

CSC 209 Assignment 4, Fall 2011: Chat Bot

Due by the end of *Tuesday* December 6, 2011; no late assignments without written explanation.

In this assignment you will write a program which functions as a “bot”. I have written a simple chat server. Your program will connect to this chat server and in addition to being a normally-functioning client, it will respond to certain messages sent by other users.

Your program will sign in with the user name “Marvin”. When another user says “Hey Marvin,” followed by an arithmetic expression, your program will compute this expression and reply “Hey [username], [result]”.

One of the distributed files is `parse.c`, which can parse and compute the value of these expressions. See `testparse.c` for a demonstration of how to use it. Note how it reports syntax errors through a global variable “`errorstatus`”. When this occurs, Marvin should instead say “Hey [username], I don’t like that.”. Additionally, display the error condition on `stdout` in square brackets (this is mostly to help you in debugging, but do leave it in because it could also help in debugging a future other “bot”).

The user of your program can also type messages on `stdin`, and the program shows everything from the server on `stdout`. Thus if no one says “Hey Marvin,”, the only difference between this and a proper chat client is that it doesn’t let you enter your “handle”.

Note that the supplied chat server broadcasts all messages to all clients, including the client that sent them; thus your user will see the messages you generate, prefaced by “Marvin:”. (That is, you don’t have to do anything to make this happen.)

The invocation syntax for your program will be like that of `telnet`: `argv[1]` is a mandatory hostname; `argv[2]` is an optional port number. The default port number is 1234. Thus you do not need to use `getopt()` in your program, although `getopt()` is used in the supplied server where “`-p`” introduces the port number.

The `chatsvr` protocol

There is a specific, although simple, protocol for communicating with `chatsvr`. Upon connection, the `chatsvr` sends the string “`chatsvr 305975789`”, followed by a network newline. (You should get this string from the `#define` in `chatsvr.h`.) Check this string, because if you get something else, you are talking to a different program. (You can test this check by running something like “`marvin cdf.toronto.edu 22`”.)

The client program then first has to send a “handle” that the user will be known as (followed by a network newline). The maximum handle length is `MAXHANDLE` in `chatsvr.h`.

After that, at any time the client can send a line which the user has typed, and the server may send data which other people have typed. The data from the client is free-format, with a maximum string length (not including the newline) of `MAXMESSAGE` in `chatsvr.h`. The data from the server will consist of the other user’s handle, colon, space, then the message; this is meant to be a form suitable for direct display to the client (after converting the network newline).

Note that you can connect with “`telnet`” to experiment with the raw protocol; remember that the first line is your handle. There is also a supplied compiled “`chatclient`” program you can use, as well as a supplied compiled “`marvin`” for comparison. Make multiple connections to see it from all angles.

You might find a chat server running on `course.dgp.toronto.edu` which you can play with (in addition to ones you start yourself elsewhere).

Note that your assignment submission must work with anything which implements the specified protocol, not only the example server and client. For this reason and to deal with the entire range of allowable TCP behaviour, you need to read characters in a loop until you see the network newline; you can’t assume that you will get a line all at once.

The example server code does this correctly, and your code for reading a line from the server should be based on what the server does with each client (buffering a line until it is complete, and recognizing that a single `read()` call might return both the end of one line and the beginning of the next). The supplied “`trickysvr`” isn’t a full chat server, but you can use it with your client in order to test your behaviour in the event of some more-difficult clustering of data as reported by `read()`. Compare your program’s behaviour to that of the supplied compiled client or `marvin` programs.

(over)

Sequence

I suggest first connecting to a running server, from several different windows, probably from both telnet and chatclient, and also using my compiled “marvin”. Type assorted chat text, and also invoke Marvin’s automated response, with well-formed expressions and with syntax errors.

Then, I suggest implementing in the following sequence:

1. Process the command-line arguments, connect to the server (using the host name lookup code from the supplied lookup.c), and send the handle “Marvin”. (You can test it at this point—look at the server output to see that it got the “Marvin” handle.) You should be checking the banner before sending the handle, but that requires code you’ll write in step 3 (and don’t worry if the server says “Broken pipe” at this point).
2. Use select() to choose amongst server or stdin for next data.
3. Process data from the server similar to myreadline() in chatsvr.c. (*Very* similar.) However, data from stdin can simply be read with fgets(). Use your myreadline()-like code to read the banner, too. A difference is that you have to keep reading until you have the entire banner line, unlike the usual case in which you do only a single read() to avoid blocking. In my solution for banner-checking I simply call this function in a loop until it has an entire line. (At this point you have a complete chat client which you can test.)
4. After printing the data from the server, check whether it has the appropriate form for a request which you need to process.
5. Call parse() and everything, print to stdout if it’s an error, and send to the server in either case.

And remember to Keep It Simple.

Other notes

You will want to begin by making a subdirectory to hold the .c files. You’ll want to copy in the supplied .c and .h files from /u/ajr/209/a4, i.e. “cp /u/ajr/209/a4/*.[ch] .”. To compile your program you will need a command such as “gcc -Wall marvin.c parse.c util.c”. Or, it would be good practice to produce a Makefile; it will not be graded, but you’re welcome to ask me questions about it.

Your program must be in standard C. It must compile on the CDF machines with “gcc -Wall” with the original chatsvr.h, parse.c, parse.h, util.c, and util.h, with no errors or warning messages, and may not use linux-specific or GNU-specific features.

When running a test server, if you get “address already in use”, this might mean that another user is using the same port number on the same machine; *or* it might simply be you. After your program exits, the port number is still marked as in-use for a few minutes to make the behaviour caused by stray packets less confusing. This is why the server has the “-p” option.

Check errors from *every* system call, with the possible exception of write(s) to sockets. You needn’t handle errors in a sophisticated way, but you should at least call perror() for unexpected errors, and you *must not* perform subsequent system calls which no longer make sense given the previous error condition. For example, if socket() fails, do *not* try to bind to file descriptor -1.

As always, keep it simple; avoid frills which introduce bugs or complexity and don’t significantly enhance the value of the program.

Please see the assignment Q&A web page at

<http://www.dgp.toronto.edu/~ajr/209/a4/qna.html>

for other reminders, and answers to common questions.

To submit

Once you are satisfied with your code, you can submit it for grading with the command

```
submit -c csc209h -a a4 marvin.c
```

and the other “submit” commands are also as before. Your marvin.c file will be compiled with the original versions of chatsvr.h, parse.[ch], and util.[ch] for automated testing.