CPE464 - Programming Assignment #2
Chat Program (client and server) using TCP
Due **at 11:59 pm Monday April 26, 2021**

The programs (cclient[1], server) are due **at 11:59 pm Monday April 26, 2021.** If your program is late you will lose 15% per day. The last late day you may work on this to receive any grade is Thursday, April 29th after that you will receive a grade of 0.

Extra Credit: If you turn in the program (**all** functionality working) by Friday, April 23rd you will receive 20% extra credit.

You are to write a chat program using **TCP**. To do this you will implement two programs in C (or C++). The first is **cclient** (the chat client program). The second program is the **server** which will act as a router/forwarder between all of the clients. The clients connects to the server and then using this connection transmits data to server. The server forwards the received data to the designated client. All communications between the clients goes through the server. The server acts as a forwarding agent for the clients. You can think of this configuration as a logical star with the server as the center.

These programs (server, cclient) must work on and between unix1-5.

You must include a file called README (all caps) with the files you handin. The first line of the file must include your full name (first last) and your lab section time (9am, noon, 3pm or 6pm).

You will create your own user level packet and then use TCP to send the packet. All packets you create will start with the same first 3 bytes[2]. Your packet header's first 3 bytes (I call this the "chat-header") will start with:

a. 2 bytes PDU length in network order (this is the length of the entire user level packet which includes the chat-header, handles and any payload)
b. 1 byte flag (see below for flag values)

Your header must be in the above format. One of the grading tests for your program is verification of header format.

Details:

1) **cclient:** This program connects to the server and then sends and receives message from the server. The client takes the servers IP address and port number as run time parameters. You will also pass in the client's handle (name) as a run time parameter.

   The client uses the "$: " as a prompt for user input (so you output this to stdout). The client program supports four commands (case **insensitive**). See below for discussion of the commands.

[1] I call it cclient for chat client. There is a chat program already under Linux and I don't want any confusion that could mess up your testing.

[2] You are creating a user level header. The normal TCP/IP/Ethernet headers are created by the kernel – you will not access these headers.

Running cclient program has the following format:[1]

$: cclient *handle server-name server-port*

handle[2]	is this clients handle (name), max 100 characters
server-name:	is the remote machine running the server
server-port:	is the port number of the server application

Commands:

a. %M num-handles destination-handle [destination-handle] [text]

- Must also work for %m (lower case)
- These are message packets, where "handle" is the name(s) of the client(s) you wish to talk to.
- Example: %M 1 handle1 Hello how are you
  - Sends the message to the user with then name: handle1
- Example: %M 2 handle5 handle6 Another message to send
  - Sends the message to two users with the names handle5 and handle6

- This command allows you to send a message to one or more other handles.
  - The num-handles can be a number between 1 and 9. This number says how many handles follow.
- The client will only send one message packet to the server. This message packet contains the list of destination handles (see message packet format below). It is the servers responsibility to forward the message to each handle.
- This command is used to send a text message (the text is optional, if no text is provided then send an empty message which when received is processed/outputted like a new line (\n)
- The maximum length of the text message is 200 bytes. A message ends when the user hits return. If a message is too long, break it up into multiple message packets of at most 200 bytes of user data. You will send a new message packet (new normal header and the rest of the header information) for each packet. The receiver of these messages will receive each message individually and process them as individual messages.
- To output a message on the screen (STDOUT) you should put each message on a newline, then output the senders handle, a colon with a space ": ", and then the message with a new line at the end (remember to put the prompt "$: " back out).
- If the server responds that the handle does not exist, the client should output the error message: Client with handle <put handle name here> does not exist. You will receive one packet from the server for every non-existed client sent in a message packet.
- **The format for a message packets is**:
  - Normal 3 byte chat-header (message packet length (2 byte), flag (1 byte))
  - 1-byte containing the length of the sending clients handle
  - Handle name of the sending client (no nulls or padding allowed)
  - 1 byte specifying the number of destination handles contained in this message.
  - For each destination handle in the message

[1] To receive a non-zero grade your cclient/server must be able to send messages between 3 cclients.

[2] Every client has a handle. You can think of this as the name of that client. The handle must start with a letter (a-z, A-Z).

        - 1 byte containing the length of the handle of the destination client you want to talk with[1]
        - Handle name of the destination client (no nulls or padding allowed)
    - Text message (a null terminated string)

b. %B [text]
    - Must also work for %b (lower case)
    - Broadcast a message to all clients
    - Example: %B This is a broadcast message
        - Sends the message to all of the other users.

    - This (%B) is worth at most 10% of the program grade. Do this last after everything else is working. This is required to be eligible to receive the extra credit.
    - The broadcast packet should not be sent back to the sender
    - The maximum length of the text message is 200 bytes. A message ends when the user hits return. If a message is to long, break it up into multiple message packets of at most 200 bytes of user data. You will send a new message packet (new normal header and the rest of the header information) for each packet. The receiver of these messages will receive each message individually and process them as individual messages.
    - The format for a broadcast packet is:
        - Normal 3 byte chat-header (length, flag) (flag = 4)
        - 1 byte containing the length of the sending clients handle
        - Handle name of the sending client (no nulls or padding allowed)
        - Text message (a null terminated string)
        - This packet type does not have a destination handle since the packet should be forwarded to all other clients.

c. %L
    - Must also work for %l (lower case)
    - Prints out a list of the handles names currently known by the server
    - The print order of the handle names is not defined (you can print them out in any order based on what is easiest for you). No sorting required.
    - The flow for the %L command is
        - Client tells server it wants a list of the handles (flag = 10)
        - Server responds with a flag =11 packet. This packet includes the chat-header, then a 32 bit number (in network order) containing the number of handles at the server.
        - Server then responds with (one per handle) flag = 12 messages. You will send one flag = 12 message for each handle. See discussion of flag 12 below.
        - Client prints out the number of handles and then each handle on a new line.

d. %E
    - Must also work for %e (lower case)
    - This is the exit command.

[1] Handles are packed into the message. There is a 1 byte length and then the characters of the handle. The handle cannot be null terminated in the packet. You must use the 1 byte length to process the handle.

- Client should notify the server that it is exiting (flag = 8).
- After receiving ACK (flag = 9) for this from server, the client terminates cleanly.

2) **server**: The server acts as a packet forwarder between the cclients. The main goal of the server is to forward messages between clients. You can view the server as the center of a logical star topology. This program is responsible for accepting connections for clients, keeping track of the clients' handles, responding to requests for the current list of handles and forwarding messages to the correct client.

   The server program does not terminate (you kill it with a ^c).

   The server program has the following format:

   $: server [*port-number*]

port-number	an optional parameter If present it tells the server which port number to use[1] If not present have the OS assign the port number

3) Required Flag values and packet formats. The flag byte in the normal 3-byte packet header dictates the packet format as described below. Your packets MUST match these formats.

   - Flag = 1
     - Sent from cclient to server - Client initial packet to the server
     - Format: chat-header, then 1 byte handle length then the handle (no nulls/padding)
     - cclient blocks until it receives a packet with Flag = 2 or Flag = 3

   - Flag = 2
     - Sent from server back to cclient - confirming a good handle
     - This is a positive response to flag = 1
     - Format: chat-header
   - Flag = 3
     - Sent from server back to cclient - Error on initial packet (handle already exists)
     - This is an error response to a flag = 1
     - Format: chat-header
     - When the cclient receives this it prints out an error message and terminates

[1] Our testing uses this port-number. If your code does not support a port-number parameter for the server you program will fail our tests.

- Flag = 4
  - Sent from cclient to all other clients via the server - Broadcast packet
  - See format above under the %B (broadcast) command.
- Flag = 5
  - Sent from cclient to another cclient via the server. This is a message (%M) packet
  - See format above under the %M (message) command.
  - Sending cclient does **not** block waiting for a response from the server after sending the %M message
- Flag = 6 (no longer used)
- Flag = 7
  - From server to cclient - Error packet, if a destination handle in a message packet (flag = 5) does not exist.
  - You should send one Flag=7 packet for each handle in error (so if a message contains 4 destination handles and 2 of them are invalid the server will send back two different Flag=7 packets).
  - Format: chat-header then 1 byte handle length then handle of the destination client (the one not found on the server) as specified in the flag = 5 message.
- Flag = 8
  - Client to server when the client is exiting (client program cannot terminate until it receives a flag = 9 from the server)
  - Client should not block while it is waiting for the Flag = 9 response packet. Client needs to process any incoming message until the Flag = 9 response arrives.
  - Client cannot exit until the Flag = 9 packet arrives.
  - Format: chat-header
- Flag = 9
  - Server to client, ACKing the clients exit (flag = 8) packet
  - Format: chat-header
  - After receiving this the client can terminate
- Flag = 10
  - Client to server, client requesting the list of handles. (So %L command)
  - Format: chat-header
  - After sending the Flag = 10, client must continue processing any other messages that come in until the Flag = 11 message arrives.

- Flag = 11
  - Server to client, responding to flag = 10, giving the client the number of handles stored on the server.
  - Format: chat-header, 4 byte number (integer, in network order) stating how many handles are currently known by the server.
- Flag = 12
  - Server to client, immediately follows the flag = 11 packet. Each flag =12 packets contain one handles currently registered at the server. There will be one Flag =12 packet per handle.
  - Format: chat-header, then one handle in the format of: 1 byte handle length, then the handle (no null or padding).
  - The handles are sent one right after the other. The server will not send any other packets until all the handles have been sent. (chat-header with flag =12, 1 byte handle len, handle, chat-header…)
  - After the Flag = 12 message header arrives you will only receive the handle data (so flag = 12 packets) until all handles have been sent. No other message should be intertwined with sending the handles from the server to the client. You do not need to process any STDIN input while processing the Flag = 12 messages.
- Flag = 13
  - Server to client. This packet tells the client the %L is finished.
  - Immediately follows the last flag = 12 packet
  - Format: chat-header

Server Handle Table (maps handle to socket number) requirement:

To implement this program the server needs to maintain some type of data structure that maps the client's handle name to the server socket number for that handle. This table must be able to grow dynamically as the number of users increases. The table does not need to shrink in size.

The data structure and accessor functions for the server's handle table MUST be in a separate file and the only code in that file (.c and .h) is code for managing the handle table. You can think of this as a handle table object or a handle table API. While you may have some global variables in the handle table .c file you may NOT directly access any of the handle table data structure directly from other C files. In order to access or manipulate your handle table (e.g. add, lookup) you must use accessor functions that are defined in your handle table .c and .h files.

Other Program Requirements:

1) Do not use any code off the web or from other students. You may use code that I have given you this quarter.

2) Your programs must work on and between unix1-5.

3) Your main() function can only be 15 lines or less. No exceptions unless you talk with me in advance. Your main function should coordinate work, not do work.

4) You should try not duplicate code between the client and the server or within the client/server. Use a separate file for commonly used functions. Duplication of code may result in lost points.

5) The server should never block waiting on a response from the client. In this respect the server is stateless, it receives connections and accept()s them, receives packets and processes them but after either of these actions it just goes back to select. (e.g. after the server accept()s a TCP connection request from a client the server may NOT block waiting on a flag = 1 packet from that client.)

6) Use of usleep(), sleep() or any other delay functionality (other than select()) will result in a 0. Your program should not require this. Using select() with a timeout value other than NULL will result in a zero.

7) You cannot use a busy loop to cause your program to block. Use of a busy loop will result in a grade of 0. A busy loop is a loop that just uses the CPU for almost no reason until an event happens. Instead, you must use the select() call on the client and the server to block your program until a packet/STDIN is ready for receving/reading.

8) If you think you need to use sleep() or a busy loop you are doing something wrong! Stop and ask for help – and redesign your program.

9) The server does not change a message or broadcast packet. The server receives these packets, looks up the destinations and forwards the original packet it received to these destinations.

10) All calls to the select() system call must use a time value (timeout) of NULL. This value must be in the actual call to select(). We will look at all of your select() calls to confirm this. A non-null value is the equivalent to using sleep() and will result in a 0 for the assignment.

11) You cannot make your sockets non-blocking or use your sockets in any non-blocking fashion. You must use select() with a NULL time value to control the flow of your program on both the client and the server. If you think you have a good reason for making your sockets non-blocking talk to Prof. Smith first.

12) You need to check that there are enough runtime parameters and print out an error message if they are not correct (segfaulting because of this is not acceptable.)

13) A segfault is never the correct answer to any of life's problems.

14) malloc() is not the answer to all of life's memory needs. You may not use malloc()/calloc() (or new in C++) in the cclient program. On the server, malloc()/calloc()/new may only be used in support of the data structure you use for your table of client handles.

15) Your program should NOT malloc() for every packet or every STDIN message. While you need to use malloc (or similar) for your server's handle-name storage data structure, you do not need to use dynamic memory allocation for packets and STDIN since both have a fixed maximum size and only one packet/message is being processed at any one time. Poor use of memory allocation will result in lost points.

16) Your **server** must use a dynamic data structure (mallac/realloc, tree, link list, hash…) for storing the handles. You pick the data structure. It cannot be fixed size. You can use a built in data structure (more likely in C++) or make your own. You cannot use someone else's code.

17) The input from STDIN will never exceed 1400 at any one time. This includes the command, handles and text message.

18) A client needs to be able to send (%M) to itself.

19) You must use the header and packet format given above. (length, flag…)

20) In the chat-header the 2 byte length field must be in network order.

21) Handles are limited to 100 characters. If a handle (e.g. in a %m, %b, initial cclient setup) is longer than 100 characters you should just print out an error message and ignore the command (or terminate the client if it is the initial setup).

22) The maximum message length (command, handles and text) is around 1400 characters. If more than 1400 characters are entered, the programs behavior is up to you. Some options include:

    a. Print out an error message and ignore all input until you hit the next newline. (Completely ignore the command – don't send anything.)

    b. Send the message as normal (breaking it up into 200 byte text messages)

    c. Send out the first 1400 bytes and ignore the rest

    d. Segfaulting is not correct! Your cclient must correctly process the next correctly formatted command.

23) Name your makefile: Makefile. Your makefile MUST provide a **clean** target that deletes all of the .o and executable files.

24) For testing, we should only need to type *make* once. If you decide to use a tar file, you must provide a separate makefile that will untar your code and then compile it. If our scripts fail because your makefile fails to build your program the grader will become upset.

25) The command names **cclient** and **server** must be used. Also the run-time parameters should be in the order given. See the provide Makefile for help on this. Since we may be using a script to make and execute your code, all executable names and parameters must be as listed. If your program fails to run with my script you will lose 20%.

26) To implement the server you must use the select() function. You cannot use threads or processes in order to handle the multiple clients. (We will use threads/processes in the next program).

27) See the flag MSG_WAITALL that works with the recv() system call. This flag forces recv() to wait until an entire message (based on the length field in the recv() call) has been received.

28) If the server receives 0 bytes on a socket, this means the other end has closed the socket and ended (usually means someone ^c the client). In this case the server should clean up after the client (e.g. remove the client handle from its handle list) but NOT try to send a response to the client since at this point the client is already gone. If the cclient receives 0 bytes on a receive, the client should print out the message "Server Terminated" and then end cleanly.

29) The server should never terminate. It must clean up any state information maintained for a client once the client terminates (e.g. remove the client's handle from its table, close the socket). To kill the server we will use ^c.

30) Regarding the %M with multiple destinations:

If the user enters the same destination handle multiple times for the same message, the server should forward the message to the client with that handle multiple times.

For example: $M 2 client1 client1 Hello

This would result in client1 seeing the text message "Hello" twice.

31) cclient error messages

i. Handle already in use: <handle name>[1]
ii. Invalid handle, handle longer than 100 characters: <handle name>
iii. Invalid handle, handle starts with a number[2]
iv. Server Terminated[3]
v. Client with handle <handle name> does not exist[4].
vi. Invalid command[5]
vii. Invalid command format
viii. You should make up error messages for other cases such as where the number of handles on a %M is invalid (e.g. > 9).

(continued on next page)

[1] Case where the server already has someone with that handle

[2] Error that may occur when you are starting up a new cclient

[3] Case where the server terminated but the client was still running

[4] Case when sending a message and the destination client does not exist

[5] User enters an invalid command

32) A run of these programs would look like the boxes below:

Note 1 – Your output may not match the output below exactly. That is ok. The error messages should match but sometimes printing out of the prompt ("$:") will result is different looking output.

Note 2 – in this example all machines are in the domain .csc.calpoly.edu. You will probably need to specify the whole machine name when connecting a client to a server (e.g. unix1.csc.calpoly.edu)

In a window – using unix1:

bash$: server 54321
Server is using port 54321

In another window – using unix3:

bash$: cclient client2 unix1 54321
$:
client1: Hello how are you?
$:
client4: Hello back
$: %M 1 client1 Fine, thanks.
$:
client1: Going Home
$: %B Hello to all
$:

In another window – using unix1

bash$: cclient client1 unix1 54321
$: %M 1 clientX Hello how are you?
$:
Client with handle clientX does not exist.
$: %M 1 client2 Hello how are you?
$:
client4: Hello back
$:
client2: Fine, thanks.
$: %M 3 client2 client4 clientX Going Home
Client with handle clientX does not exist
$:
client2: Hello to all
$: % E
bash$:

In another window – using unix5

bash$: cclient client2 unix1 54321
Handle already in use: client2
bash$ cclient client4 unix1 54321
$: %L
Number of clients: 3
client1
client2
client4
$: %M 2 client2 client1 Hello back
client1: Going Home
$:
client2: Hello to all
$: % E
bash$: