Programming Project: Client-Server Application

CPSC 3500 Computing Systems

Spring 2020 v1.0

Overview

In this assignment, you will be creating a client and server program to play Hangman (a simple word guessing game - if you've ever seen Wheel of Fortune, it's a bit like that except players get to choose a letter instead of getting it randomly from a wheel; also players are only guessing a single word). The one key difference in this assignment will be that the player never loses - they just keep making guesses until they get it right, and the program will keep track of how many guesses they took to guess the word.

Their score for the game will be the number of guesses they took divided by the number of letters in the word. (So, scores in this game are like in golf - i.e., lower scores are better.) The ratio helps reduce the advantage of smaller words being easier to guess. Note that this means scores will be floating point numbers, and scores may be less than 1.0 in rare cases. (e.g., it's possible to guess ABBA in two guesses, and 2.0 / 4.0 = a score of 0.5)

Client programs can connect to the server to play the game. Clients send their guess to the server. The server responds by sending the current game status back to the client.

The server will choose a new word at random for each client that connects, so different players get different words and the same player will also get a different word if they connect to play another time. (Well, technically there's a very tiny chance that the same word might get picked again purely by random chance - that's okay.) There is a text file at the following location on the cs1 server that contains a bunch of words, one on each line:

```
/home/fac/lillethd/cpsc3500/projects/networking/words.txt
```

Feel free to hard code that path into your program. (We will use the exact same file for all testing & grading.) You should hard code the complete (absolute) path; do *not* make a copy of this file in your local directory. The file contains **57127 lines**, with one word on each line. Feel free to hard code that number as a constant as well, if you want to.

Each word in the file consists of letters only (so no hyphenated words or possessive words with apostrophes, etc.). The words are also in all capital letters, so you can avoid case sensitivity issues when you make comparisons. Your game should ignore the case of player guesses (or simply require guesses to also be capital letters – but then be sure to check for bad inputs). Finally, the words in this file use the common British spellings, rather than American spellings of words (e.g., colour instead of color) - that doesn't really affect the code, but bear it in mind when you're testing the program and making guesses at words.

Here is the process to play the game:

1. The player (client) is prompted to enter their name and the name is sent to the server.
2. A word is chosen at random for the player to guess. The number of letters in the word is sent to the client.
3. The player may guess a letter.
4. The client sends the guess to the server.
5. The server checks if the word contains the guessed letter, and if so, at which positions in the word the letter is found. (We want to find *all* matches, not just the first match.)
6. The server responds to the client with the result of the guess, which should include:
    - whether the guess was correct or not
    - how many guesses have been made so far (including this one - so the server will return 1 after the first guess)
    - A correct guess should also include all the index positions of where in the word that letter was found. (Be sure to include *all* matches, so there may be more than one index value to return...)
7. The server should also check if the game is won after each guess. The player wins when they have guessed all the letters in the word. If the game is won, continue to step 8; otherwise go back to step 3 and repeat.
8. When the game is won, the server will respond with a victory message that indicates the number of turns it took to guess the word. (Don't forget to include the final guess as a turn!)
9. In addition, the server will send a leader board indicating the name of the top three players along with their scores (lower scores are better).

Hangman Client

Here is a more detailed description of the client:

1. Start the client with two *required* command line arguments (in this order):
    - Required arguments are:
        - IP address of the server
        - port of the server process (use one of the ports assigned to you - see below)
    - Print an error and exit if the necessary command line arguments are not provided, or they are not valid
        - e.g., giving "foo" for the port number would be invalid, since it is clearly not a number
        - Do not spend effort writing complicated code to check if numbers and IP addresses are valid - there are functions available that will produce errors if given invalid data (number of IP addresses), so all you need to do is handle the errors returned from those functions.
2. Connect to the server. Exit with an appropriate error message if a connection could not be established.
3. Ask the user for their name.
4. Send the name to the server.
5. Ask the user for a guess.
6. You may check the user input for validity if you want - but you still need to check it on the server as well, even if you check it here, though! If the guess is invalid, print an error message and go back to step 5.
7. Send the guess to the server.
8. Receive the guess response from the server.
9. If the guess was incorrect, print the number of guesses to the screen and go back to step 5. If the guess was a correct letter guess, the client should print the number of guesses and display *all* letters the player has correctly guessed so far with some character (e.g., - or _) to indicate spaces where the letter has not yet been guessed correctly. (For example, if the word is "HELLO" and the player has previously guessed 'H', 'A', and just guessed 'L', then you would print H_LL_)

Note: The remaining steps are only carried out if the guess was successful.

10. If the player made a correct guess and it was the last letter needed, print the complete word and number of guesses taken to the screen.
11. Receive leader board information from the server.
12. Print leader board information to the screen.
13. Close the connection with the server.

Hangman Server

Here is a more detailed description of the *multithreaded* server:

1. Start the server which has one *required* command line argument:
   - port number (use one of the ports assigned to you - see below)
2. Listen for a client. (You should listen on the provided port number and INADDR_ANY address.)
3. When a client connects, *create a new thread* to process that client (see below).
4. The server will run forever until aborted (Ctrl-C).

The following steps correspond how to process each client:

1. Choose a word at random for the player (client) to guess.
   - Note that each client that connects will get their own unique random word!
2. Print the word to the screen (printed on the server process' output; the client process should never know what the word is until the player has guessed it correctly)
   - Printing the word is important!! While this is not necessary for actual gameplay, it helps both your testing & debugging and my grading. :)
3. The first communication step is to receive the player's name from the client.
4. Then receive a guess of from the client. Check that the guess is valid!
5. If the guess was not the final guess for the word, send the result of the guess and go back to step 4.

Note: The remaining steps are only carried out when the player has won.

6. Send the result of the final guess, as normal.
7. Update the leader board, if the player made the top three.
8. Send the current leader board to the client.
   - Note that the scores are floating point numbers - there are minor complications sending binary floating-point numbers over the network. However, sending strings is relatively easier. Feel free to send the entire leader board as one string (possibly containing newline characters in the string, if necessary).
   - The leader board should be displayed with scores to two decimal places (although position in the leader board may be determined by the complete floating-point values – so apparent ties may not really be ties).
9. Close the connection with the client.

Leader Board

A few notes about the leader board:

- When the server is started, the leader board initially starts empty.
- If the leader board contains fewer than three entries, only display the current number of entries.
- Ordering in the event of a tie is unspecified - i.e., either ordering option is okay. (Since scores are floating point numbers, we expect exact ties to be extremely rare.)
   - This means you should probably consider handling ties in whichever way is easiest / least complicated to code - don't make it more complicated than necessary!
- The output of the client should display each person on the leader board on its own line with each line displaying the user name and the score (as shown in the example below.)
- Did you notice that the data structure storing the leader board is shared among the different client threads? *Your program should protect against any issues that may arise from sharing the leader board.*

```
Leader Board

Huey 1.20
Dewey 1.95
Louie 2.67
```

Compiling and Running Your Program

You will need to create your own Makefile that compiles your client and server programs. (You've worked with a few of them now, so go back and look at those. Feel free to copy-and-modify from the Makefiles you've used before.) Since you are creating two programs, it is necessary to use the `-o` option in `g++` to give your client and server different names: `game_client` and `game_server` respectively. (Your programs must have those exact names in order for the submission to work correctly.)

To avoid accidental collisions between people using the same port numbers (e.g., your client accidentally connects to someone else's server, or perhaps even more strange behaviors...) each student in the class has been assigned a range of port numbers. (Your assigned port numbers are in a separate document posted on Canvas.) *You may only use the port numbers assigned to you.* ***Ever.*** You have been given several more than you need, so feel free to rotate between your assigned ones as needed; you should never have need of any others.

To run your server and client, it is necessary to have two terminal windows open. In one window, start the server (the server must be started before the client). Then run the client in a different window. The IP address for cs1 is 10.124.72.20 Use the ports assigned to you. Here are some sample command lines assuming one of your assigned ports is 10670:

```
./game_server 10670
```

```
./game_client 10.124.72.20 10670
```

Before submitting your program, you should test your server with two or more clients connected to the server at the same time. Each client will need their own window.

If either program terminates (either Ctrl-C or program error) while the connection is still active, you may get a bind error. If this happens, select a different port. In this situation, the OS thinks the port is still being used and it takes a minute or two for the OS to figure out the process using that port has terminated. (You have been assigned a large number of ports so that you can switch between using different port numbers as this happens. Since both programs take the port number on the command line, it should be easy to switch the port number you're using at any time.)

Sample Playing of the Game

Your client output does not have to match this exactly - this is just to give you an idea of how the interaction described above works. For this sample playing, assume that the word is "KERNEL".

```
Welcome to Hangman!
Enter your name: Alice

Turn 1
Word: ------
Enter your guess: E
Correct!

Turn 2
Word: -E--E-
Enter your guess: M
Incorrect!

Turn 3
Word: -E--E-
Enter your guess: n
Correct!

Turn 4
Word: -E-NE-
Enter your guess: k
Correct!

Turn 5
Word KE-NE-
Enter your guess: L
Correct!
```

```
Turn 6
Word KE-NEL
Enter your guess: r
Correct!

Congratulations! You guessed the word KERNEL!!
It took 6 turns to guess the word correctly.

Leader board:
    1. Mark  0.83
    2. Alice 1.00
    3. Beth  1.25
```

Implementation Notes

For the most part, it is up to you to determine how the various information is transmitted over the network. You may assume that the client only connects to a program running your server and vice versa. However, there are a few rules:

- Any numeric values must be sent *in network byte order* (unless they are part of the leader board, which may be transmitted in its entirety as one string).
- You may send strings using any method you would like, subject to the following limit assumptions.
- Limit assumptions:
    - Player names are allowed to be at least 1000 characters longs (allowing longer is OK, but don't limit their length to any less than this)
    - Words should be allowed to be at least 1000 characters long
    - The number of guesses a player takes should be allowed to grow to at least 1000
- The leader board can be sent as a series of records or as a long string (the latter is easier).

- The server must generate a new word for each client.
- The server must validate any data it receives over the network (from the client) to make sure it doesn't contain anything unexpected (this is much the same as you would normally validate input taken from a user via `cin`). *The server must validate data received from the client even if the client already validated it when it was input by the user!* (The server cannot ever assume that the client did its job properly or that the client is well behaved.) It is recommended, but not strictly required for grading, that the client do the same with data it receives from the server.
- If the server does detect an error in the data sent by the client, it should return an error message to the client indicating so, and ignore the message that the client sent (i.e., it should wait for the client to re-send the same type of message but with valid data).

The client output is what the user sees, so it should look presentable to a customer who might want to buy your game! However, it is permissible for the server to print an occasional status message (such as when the client connects). In fact, it is a requirement that the server print out the word at the start of each game. Just be sure that you don't have so many server log messages that the word itself scrolls off the screen before the client is done playing.

Error Checking

- Since the server runs forever, it must be robust. In particular:
  - Any unrecoverable error that occurs when processing a client causes only that thread to end. These types of errors should not cause the server as a whole to crash.
    - You may use the `pthread_exit()` function or simply return from the thread_routine
  - The server should be free of memory leaks – dynamically allocated resources need to be reclaimed.
  - Sockets must all be closed at an appropriate time. The sockets used to communicate with the individual clients must *not* be left open forever. (This is avoiding a different kind of resource leak.)
- The server process should abort with an appropriate error message *only* when an error occurs that would prevent listening/accepting/processing any further client connections.

Submitting Your Program

On cs1, run the following script in the directory with your program:

```
/home/fac/lillethd/cpsc3500/submit/submit_networking
```

This will copy ***all** source code files* in the current directory (must have a .cpp, .c or .h suffix) as well as the **Makefile** to a directory that accessible by the instructor. (Be sure the Makefile is spelled correctly and the M is capitalized.) If a Makefile is not present, the submission script will reject your program.

The submission program will then attempt to compile your program using make and will look for executable files **`game_client`** and **`game_server`**. If make fails and/or the required executables are not present, the submission program will reject your submission. If you are unfamiliar with using make and/or creating make files, please consult the tutorial on the class website.

Your program must properly compile, or the submission program will reject your program. **Programs that fail to compile will not be graded and receive a 0.** Only the last assignment submitted before the due date and time will be graded. To resubmit, just run the submission script again, but only your last submission counts and all earlier submissions will be permanently overwritten! **Late submissions are not accepted and result in a zero, unless covered by free passes.**

The submission script will clearly display a message containing either the word "SUCCESSFUL" or "FAILED" – you should always see exactly one of those words in all caps (never both or neither).