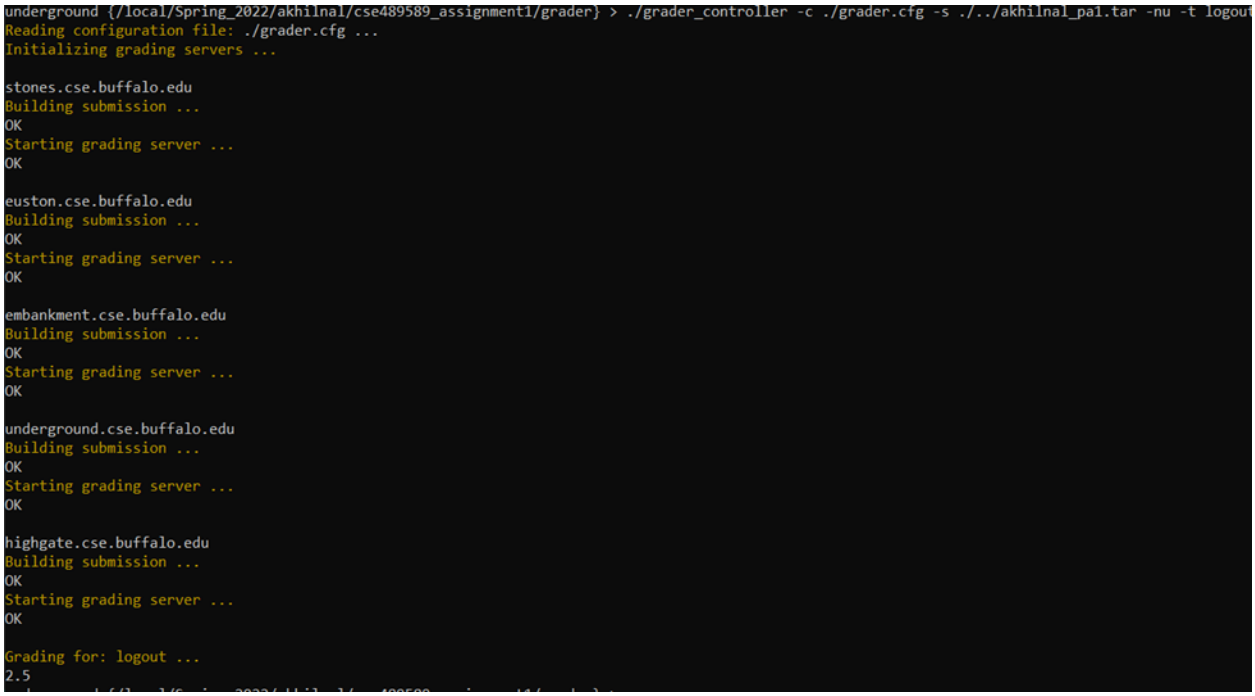
```
underground {/local/Spring_2022/akhilnal/cse489589_assignment1/grader} > ./grader_controller -c ./grader.cfg -s ./../akhilnal_pa1.tar -nu -t unblock
Reading configuration file: ./grader.cfg ...
Initializing grading servers ...

stones.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

euston.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

embankment.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

underground.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

highgate.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

Grading for: unblock ...
2.5
```

```
underground {/local/Spring_2022/akhilnal/cse489589_assignment1/grader} > ./grader_controller -c ./grader.cfg -s ./../akhilnal_pa1.tar -nu -t exception_unblock
Reading configuration file: ./grader.cfg ...
Initializing grading servers ...

stones.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

euston.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

embankment.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

underground.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

highgate.cse.buffalo.edu
Building submission ...
OK
Starting grading server ...
OK

Grading for: exception_unblock ...
2.0
```

[2.5] LOGOUT

(Describe your work here...)

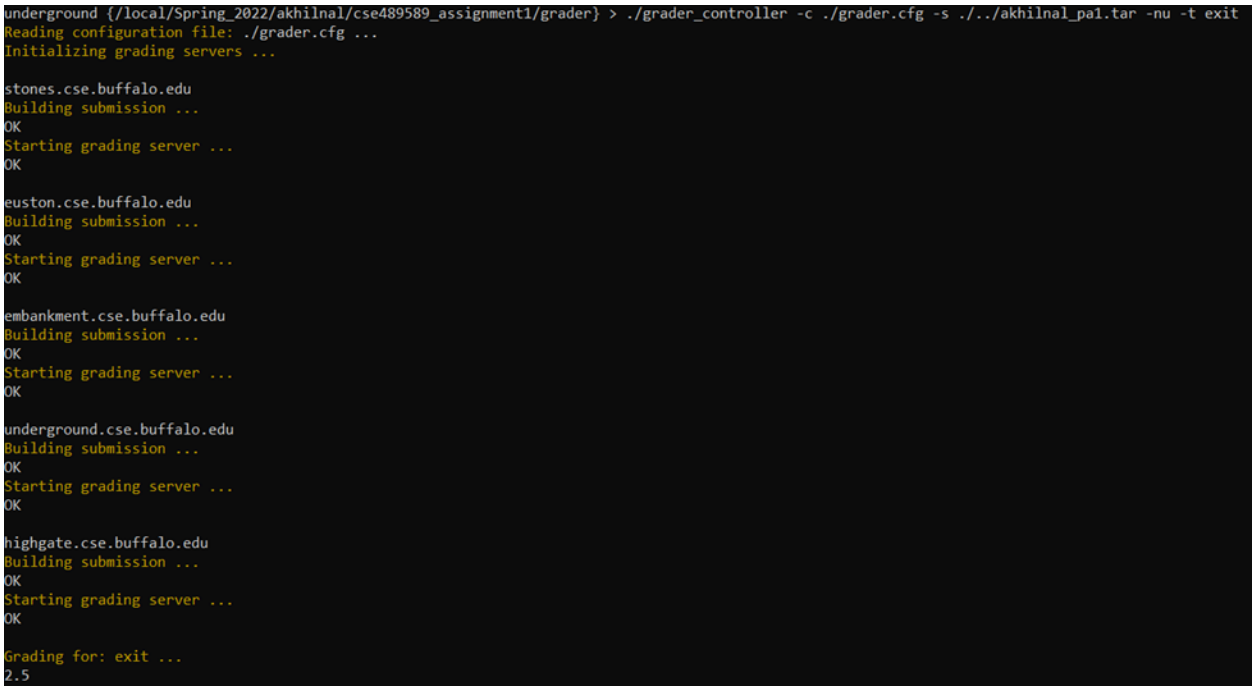
- **On the client side** - the client sends an **internalctos** object with command id 6 to the server.
- **On the server side -** the server first identifies the client host object in the linked list, sets the **is_logged_in** status of the host client to False and status to 'logged-out'.
- Apart from that, the server also closes the socket and removes the socket descriptor from the master list.
- **On the client side -** the client closes the server socket and sets **this_logged_in** to false.
- Note that even after logging out, the client will be able to accept LOGIN, EXIT, IP, PORT, and AUTHOR commands. The block/unblock status, the statistic counters also will not be changed.



[2.5] EXIT

(Describe your work here...)

- If the client wants to completely exit, it sends an internalctos command with commandid 6 to the server to log this client out. Finally, a statement, "exit(0)" will be used to completely exit.
- Note that, exit will totally terminate the client program, the client will not be able to perform any activity and no other client can communicate with the one that has already exited.



[EVENT]: Message Received

(Describe your work here...)

- On the client side, when the selected socket is not STDIN, it means that it received some communication from the server, which must be a message.
- The communication will be of the form **internalstoc** object and the message type will be 'unicast-message' or 'broadcast-message'. In this case, the client logs the RECEIVED event with the sender ip and message text.

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

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

(Describe your work here...)

Below is the submission screenshot showing a successful submission.

```
underground {/local/Spring_2022/akhilnal/cse489589_assignment1} > submit_cse589 akhilnal_pa1.tar
Submission of "akhilnal_pa1.tar" successful.
underground {/local/Spring_2022/akhilnal/cse489589_assignment1} > date
Thu Mar 24 12:13:06 EDT 2022
underground {/local/Spring_2022/akhilnal/cse489589_assignment1} >
```