CSC 209 Assignment 4, Summer 2022: A bridge between chat servers

Due by the end of Monday August 15, 2022; no late assignments without written explanation.

In /u/csc209h/summer/pub/a4 you will find source code for a simple internet chat server (chatsvr) and a client for it (chatclient). In this simple chat server, everything which anyone types gets relayed to everyone.

More than one such chat server might be running on the internet at one time, so the person you want to chat with might not be on the same chat server. For this assignment you will write a "bridge" program which will relay messages from one chat server to another when the messages seem to be for someone on the other server.

Your bridge connects to two or more chat servers simultaneously. As messages go by, it notes the "handles" of the people sending the messages. If a message is apparently addressed to someone (as described below), your bridge will relay it to all other chat servers on which it has seen someone with that handle. (Your program itself uses a handle of "bridge" (all lower-case) and does not relay any message which is itself from "bridge".)

The chat server protocol is as follows:

- The server sends a fixed string defined in chatsvr.h, and a network newline
- Next, the client sends its user's "handle", and a network newline
- Future lines from the server are messages in a form suitable for direct display to the client (after converting from the network newline)
- Future lines from the client are messages from that user (ending with a network newline).

A client-side implementation of this is in the distributed chatclient.c, which you will probably be using to test your bridging program (although you can also simply use 'nc' to connect to the chat server, so long as you remember that your first input line must be your "handle").

The server's transmissions are in a user-targetted format; but in accordance with the software tools principle "expect the output of every program to become the input to another, as yet unknown, program" they are designed to be easily parsed. All lines begin with either a user handle or "chatsvr" (which you can just treat as a user handle); then there is a colon, a space, the message, and a network newline.

Your program will only consider forwarding messages which also contain a later comma. The string up to (but not including) the colon is the user name it is from; and the string after the colon and space and up to (but not including) the comma is the user whom the bridge will deem it to be addressed to. If this user has been seen (by you) on any other chat servers, your program sends to each of those chat servers: this user name, space, "says", colon, space, message. (Those other chat servers will prepend "bridge:" to this line before broadcasting it.)

Examples:

```
        Barney: Fred, how are ya?
```

This is a message from Barney to Fred. And if we haven't seen Barney on that chat server before, we will add Barney to our list of known users on that chat server. In any case, if we have seen Fred on any *other* chat servers, we send "Barney says: Fred, how are ya?" to each such chat server.

```
        Barney: Does anyone want to chat?
```

Due to the lack of comma, this message is not deemed by the bridge to be 'to' anyone, although it would still cause us to remember that Barney is on that particular chat server (if we didn't already know that).

```
        Barney: This morning I ate my breakfast, then I went to work.
```

Your program will consider this to be addressed to the user "This morning I ate my breakfast", but since it will presumably not have seen a user by that name on any of its other chat servers, it will not end up relaying the message.

The format of the command-line arguments to chatbridge is a bit odd, but designed to be easiest to use. Chat servers might be on any host name and at any port number. But often, multiple chat servers will be on the same host (in your testing they might often all be localhost). So the syntax is that you say a host name followed by any number of port numbers, then you can repeat with another host name and port number list, and so on. In usage message syntax, this is:

```
        usage: chatbridge {host port ...} ...
```

or with regular expression's '+' notation it would be: (host port+)+

Host names and port numbers are distinguished by the rule that host names must contain a non-digit.

Command-line parsing code is supplied in /u/csc209h/summer/pub/a4/chatbridge.c.starter; you can replace the "connect_to_server()" call if you like. (There is no need to use getopt().)

To convert a host name to an IP address, please see /u/csc209h/summer/pub/a4/lookup.c

You will need to use a linked list to keep track of the list of users seen on each chat server. This can grow arbitrarily over time. If malloc() fails, you can simply exit with an error message.

If the handle of the addressee of a message has been seen on more than one chat server you are monitoring, you will relay to all of them. However, you never relay a message from a chat server to itself. Also, be sure not to relay messages which are from yourself — check specifically for the handle "bridge" and do not relay bridge's messages no matter what.

Other notes

You can compile your program with a command such as

```
        gcc -Wall -I /u/csc209h/summer/pub/a4 chatbridge.c
```

This finds include files in /u/csc209h/summer/pub/a4.

As usual, "gcc −Wall" must not yield any errors or warning messages.

Your program must be in standard C, and may not use linux-specific or GNU-specific features. It can (and should) #include "chatsvr.h"; but it must not rely on any further files.

Check errors from *every* system call, with the possible exception of close() calls and write()s to sockets. You needn't handle these 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 connect using 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 https://www.teach.cs.toronto.edu/~ajr/209/a4/qna.html for other reminders, and answers to common questions.

Submission commands are as in previous assignments and labs. Your file must be named chatbridge.c.

Remember:

This assignment is due at the end of Monday, August 15th, by midnight. Late assignments are not ordinarily accepted and *always* require a written explanation.

If you are not finished your assignment by the submission deadline, you should just submit what you have, for partial marks.

And:

Do not commit an academic offence. Your work must be your own. In each successive assignment, it becomes easier to catch students who have exceeded the bounds of allowable collaboration; more complex assignments mean that different students' correct assignments become more diverse. Recall the admonitions on the course info sheet that **you may not**:

- produce any part of your assignment submission while meeting with others
- look at someone else's assignment work, completed or partial, before the deadline
- show anyone (other than the instructor or a TA) your assignment work, completed or partial, before the deadline (or any extension they have for special circumstances — best to wait until after the instructor solutions are posted)
- type assignment code into a computer with others
- bring your solution, completed or partial, to any group discussion about an assignment
- take away any written or electronic material from any group discussion about an assignment.