Program 1

Assigned: Tuesday Oct. 10
Due Date: Thursday Nov. 2
Notes: Undergraduates may work in pairs, but graduate students must do this project solo.

DuckChat

Chat, or Instant Messaging, is one of the most popular applications of the Internet. In this assignment, you will develop a chat application that speaks the DuckChat protocol. This will familiarize you with the task of implementing a network protocol and introduce you to the Berkeley socket API. The specification for DuckChat is provided for you. Since everyone will be implementing the same protocol, your chat programs will be able to interoperate with one another.

DuckChat is a simple client-server protocol. You should implement both client and server sides of the DuckChat protocol for this assignment. Users run client software to connect to a server and communicate with other connected users. Users join a channel (sometimes called a "chat room") and communicate with other people on the channel. Users may join as many channels as they please and can even create their own channels.

Client User Interface

The client program must take exactly three command-line arguments. The first is the host name where the server is running. The second argument is the port number on which the server is listening. The third argument is the user's username.

When the client starts, it automatically connects to the chat server, joins a channel called “Common”, and provides the user a prompt (i.e. the client must send the join message to join “Common”). When the user types/enters text at the prompt and hits 'Enter', the text is sent to the server (using the "say request" message), and the server relays the text to all users on the channel (including the speaker).

The exception is when the text begins with a forward slash ('/') character. In that case, the text is interpreted as a special command. These special commands are supported by the DuckChat client:

- `/exit`: Logout the user and exit the client software.

- **`/join` *`channel`***: Join (subscribe in) the named channel, creating the channel if it does not exist.
- **`/leave` *`channel`***: Leave the named channel.
- **`/list`**: List the names of all channels.
- **`/who` *`channel`***: List the users who are on the named channel.
- **`/switch` *`channel`***: Switch to an existing named channel that user has already joined.

A user can listen to multiple channels at the same time but it can send text (be active) on only one channel at any point of time (known as "active channel" for a user). The most recently joined channel by a user is always the active channel for that user. Note that Common is the active channel for each user upon invoking the client program. All other channels other than the active channel act as listening channels only. The user may use the /switch command to switch to another channel. The client should keep track of the active channel for the user and therefore the /switch command does not require sending any message to the server. However, the join command requires the client to send a message to the server.

If the user leaves the active channel, all their typed text is discarded until s/he joins a new channel or switches to an already subscribed channel. The client should detect when the user tries to switch to a channel to which it has not already subscribed and give an error message to the user. If the switch command fails, the active channel should remain unchanged. Otherwise, the user's active channel becomes the named channel in the switch command.

When a client receives text on a subscribed channel, it must be displayed in the following format:

```
[channel][username]: text
```

Before displaying text, the client application should print many backspace characters ('\b' in C) to erase the prompt and anything the local user has typed. (Carefully test this on ix if you are using Java.) After the text is printed, the client should redisplay the prompt and any user input. This means the client will need to keep track of the user's input as they are typing it.

Below is a sample run. The bold text is typed by the user.

```
ix:~$ ./client ix 5000 Soja
> /list
Existing channels:
  Common
> /who Common
Users on channel Common:
```

```
  Soja
> Hello, Soja!
[Common][Soja]: Hello, Soja!
> /join New_Channel
> Hello, Soja!
[New_Channel][Soja]: Hello, Soja!
> /switch Common
> Hello, Soja!
[Common][Soja]: Hello, Soja!
> /exit
ix:~$
```

Server

To make the project seem more real, but not without the added complexity of TCP, DuckChat uses UDP to communicate. While UDP is not reliable, we can run the client and server on the same machine and not worry about dropping packets or out of order packets.

The server takes two arguments: the host address to bind to and the port number. The host address can be either 'localhost', the IP host name of the machine that the server is running on, or the IP address of an interface connected to the machine. Once the server is running, clients may use the host name and port to connect to the server. Note that if you use 'localhost', you will not be able to connect to the server from another machine, but you also do not have to worry about dropped packets.

The server does not need to directly interact with the user in any way. However, it is strongly recommended that the server outputs debugging messages to the console. For example, each time the server receives a message from a client, it can output a line describing the contents of the message and who it is from using the following format: [channel][user][message] where message denotes a command and its parameters (if any).

The server's primary job is to deliver messages from a user X to all users on X's active channel. To accomplish this, the server must keep track of individual users and the channels they have subscribed to. On the flip side, the server must also track each channel and the subscribed users on it.

Channel creation and deletion at server are handled implicitly. Whenever a channel has no users, it is deleted. Whenever a user joins a channel that did not exist, it is created.

If the server receives a message from a user who is not logged in, the server should silently ignore the message.

Protocol

The DuckChat protocol runs over UDP. Each UDP datagram from a client is a request, asking the server to perform some action. Each UDP datagram from the server is text meant to be displayed to the user.

Each message begins with a 32-bit type identifier. This is a code that designates what type of message the datagram contains. By examining this code, the application can determine how to interpret the rest of the message.

The following messages may be sent to the server. They correspond closely with the commands the user may issue on the client. The server must parse these messages, then perform the right action. For the login, logout, join, and leave requests, the server must update its records of which users are present and what channels the user belongs to. For the say request, the server must relay the text to all users on that channel. For the list and who requests, the server must respond to the client with the corresponding text message. The type code is listed in parenthesis. Note that this type code is used as the 32-bit identifier in the header of each packet so that you can classify each packet as it arrives.

- **login request(0)**: The message contains an additional 32 byte username field. Here's the layout of the packet:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (0)             |
+---------------------------------------------------------------+
|                     32-byte user name                         |
                             ...
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **logout request(1)**: No additional fields are included. Here's the layout of the packet:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (1)             |
+---------------------------------------------------------------+
```

- **join request(2)**: The message contains an additional 32 byte channel name field. Here's the layout of the packet:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (2)             |
```

```
+---------------------------------------------------------------+
|                      32-byte channel name                     |
                              ...
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **leave request(3)**: The message contains an additional 32 byte channel name field. Here's the layout of the packet:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (3)             |
+---------------------------------------------------------------+
|                      32-byte channel name                     |
                              ...
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **say request(4)**: The message contains an additional 32 byte channel name field followed by a 64 byte text field. Here's the layout of the packet:

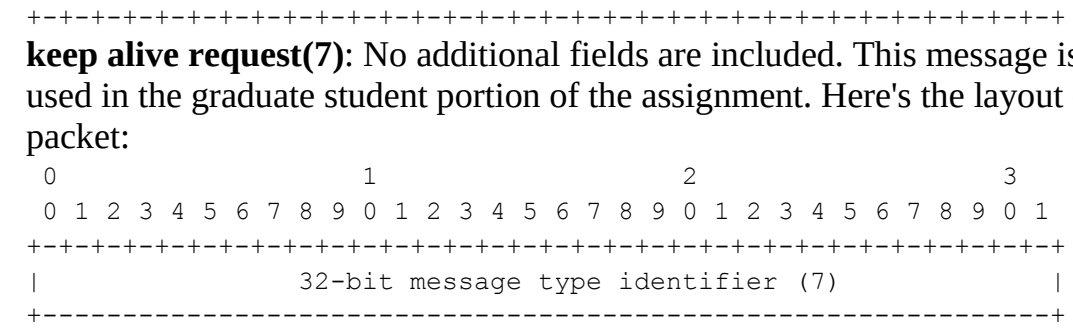
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (4)             |
+---------------------------------------------------------------+
|                      32-byte channel name                     |
                              ...
|                                                               |
+---------------------------------------------------------------+
|                      64-byte text field                       |
                              ...
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **list request(5)**: No additional fields are included. Here's the layout of the packet:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (5)             |
+---------------------------------------------------------------+
```

- **who request(6)**: The message contains an additional 32 byte channel name field. Here's the layout of the packet:

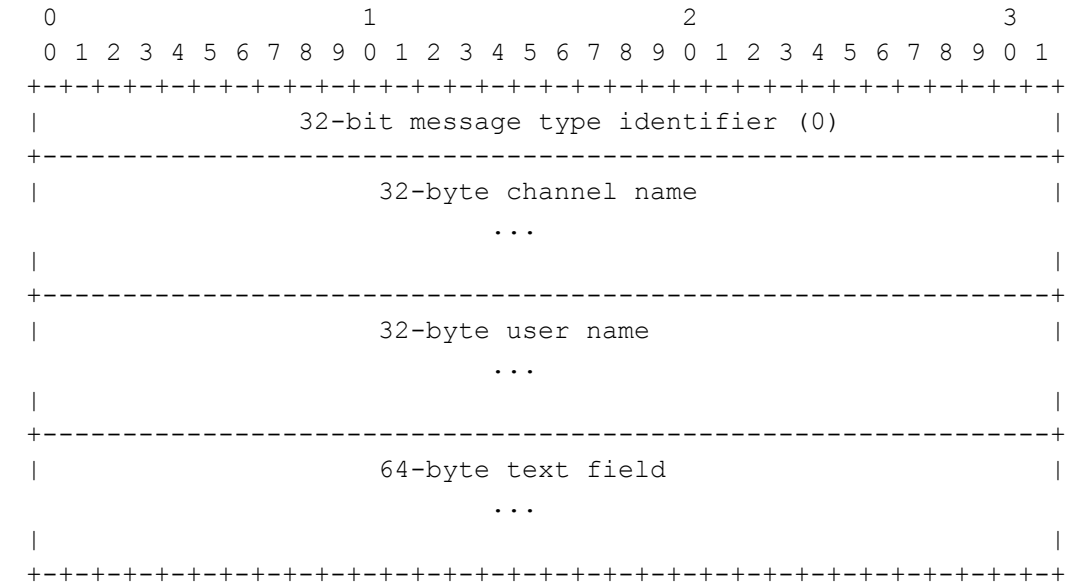
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (6)             |
+---------------------------------------------------------------+
|                      32-byte channel name                     |
                              ...
|                                                               |
```



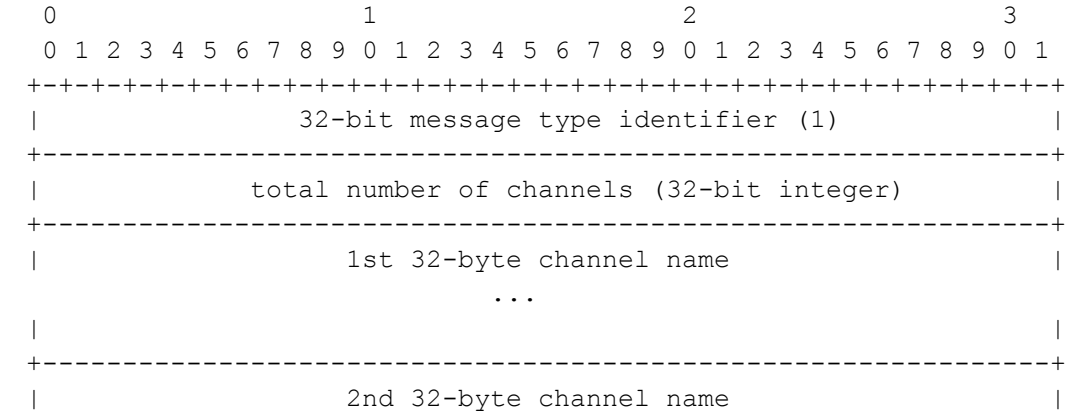
- **keep alive request(7)**: No additional fields are included. This message is only used in the graduate student portion of the assignment. Here's the layout of the packet:

If the server finds a problem with the users request, it denies the request by sending an error message back to the user. The following messages may be sent to the client:

- **say (0)**: The message contains an additional 32 byte channel name field, a 32 byte username field, and a 64 byte text field. Here's the layout of the packet:



- **list (1)**: The message contains a 4 byte field that specifies the total number of channels. Following that are 32 byte channel names, one for each channel. Here's the layout of the packet:



```
                              ...
|                                                               |
+---------------------------------------------------------------+
|                    nth 32-byte channel name                   |
                              ...
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **who (2)**: The message contains a 4 byte field that specifies the total number of users on the channel. Following that is a 32 byte channel name and 32 byte usernames, one for each user. Here's the layout of the packet:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 32-bit message type identifier (2)            |
+---------------------------------------------------------------+
|               total number of users (32-bit integer)          |
+---------------------------------------------------------------+
|                     32-byte channel name                      |
                              ...
|                                                               |
+---------------------------------------------------------------+
|                    1st 32-byte user name                      |
                              ...
|                                                               |
+---------------------------------------------------------------+
|                    2nd 32-byte user name                      |
                              ...
|                                                               |
+---------------------------------------------------------------+
|                    nth 32-byte user name                      |
                              ...
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **error (3)**: The message contains a 64 byte text field with an error message for the user. Here's the layout of the packet:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 32-bit message type identifier (3)            |
+---------------------------------------------------------------+
|                      64-byte error message                    |
                              ...
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Grading

- 50 points total, Client Program:
  - 10 points. The program compiles and runs. Login works, logout works, and the program accepts user commands.
  - 10 points. Say works.
  - 10 points. Join, Leave and Switch work.
  - 10 points. List and Who work.
  - 10 points. All client functionality and the client is robust.
- 50 points total, Server Program:
  - 10 points. The server compiles, runs, and Login and Logout work.
  - 10 points. Say works.
  - 10 points. Join and Leave work.
  - 10 points. List and Who work.
  - 10 points. All server functionality and the server is robust.

"Robust" means that the program does not crash, even if we send it bogus packets. Be particularly careful to test your program with maximum length usernames, channel names, and text messages.

How We'll Test

- Your client will be tested against our server program.
- Your server will be tested against an armada of our clients.
- Your client will be tested with your server.
- To test robustness, your programs will be tested against badly behaved programs that send bogus datagrams.
- Automated scripts will be used to try many different scenarios.

Graduate Students

In addition to everything described above, graduate students must also implement "soft state" tracking of users. This feature allows the server to figure out that a user has logged off, even if a client abruptly dies and does not send a Logout message.

Each client must send a Keep Alive message once per minute, if and only if it hasn't sent any other data in the past minute. The server must track when it last received data from each client. Once every two minutes, the server must search through all users and forcibly logout those who have not sent anything in the last two minutes. Clients should still send the Logout message before they exit, though you may want to temporarily disable this feature to test the server's timeout.

The grading for graduate students is similar to the grading for undergraduates:

- 50 points total, Client Program:
  - 8 points. The program compiles and runs. Login works, logout works, and the program accepts user commands.
  - 8 points. Say works.
  - 10 points. Join, Leave and Switch work.
  - 8 points. List and Who work.
  - 4 points. Client sends Keep Alive message when needed.
  - 4 points. Client does not send Keep Alive when it is not needed.
  - 8 points. All client functionality and the client is robust.
- 50 points total, Server Program:
  - 8 points. The server compiles, runs, and Login and Logout works.
  - 8 points. Say works.
  - 10 points. Join and Leave work.
  - 8 points. List and Who work.
  - 8 points. Server times out users correctly.
  - 8 points. All server functionality and the server is robust.

Undergraduates may complete the graduate student portion of the assignment for up to 8 points of extra credit.

Code Snippets

We're providing a few links to code to help you along if and only if you are using C. Those using java should still look at these for a general idea of how to structure your program:

- duckchat.h: the header file that defines the main structures and codes used in the program. Use this to help you get started.
- Makefile: a Makefile to help you compile your socket code on the department server (ix-trusty). (You might have to change this file depending on the files you use)
- raw.h: header file for raw.c (see below).
- raw.c: raw.c, which has the functions needed to turn off and on character (instead of line) input from the terminal.

The following links are for working binaries that you can test your program against.

For Undergrads: (Right click & save as...)

- client: the client program binary.
- server: the server program binary.

For Grad. students:(Right click & save as...)

- client: the client program binary.
- server: the server program binary.

Notes

Sometimes the “network byte order” which is the order used in the TCP and IP headers may differ from the byte order used on the host computer. So, if you are using C, for the sake of proper network programming, it is important to use `htonl()` and `ntohl()` in order to translate between host byte order and network byte order when writing to and reading from network packets. By default, Java streams use Network Byte Order so there is nothing to worry about if you are using Java, but it would be wise to test it anyway.

The maximum character length of channels, users, and "say" text messages can be found in duckchat.h. For your convience they are 32 for both channels, and users, and 64 characters for chat messages. The maximum number of channels and users is expected to be at least 10.

Tips

Don't leave this assignment until the last minute! You won't be able to finish it, and there'll be less time to ask questions if you have any.

Develop your program gradually. Get a minimum set of functionality coded, tested, and debugged before moving on to additional functionality. We recommend writing the client first. Here is one well-tested order to write the code in:

- Client connects, logs in, and joins “Common”.
- Client reads lines from the user and parses commands.
- Client correctly sends Say message.
- Client uses select() to wait for input from the user and the server.
- Client correctly sends Join, Leave, Login, and Logout and handles Switch.
- Client correctly sends List and Who.

- Server can accept connections.
- Server handles Login and Logout from users, and keeps records of which users are logged in.
- Server handles Join and Leave from users, keeps records of which channels a user belongs to, and keeps records of which users are in a channel.
- Server handles the Say message.
- Server correctly handles List and Who.
- Create copies of your client and server source. Modify them to send invalid packets to your good client and server, to see if you can make your client or server crash. Fix any bugs you find.

The following points may be helpful in developing your program.

- The sockets to be used here are NOT domain sockets.
- The Map class in STL of C++ may be helpful in implementation.

Submission

Step 1:

Fill the correct submission form and compress all the source files, makefile and the submission form, into a file (.zip, .tar , ...)

Links to the submission forms are given below:

- Undergraduate Students
- Graduate Students

Again, put all the source files, makefile (if you are using C) and the submission form into a folder by your last name (or the last name of one of the group members if you are working in a group of two). Make a compressed file from the folder. Please include any special instructions for compiling or running your program on ix.

Step 2:
Please use the TurnIN Homework submission page (Click Here) to upload the compressed file of your project .
First you need to login if you are not logged in, using your CIS login name and password - @cs.uoregon.edu -.
Then, Choose the "CIS 432" (532 students also should select this option) and press "GO",
Choose "prog1" (532 students choose "prog1_grads") and press "GO"

choose your compressed file and press the "GO" for the last time.

That's it!

Make sure that you submit your program before the deadline. (11:59pm on Thursday Nov. 2)