CSC 209 Assignment 4, Fall 2012: Network communication

Due by the end of ***Tuesday*** December 4, 2012; no late assignments without written explanation.

In this assignment you will write a program which relays messages, inspired by the children's "telephone game" in which children whisper a message to one another and we see how it gets distorted in the relaying. Your assignment four program will not distort the messages, but rather, will keep track of the random path the messages take until they get back to their originator.

Your program will be called "yak.c" ("telephonegame.c" was too long), and it will communicate with other instances of itself (or other students' or my solutions), which are referred to below as "yakkers".

Yak repeatedly does the following: It either reads a line from the standard input or receives a message from a connected "peer" (using select()). If this message originated with me, or has been circulating on too long a path (there is a "−c" option to set the maximum length of this path), it is displayed to the user. Otherwise, it is relayed to a random one of the connected peers.

Each message is prefaced by a list of IP address/port numbers which show the route the message has taken so far. This list is prepended to each time the message is relayed.

As messages are received (and possibly relayed), we may learn of other yakkers we didn't know about. Since the message contains their IP address and listening port number, we can connect to them to expand our network.

Furthermore, a blank line on stdin means to display a list of all current connections.

New peers may connect at any time (so your listening port number will also be included in the select()).

Some other details

Yak by default listens on port 1234, and optionally connects to another yakker upon startup. The '−p' option makes it listen on a different port number, and the optional hostname plus further optional port number on the command line specify a yakker to connect to upon startup in a syntax similar to that of *telnet* or *nc*.

I have written data structures to maintain the list of connections (see peer.h) as well as to parse the list of relayers in the messages (see parsemessage.h), and a few other useful functions (util.h), and suggested partial yak.c contents (see yak.c.starter). The random choice of peer to send to is also implemented for you in peer.h/peer.c. All of these files can be found in /u/ajr/209/a4 on CDF.

The protocol between the yakkers is very simple, but you do have to follow it, exactly; check the distributed "PROTOCOL" file, and test your yak against mine in /u/ajr/209/a4/yak on CDF.

You can modify peer.h to add fields in "struct peer" as needed. The distributed peer.c must be able to compile with your peer.h; just add the fields in struct peer; don't remove anything, and don't change anything outside struct peer. Your yak.c will also #include both parsemessage.h and util.h (you don't submit those; the supplied ones will be used in testing); and it will be linked with the recompiled peer.c, as well as the distributed parsemessage.c and util.c. Use the distributed Makefile to get this right. Please don't try to add any other .c or .h files, because you can only submit yak.c and peer.h.

Since you can't modify peer.c, you will have to initialize any additional fields in your yak.c when add_peer() returns (after checking that it returns non-NULL). Similarly, you will need to do any required finalization of those fields (hopefully little or none) before calling delete_peer().

For this assignment you do not need to check your write() calls for error. But be sure to check all other system calls except close().

Suggested rough implementation sequence:

1. Read and understand the "PROTOCOL" file which describes the communication protocol between yakkers.

(continued)

2. Make sure you understand the use of the parsemessage and peer modules. Read parsemessage.h and peer.h to see the functions available, and testparsemessage.c and testpeer.c to see example uses of them. Also see extractline() and format_ipaddr() in util.[ch].

3. Examine and understand the starter code in yak.c.starter. Except that you don't need to understand the insides of hostlookup(), which contains some obscure stuff we haven't covered; you can just use this function to turn a name into an IP address, as done in the supplied partial main().

Other than hostlookup(), though, you want to make sure you understand everything in your program. Similarly, if you copy in text from other examples (which you probably will), make sure that you fully understand everything which goes into your yak.c file. If you have bits of your file which you don't understand, you're not going to be able to make it work.

Your program will probably be a bit chatty in terms of discussing new connections from other yakkers, but there will be some kinds of output which would be too much clutter; for example, you might want the option (especially for debugging) of seeing every message which comes in and gets relayed, and every transmission. Please put such print statements in an "if (verbose)"; that's what the −v command-line option is for.

4. Now we begin programming. Make your program listen on port "myport"; also, connect to the specified host and port number if applicable (that is, write doconnect()). To make it compile you will need to add "int fd;" to struct peer; set this fd value some time after you call add_peer(). (You can test doconnect() at this point by connecting to various servers running on computers near you, e.g. the ssh port (port 22), mail transfer port (port 25), web server port (port 80), etc. If you try to connect to a valid hostname and a valid port number for which no server is listening, you should get "connection refused". You can also try it against my 'yak' in /u/ajr/209/a4/yak.)

5. Write the main select() loop. Do a select(), with no timeout (i.e. only the first two parameters to select() are non-null), selecting amongst stdin, the listening file descriptor, and all connected peers (loop through the linked list). If a new connection happens, or stdin input happens, or a peer transmits, for now just output that fact and exit.

6. Test this. Run your program without a hostname and try connecting to it with 'telnet'; type on stdin (and press return); run "nc −l portnumber" and use yak to connect to it and try typing into nc to transmit back to your yak.

7. Make doconnect() send the YAK line with the IP address and port number (the target's IP address but *my* port number). Again, test this by running a listener with "nc -l portnumber".

8. Start implementing appropriate things for when you receive data from a peer. Add fields to peer.h so that you can keep track of data received from each peer; when an entire line has been received according to extractline(), do some processing of it. Note that you need to keep track of a potential line in progress separately in each peer; you can see how I do this in a simple chat server in /u/ajr/209/chat. Try to keep your program compileable and runnable so that you can test incrementally.

9. Add a field in peer.h so that you can keep track of whether you've received the YAK line from the peer. If this flag isn't set and you get a line which begins with "YAK ", pass msg+4 (i.e. skipping the "YAK ") to analyze_banner().

(continued)

10. Start to implement the processing of data from stdin. First of all, implement a blank line, which means to print the list of known peers; this will help you in development. Then, anything else typed on stdin is a message, to be sent on its random way, with my IP address and my port number prepended (then the double-semicolon—see the "PROTOCOL" file). Remember the network newline. You can read stdin with fgets(), but only after select() tells you that file descriptor 0 has read activity pending.

11. By this point you can go your own way in completing the program; but at some time soon you will want to add code to deal with new connections appropriately, calling add_peer() and initializing whatever other fields you've added to struct peer (such as fd). Send the "YAK" line to new peers as soon as you acquire them (and set the flag you added in #9 so that you'll deal with a YAK line from them). To find the IP address of a client whose connection you just accepted, if "&r" is the second parameter to accept then you want the formula `ntohl(r.sin_addr.s_addr)`. Again, try to keep your program useable so that you can test incrementally.

12. You can follow the example in testparsemessage.c to deal with incoming messages, to decide whether to relay them to a new random peer or to output them to your user; and also for how to display the message once you decide to display it. Also remember to check the "relaymax" limit at this point—if there are this many or more peers in the message's history, we don't relay it further, we output it now with an explanatory message.

13. I suggest doing the following point only after everything else is working: After dealing with an incoming message, and either displaying it or relaying it, go through it again and look for any peers that we don't already know about, by trying find_peer() on each of the ipaddr/port number pairs. Also check against the global variables ipaddr and myport. If we see a peer we didn't know about, add a connection to them with doconnect(). This way you will soon have a large and fully-connected network, and the game gets to be much more fun.

Other notes

Your program must be in standard C. It must compile on the CDF machines with "gcc −Wall" with the original auxiliary files as specified above (you can only submit yak.c and peer.h), with no errors or warning messages, and may not use linux-specific or GNU-specific features.

When testing your program, 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 one of the reasons that yak has the "−p" option.

Check errors from *every* system call, with the possible exception of write()s to sockets and close() calls. You needn't handle such 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.cdf.toronto.edu/~ajr/209/a4/qna.html`

for other reminders, and answers to common questions.

Submission commands are as in previous assignments.