Programming assignment 4.
Best mismatch. INET sockets.

There is no place like 127.0.0.1!

In this assignment, you will implement a network version of the *categorizer* program that you wrote for Assignment 2, but with a slight twist. You will develop an online dating service called "Best Mismatch". Assuming that opposites attract, our program will accept user preferences as before, but will find a group of users that have exactly opposite interests. It will then recommend these users as potential best mismatches for dating or friendship. It will print the list of best mismatches and will allow the current user to send messages to any user from this group.

You will write a socket server which implements this functionality. Clients can connect to your server from any machine, and get the recommendations. For a client program, we use *netcat*. You **do not** need to implement a socket client for this assignment.

Below you will find a step-by-step description of the product and the code examples with explanations. The lecture about sockets is scheduled for November 29, and these (probably too detailed) explanations should help you to get started with the assignment.

1. Reusing code from Assignment 2

The first step is to review a *categorizer* prototype you have written for Assignment 2. If your code does not work properly, as indicated by your mark for Assignment 2, it is a good time to come to office hours or to the lab and fix everything with your instructor or TAs. Do not start writing a socket version before you make sure that your local code works as expected.

You have already implemented the functionality of producing a list of users based on a given path in the question tree. It is not difficult to extend this functionality to the case when given a list of answers you move to the leaf in the direction *exactly opposite* to the answers and reach the bucket with the users who have exactly opposite interests. Add this functionality and test that it works.

Another useful feature is to be able to get a list of answers for a given user name. For this, you may want to modify your depth-first search to maintain a current path that lead to a given leaf. In case that you found a user with a given name in one of the leaves, you will now have the list of their original answers which you can pass to the function above and get the best mismatches for the current user.

2. Socket server protocol

Users and clients

In you program, you need to collect all users with their answers into a question tree.

In addition, you need to maintain a list of currently connected clients. When user logs in, you create a new node and add it to the list of clients. When user disconnects, you remove its node from the client list.

It might be useful to allocate an array of answers and store it alongside each connected client, because clients may request the list of recommendations at any point, and in this way you would avoid traversing the question tree with each new request.

IMMP - protocol for mismatch server

Client login

You should be able to connect to your server by typing the following shell command:
`netcat -C wolf.teach.cs.toronto.edu 12345`,
where 12345 will be your port number.

Once you, the user, are connected, you will be asked for the user name. You type the name and hit enter. You may use the same rules for the user name as in Assignment 2, but we will not test for a valid user name while marking this assignment. The only requirement is that the user name does not exceed 128 characters, and if it does, it is truncated by the server. The user name is case-sensitive, as before.

If you provide a new user name (never before seen by the server) at the time you connect, the server will create a new user, but if a user by this name already exists in the question tree, it will assume that you are this user. No users are ever deleted from the question tree.

To terminate the current session, client either types quit, or terminates the *netcat* by sending an interrupt signal Ctrl+C.

When the server is ready to accept commands, it reacts to the commands listed in the table below. Any other command issued by a client should result in a “command not supported” error message returned to the client.

List of poll commands

Command	Description
do_test	Signifies that the current user is ready to answer questions about their preferences. Server reacts by asking each question from the provided questions file.
yes/no	Answers to the questions of the current test. The rules are as in Assignment 2: any of YXX and NXX commands are accepted, and the yes/no answers are case-insensitive. Server collects the answers and assigns the user to the corresponding leaf list. In addition, it might choose to record user answers in a separate array, in order to use its reverse for the next command.
get_all	At any point, the user may request the list of best mismatches to be returned. This list will only be produced if the user has already taken a test of preferences. If the user did not take the test yet, the appropriate error message is returned.
post <target_name> <message>	Delivers *<message>* from the current client to the user whose name is specified as *<target_name>*. The message can contain several words, but has a restriction on the total length: at most 1024 characters.
quit	Disconnects the client and removes him from the list of active clients.

3. Implementing a single-client server

3.1. Establishing communication

Set up your socket interface, bind it to the known port, and implement message passing between the server and a single client connected with *netcat*. You may start from implementing a simple echo server that echoes each message back to the client. There are plenty of echo server implementations, see for example here.

The entire program should be compiled using *make*. You will create a *Makefile* that produces an executable called *mismatch_server*.
It must use the following GCC flags: -std=c99, -Wall, and -Werror.

You should be able to start your server by typing the following command:
./mismatch_server <questions_file_name>

In addition to building your code, your *Makefile* must permit choosing a port at compile-time. In total, there should be three ways of defining the port.
First, add a #define preprocessing directive to your program to define the port number on which the server will expect connections (this is the port <x> based on your student number, as described in lab 11):

```
#ifndef PORT
  #define PORT <x>
#endif
```

Secondly, in your *Makefile*, include the following code, where `<y>` should be set to your student number port plus 1:

```
PORT=<y>
CFLAGS+= -DPORT=\$(PORT)
```

Now, if you type `make PORT=53456,` the program will be compiled with PORT defined as 53456. If you type just `make`, PORT will be set to y as defined in the *Makefile*. Finally, if you use *gcc* directly and do not supply a port number, it will still have the *x* value from your source code file. This method of setting a port value will make it possible for us to test multiple submissions by compiling with our desired port numbers. (It is also useful for you to know how to use -D to define macros at command line.)

You should also make sure to add the following lines to your server code so that the port will be released as soon as your server process terminates.

```
int on = 1;
int status = setsockopt([sock_fd], SOL_SOCKET, SO_REUSEADDR,
                            (const char *) &on, sizeof(on));
if(status == -1) {
    perror("setsockopt -- REUSEADDR");
}
```

Once you familiarize yourself with the steps needed to configure a socket server, you may start implementing the required functionality, gradually adding each new feature after you have tested the previous one.

3.2. Client connects

When a new client connects, add them to the list of active clients, store their personal file descriptor returned by *accept*, and then ask for and store their name. You can use a predefined string buffer of at most 128 characters (including the terminator '\0') for the user name, and you can truncate it if the user enters longer name, but you need to notify the user about it.

You will need to maintain an independent linked list - to store all active clients, currently connected to the server.

The suggested structure of each Client node is presented below.

```
typedef struct client {
      int fd;    //file descriptor to write into and to read from
      int *answers;
      //before user entered a name, he cannot issue commands
      int state;
      char name [MAX_NAME];
      char buf [BUFFER_SIZE];  // each client has its own buffer
      int inbuf; // and a pointer to the current end-of-buf position
      struct client *next;
} Client;
```

Note that the server keeps all its data about clients and users in memory. Once the server is killed, all user information is gone.

Sample code for new connection may look like this:

```
int fd;
struct sockaddr_in r;
socklen_t socklen = sizeof(r);

if ((fd = accept(listenfd, (struct sockaddr *)&r, &socklen)) < 0) {
      perror("accept");
      return;
}
add_client (fd, r.sin_addr);  //call insert into linked list
```

3.3. Reading client commands into a dedicated buffer

Once the connection is established and a new file descriptor for each client is added, we can use regular *read* and *write*, as with all file descriptors, to exchange messages between the server and the client.

Useful note 1. Network new line convention

In the case of transmitting text, the ASCII standard gives us standard byte values for just about everything except newlines. There is an accepted convention that the network newline is CRLF.

That is, a newline is represented by the two bytes (in order) which we could call CR and LF, or control-M and control-J, or 13 and 10, or \015 and \012, or \r\n.

Thus, if the user sends two commands separated by a new line, we need to be able to extract each line and process it separately. The following function suggests the simplest way to find the position of a new line in a network message:

```
int find_network_newline (char *buf, int inbuf) {
      int i;
      for (i = 0; i < inbuf - 1; i++)
            if ((buf[i] == '\r') && (buf[i + 1] == '\n'))
                  return i;
      return -1;
}
```

Useful note 2. Partial reads problem

A single message may arrive in packets, so we should be able to read and parse everything until the new line, and then keep the remaining data in buffer until the end of the message arrives later.

This can be implemented in the following way:

```
char *after = buf + inbuf;
int room = BUFFER_SIZE - inbuf;
int nbytes;
//read next message into remaining room in buffer
if ((nbytes = read(fd, after, room)) > 0) {
      inbuf += nbytes;
      int where = find_network_newline (buf, inbuf); //find new line
      if (where >= 0) {
            buf[where] = '\0'; buf[where+1] = '\0';
            do_command (buf); //process buffer up to a new line
            where+=2;  // skip over \r\n
            inbuf -= where;
            memmove (buf, buf + where, inbuf);
      }
}
```

3.4. Parsing client commands

Now, when you have the properly extracted message from the client, you need to parse it to handle client request. To parse the message, you may use the `strtok` function. An example is presented below:

```
/*
** This program extracts tokens from a string using all characters
specified in a delimiter.
*/

#include <stdio.h>
#include <string.h>

int main () {
            char str[] ="This, a sample string! \n'";
            char * pch;
            char delimiter[] = " \n";
            printf ("Splitting string \"%s\" into tokens:\n", str);

            pch = strtok (str,delimiter);
            while (pch != NULL) {
                        printf ("%s\n", pch);
                        pch = strtok (NULL,delimiter);
            }
            return 0;
}
```

When you have extracted the command and its arguments, you handle each command, check for a correct message format, and write the corresponding message to the client, using the same file descriptor. The *netcat* client takes care of all the problems described above, so you may use regular `write` command.

When implementing the *post <target> <message>* functionality, you need to locate the target user in the list of active clients first. If not found, you need to return the message "Your post cannot be delivered. User *<user_name>* is not online". Otherwise, you post the original message by writing it to the file descriptor of a target client.

3.5. Disconnecting

When the user types *quit*, the server should remove him from the list of active clients and probably send back some sort of goodbye message. The server will also remove the clients who terminated the session without typing *quit*, by periodically checking closed file descriptors.

3.6. Constraints on the client-server interaction

If the client issues an unsupported request or provides invalid parameters, he should be notified about the error. The client cannot issue the *get_all* command before he issued the *do_test* command and finished the test. You may need to store a status of each client in the current session in the field status (see sample definition of Client). All client commands should be handled in your code and appropriate data or error notifications should be sent back to the client.

4. Support for multiple clients

When several clients are connected and issue commands, you'll need to ensure that the server is not blocked waiting for the response from one of the clients.
The server must never block waiting for input from a particular client or the listening socket. After all, it can't know which client will talk next or whether a new client will connect. This means that you must use *select* rather than blocking on one file descriptor.

An example of using *select* is shown below:

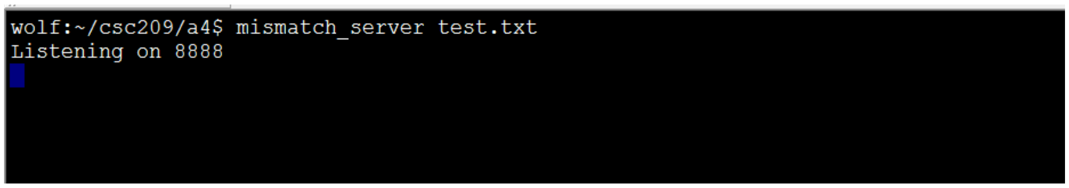
```
if (select(maxfd + 1, &fdlist, NULL, NULL, NULL) < 0) {
     perror("select");
} else {
     for (p = top; p; p = p->next)
          if (FD_ISSET(p->fd, &fdlist))
                    break;
     if (p)  //client message received
          handle(p);
     if (FD_ISSET(listenfd, &fdlist)) //new connection
          newconnection(listenfd);
}
```

5. Testing

Since you're not writing a client program, the *netcat* tool mentioned above can be used to connect clients to your server and to test your program.
To use it, type netcat -C hostname yyyyy, where hostname is the full name of the machine on which your server is running, and yyyyy is the port on which your server is listening. If you aren't sure which machine your server is running on you can run hostname -f to find out. If you are sure that the server and client are both on the same machine, you can use localhost in place of the fully specified host name.

Test your final product with the following basic use cases. Each use case is accompanied by a screenshot of a running demo. Try to keep your output formats close to the ones presented in these screenshots, to avoid problems with automated testing.

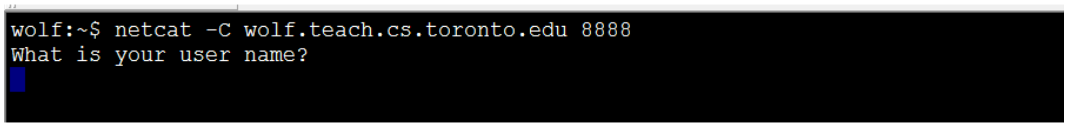
Case 1: We can start the server by typing:



```
wolf:~/csc209/a4$ mismatch_server test.txt
Listening on 8888
```

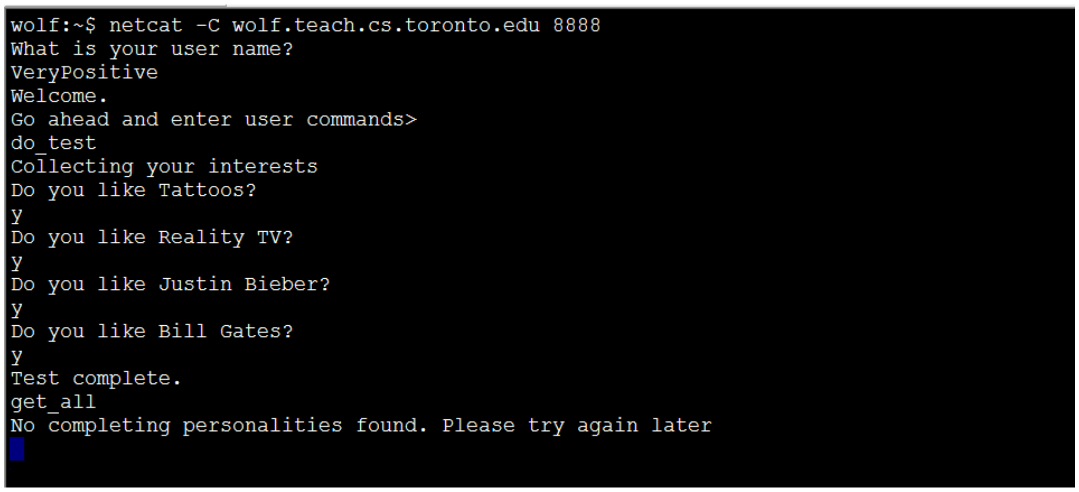
Server starts and prints the port it is listening at.

Case 2: The first client connects (in a separate window) with netcat:



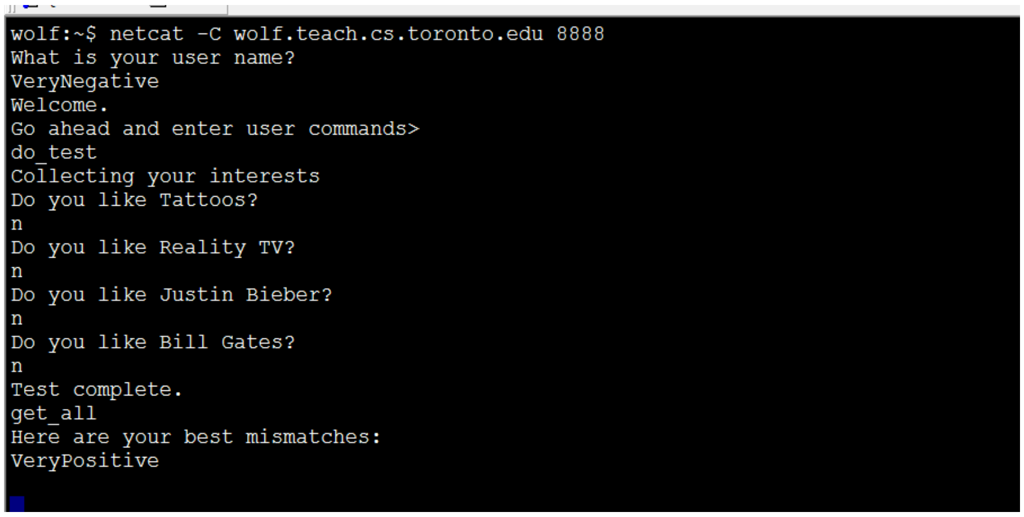
```
wolf:~$ netcat -C wolf.teach.cs.toronto.edu 8888
What is your user name?
```

Case 3. Performing the test and asking for recommendations:



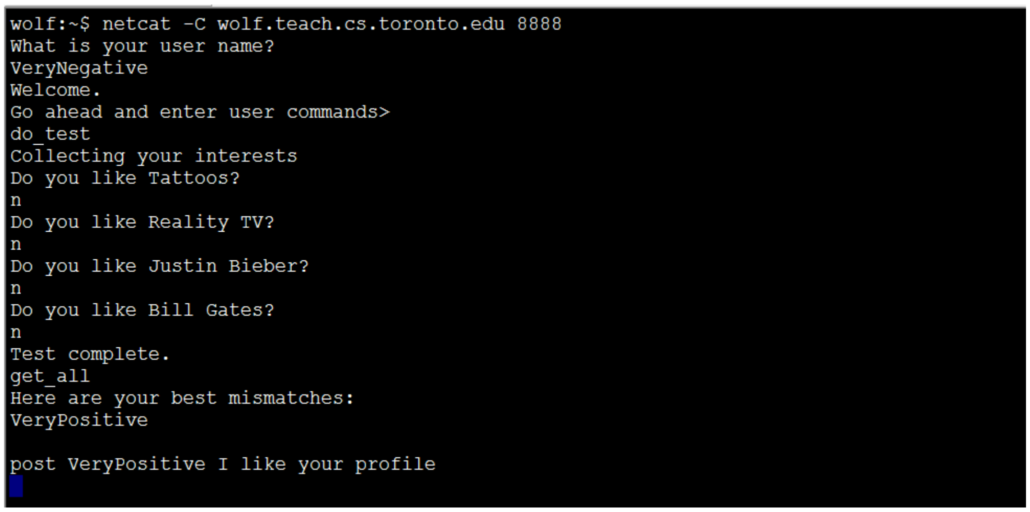
```
wolf:~$ netcat -C wolf.teach.cs.toronto.edu 8888
What is your user name?
VeryPositive
Welcome.
Go ahead and enter user commands>
do_test
Collecting your interests
Do you like Tattoos?
y
Do you like Reality TV?
y
Do you like Justin Bieber?
y
Do you like Bill Gates?
y
Test complete.
get_all
No completing personalities found. Please try again later
```

Case 4. The second client can connect (in a separate window) while the first one is still connected.



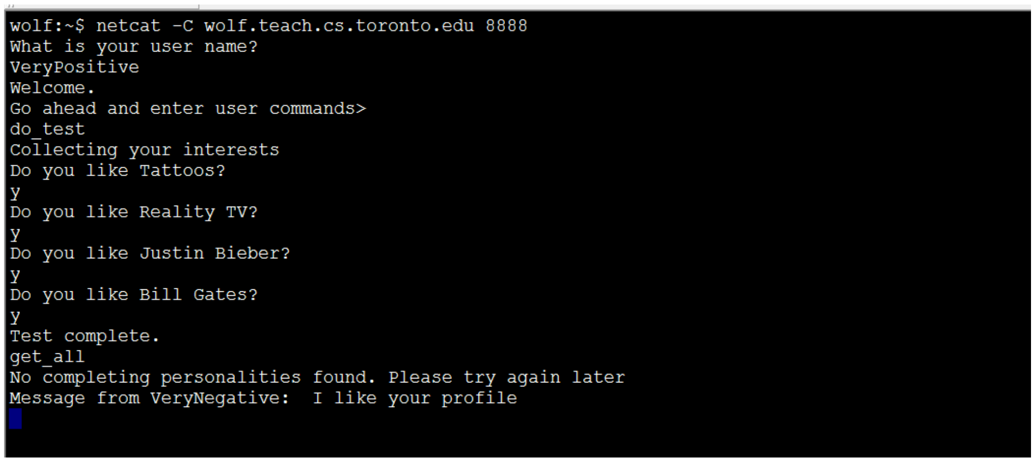
```
wolf:~$ netcat -C wolf.teach.cs.toronto.edu 8888
What is your user name?
VeryNegative
Welcome.
Go ahead and enter user commands>
do_test
Collecting your interests
Do you like Tattoos?
n
Do you like Reality TV?
n
Do you like Justin Bieber?
n
Do you like Bill Gates?
n
Test complete.
get_all
Here are your best mismatches:
VeryPositive
```

Case 5. Posting message from VeryNegative to VeryPositive:



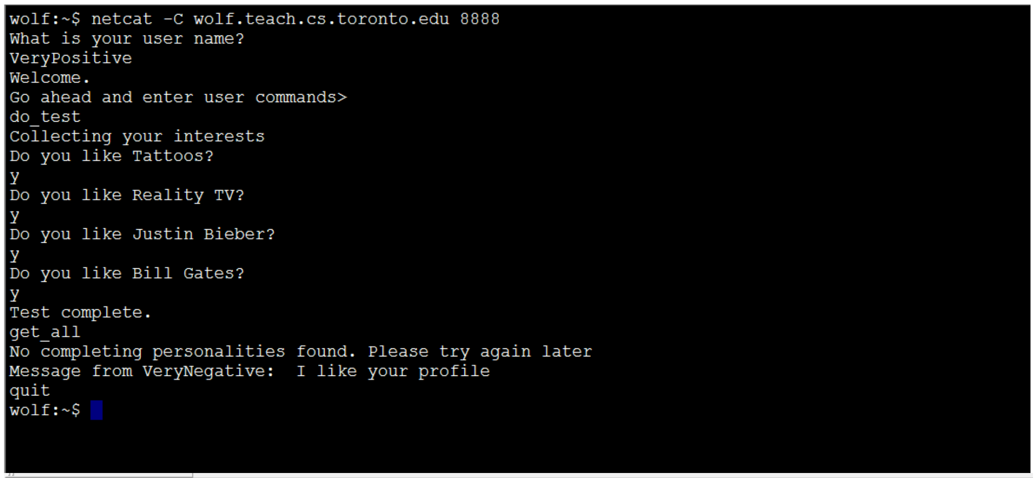
```
wolf:~$ netcat -C wolf.teach.cs.toronto.edu 8888
What is your user name?
VeryNegative
Welcome.
Go ahead and enter user commands>
do_test
Collecting your interests
Do you like Tattoos?
n
Do you like Reality TV?
n
Do you like Justin Bieber?
n
Do you like Bill Gates?
n
Test complete.
get_all
Here are your best mismatches:
VeryPositive

post VeryPositive I like your profile
```

Message delivered:

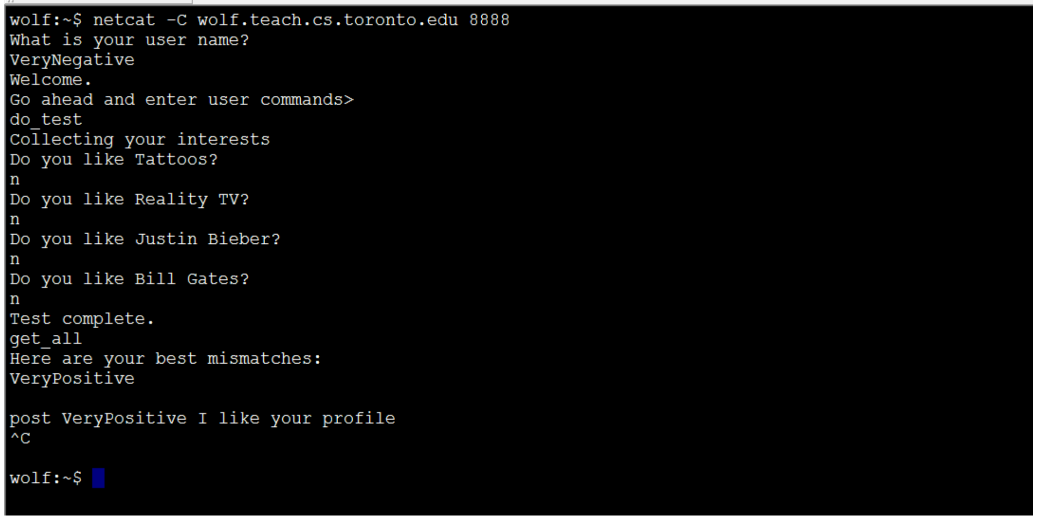


```
wolf:~$ netcat -C wolf.teach.cs.toronto.edu 8888
What is your user name?
VeryPositive
Welcome.
Go ahead and enter user commands>
do_test
Collecting your interests
Do you like Tattoos?
y
Do you like Reality TV?
y
Do you like Justin Bieber?
y
Do you like Bill Gates?
y
Test complete.
get_all
No completing personalities found. Please try again later
Message from VeryNegative:  I like your profile
```

Case 6. Clients disconnect with either quit command, or with an interrupt signal.

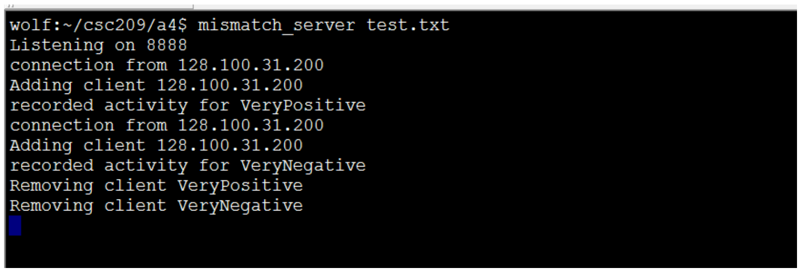


```
wolf:~$ netcat -C wolf.teach.cs.toronto.edu 8888
What is your user name?
VeryPositive
Welcome.
Go ahead and enter user commands>
do_test
Collecting your interests
Do you like Tattoos?
y
Do you like Reality TV?
y
Do you like Justin Bieber?
y
Do you like Bill Gates?
y
Test complete.
get_all
No completing personalities found. Please try again later
Message from VeryNegative:  I like your profile
quit
wolf:~$
```



```
wolf:~$ netcat -C wolf.teach.cs.toronto.edu 8888
What is your user name?
VeryNegative
Welcome.
Go ahead and enter user commands>
do_test
Collecting your interests
Do you like Tattoos?
n
Do you like Reality TV?
n
Do you like Justin Bieber?
n
Do you like Bill Gates?
n
Test complete.
get_all
Here are your best mismatches:
VeryPositive

post VeryPositive I like your profile
^C

wolf:~$
```

Server handles client disconnect:



```
wolf:~/csc209/a4$ mismatch_server test.txt
Listening on 8888
connection from 128.100.31.200
Adding client 128.100.31.200
recorded activity for VeryPositive
connection from 128.100.31.200
Adding client 128.100.31.200
recorded activity for VeryNegative
Removing client VeryPositive
Removing client VeryNegative
```

6. Sample Code

Here is a sample server written by Alan Rosenthal that you might find helpful. Feel free to yank code from there, with two important warnings:

- Don't copy-and-paste stuff into your program and then fuss with it to make it work. You should know exactly what the code does and why. We won't take kindly to extra stuff in your code that is unnecessary or does not work, and is clearly left over from the sample server.
- Clearly indicate the code that you copied from the sample server.

7. Coding style

Coding style and code readability are very important. Use good variable names, appropriate functions, descriptive comments, and blank lines. Remember that someone needs to read your code. You may write extra helper functions. You MUST perform error-checking for all system calls (and functions which use system calls). We recommend defining a set of helper functions to wrap these calls to reduce duplication.

8. What to submit

Commit all the files required to build your executable. That includes **.h* files, **.c* files, and your *Makefile*. Make sure your code compiles on CDF with the required flags **before** your final submission. Also, make sure to check your repo URL *on MarkUs* before your final submit. Remember to check that your program compiles without warnings when you type `make` and that it produces an executable called mismatch_server. Make sure that you've done `svn add` on every file. If you fail to commit some files and your code does not compile, your work will not be graded!

As a final step, checkout your repository into a new directory to verify that everything is there.

9. Marking scheme

- *Server setup [10%]:* The code compiles without warnings. Servers starts and releases the port after shutdown. The client can connect and issue commands according to the IMMP protocol.
- *One client at a time [40%]:* One client at a time can connect and issue all the commands. The server returns a correct list of mismatches. The server issues relevant error messages.
- *Multiple clients connected and served at the same time [30%]:* Support for multiple clients using select.
- *Coding style and error checking [10%]:* All system calls need to have error checking. The best way to do it without cluttering your code is to write your own helper functions for each system call which check for errors, and use these custom functions instead of the original ones.
- *Proper port setting [10%]:* We should be able to run your program with the port number set from the command-line during the compilation.

For a total of 12% of the course grade.

Happy final coding!