CPE464 - Programming Assignment #2
Chat Program (client and server) using TCP
See Canvas for Due Dates

See Canvas page for due dates.

If your program is late, you will lose 10% per day. After three late days you will receive a grade of 0.

Extra Credit: If you turn in the program (**all** functionality working) by the extra credit date you will receive 20% extra credit (see canvas for extra credit date).

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[1]. Your packet header's first 3 bytes (I call this the "chat-header") will start with:

a. 2 bytes PDU length <u>in network order</u> (this is the length of the entire user level packet which includes the chat-header, handles and any payload, you created this in your lab program)[2]
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 cclient must support the following commands %M (send message), %C (multicast a message), %B (broadcast a message), %L (get a listing of all handles from the server).

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] You are creating a user level header. The normal TCP/IP/Ethernet headers are created by the kernel – you will not access these headers.
[2] The sendPDU() function you created for lab already puts this 2-byte length field on the PDU. So, you have implemented this already. Use your sendPDU() function and the length part of the PDU is done.

The client terminates with a ^c. The server must deal with the clients terminating via the return value from the recv() function. DO NOT handle SIGINT in either the client or server.

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 destination-handle [text]
- Must also work for %m (lower case)
- These are message packets, where "handle" is the name of the client you wish to talk to.
- Example: %M handle1 Hello how are you
  - Sends the message to the user with then name: handle1
- 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 including the null. A message ends when the user hits return. If a message is too long, break it up into multiple message packets of at most 199 bytes of user data (plus one null byte – so 200 bytes total). 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.
- **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 with the value 1 (one) to specify that one destination handle is contained in this message. %M can only send to 1 other handle. This field with the value 1 (one) is to allow the %M and %C packets to have the same basic format and therefore simplify the processing of the %M and %C packets.
  - The destination handle in the format:
    - 1 byte containing the length of the handle of the destination client you want to talk with[3]

---

[1] To receive a non-zero grade your 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 will start with a letter (a-z, A-Z) and will contain only letters and numbers.

[3] 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.

- - Handle name of the destination client (no nulls or padding allowed)
    - o 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
    - o 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 including the null. A message ends when the user hits return. If a message is too long, break it up into multiple message packets of at most 199 bytes of user data (plus one null byte – so 200 bytes total). 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:
    - o Normal 3 byte chat-header (length, flag) (flag = 4)
    - o 1 byte containing the length of the sending clients handle
    - o Handle name of the sending client (no nulls or padding allowed)
    - o Text message (a null terminated string)
    - o This packet type does not have a destination handle since the packet should be forwarded to all other clients.

c. %C num-handles destination-handle destination-handle [destination-handle] [text]
  - This is a multicast (send a message to more than one but not necessarily all)
  - Must also work for %c (lower case)
  - This command allows you to multicast (send) a message to multiple (2 to 9) other handles.
  - Example: %C 2 handle5 handle6 Another message to send
    - ▪ Sends the message to 2 (two) other users with the names handle5 and handle6
  - num-handles must be between 2 and 9. (if you want to send to 1-user use %M).
  - The client will only send one message packet to the server. This message packet contains the list of destination handles (see %C packet format below). It is the servers responsibility to forward the message to each handle.
  - The maximum length of the text message is 200 bytes including the null. A message ends when the user hits return. If a message is too long, break it up into multiple message packets of at most 199 bytes of user data (plus one null byte – so 200 bytes total). 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.
  - Output packet info/message to the screen similar to %M

- If the server responds that a 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 multicast packet.
- **The format for a multicast packets is**:
  - o Normal 3 byte chat-header (length in network order, flag) (flag = 6)
  - o 1-byte containing the length of the sending clients handle
  - o Handle name of the sending client (no nulls or padding allowed)
  - o 1 byte specifying the number of destination handles contained in this message. This number should be between 2 and 9.
  - o For each destination handle in the message
    - 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)
  - o Text message (a null terminated string)

d. %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
  - o Client tells server it wants a list of the handles (flag = 10)
  - o 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.
  - o 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.
  - o Client prints out the number of handles and then each handle on a new line.

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 from clients, keeping track of the clients' handles, responding to requests for the current list of handles and forwarding messages to the correct client. [2]

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
                    If not present have the OS assign a random port number

---

[2] The server does NOT modify the %m (unicast), %b (broadcast) or %c (multicast) PDUs!!! The server just forwards the exact PDU it receives to the destination clients.

3) **Required Flag values and packet formats.** The flag as specified below dictates the packet format. Your packets MUST match these formats.

- All PDUs start with a 2-byte length field in network order and then a flag value as discussed below.[1] These three bytes are referred to as the chat header.

- Flag = 1
    - Sent from cclient to server - Client initial packet to the server
    - Format: 3-byte 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: 3-byte 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
    - When the cclient receives this it prints out an error message and terminates

- 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.
    - The sending cclient does **not** block waiting for a response from the server after sending the %M message

- Flag = 6
    - Sent from the cclient to multiple other clients – multicast packet.
    - See format above under the %C (multicast) command.

- Flag = 7
    - From server to cclient - Error packet, if a destination handle in a message/mulitcast packet (flag = 5 or flag = 6) does not exist.

---

[1] This 2-byte PDU length is added to your PDU by the sendPDU() function you created for lab.

- 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 or flag = 6 message.

- Flag = 8 (Unused in this version of the program)

- Flag = 9 (Unused in this version of the program)

- 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, sent immediately following 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 name 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 to 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) If you are using your recvPDU()/sendPDU() functions (which you should be using) that you developed in lab, these functions put the two byte length on your PDU. So you have the length part of the PDU done. Do NOT add a second length field to your PDU.

6) Your PDU length field (which you wrote in the lab assignment) must be the length of the entire PDU (so includes counting the two byte length field in the total PDU length). This PDU length must be in network order. Our testing expects this.

7) The server should never block waiting on a response from the client. In this respect the server is stateless, it receives TCP connections and accept()s them, receives packets and processes them but after either of these actions it just goes back to poll() waiting for the next thing to do. (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.)

8) Use of usleep(), sleep() or any other delay functionality (other than poll()) will result in a 0. Your program should not require this. Using poll() with a timeout value other than -1 will result in a zero.

9) 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 poll() call on the client and the server to block your program until a packet/STDIN is ready for receiving/reading.

10) 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.

11) 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.

12) All calls to the poll() system call must use a time value of -1 (blocking call). We will look at all of your poll/pollLib calls to confirm this. A time value to poll() other than -1is not allowed.

13) You cannot make your sockets non-blocking or use your sockets in any non-blocking fashion. You must use poll()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.

14) 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.)

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

16) 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.

17) 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.

18) 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.

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

20) A client needs to be able to send (%M, %C) to itself.

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

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

23) Handles are limited to 100 characters. If a handle 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).

24) The maximum message length (command, handles and text) is 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 -199 of text, 1-byte null for 200 bytes total)

   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.

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

26) 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.

27) The command names **cclient** and **server** must be used. Also the run-time parameters should be in the order given. See the provided Makefile for help on this. Since we will 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 make the graders upset and may lose 20%.

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

29) 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.

30) If the server receives 0 bytes on a socket, this means the other end has closed the socket and the cclient has 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 table) but NOT try to send a response to the client since at this point the client is already gone.

31) If the cclient receives 0 bytes on a recv() on the socket to the server, the client should print out the message "Server Terminated" and then end cleanly.

32) 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. Do not handle SIGINT on either the client or server.

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

   (continued on next page)

---

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

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

[3] Case when sending a message and the destination client does not exist on the server

[4] User enters an invalid command

34) A run of these programs would look similar to the boxes below:

Note – **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 in different looking output.

Note – The client ends using ^c. The server determines a client has terminated based on the return value from the recv() call.

In a window – using unix1:

bash$: server 54321
Server is using port 54321

Client1

In another window – using unix1

bash$: cclient client1 localhost 54321
$: %M clientX Hello how are you?
$:
Client with handle clientX does not exist.
$: %M client2 Hello how are you?
$:
client2: Fine, thanks.
$: %C 3 client2 client4 clientX Going Home
Client with handle clientX does not exist
$:
client2: Hello to all
$: ^c
bash$:

Client2

In another window – using unix3:

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

Client 4

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 client2 Hello back
client1: Going Home
$:
client2: Hello to all
$: ^c
bash$: