CS 3251 Project 2: Hangman Game

1. Overview

The aim of this project is to have you write a game-server and a player-client for a simple hangman game. You have to develop a game-server and a client(player) that establishes connection with the game-server and plays a game of hangman. You will first implement a single player version and then add support for two player games. The client will have the option to choose between single and multiplayer mode. The client will be much more simple than the server. This project is much more complicated than the first so don't wait until the last minute. Please read through the whole description before starting. You are allowed to work with at most one other student.

2. Details

First, a (perhaps obvious) note on command syntax for this instruction:

- "$" precedes Linux commands that should be typed or the output you print out to the stdout.

2.1 Server - Hangman Game Server (Single player)

- The server should take in a port number as it's only command line arg.
- The server should be able to manage 3 games at any given time.
- Each client will be playing their own separate game.
- The server should handle all administrative duties of each client's game such as choosing the word and keeping track of the score.
- The server will tell the client if the guess is correct or not and keep track of the number of guesses the client has made.
- Loop listening for connections
    - When server starts, a connection handler starts listening for incoming game request connections from the client.
    - Server keeps listening during and after play.
- On successful connection with a client the game-server should choose a game word and wait for the packet sent from the client signaling the game should start.
- It should then respond using a game control packet (seen below), the client will add the formatting and display the following to the player:
    - For the game word "hangman"

      $ _ _ _ _ _ _ _
      $ Incorrect Guesses:
- After a client guesses a letter the server replies back with the updated word, filling in correct letters and supplying incorrect guesses.

    - For example, if the client has guessed “n” (correct guess). Server responds with:
      $ _ _ n _ _ _ n
      $ Incorrect Guesses:
    - Client then guesses “p” (incorrect). Server responds with:
      $ _ _ n _ _ _ n
      $ Incorrect Guesses: P
- When the game is over tell the client if they won or lost then terminate the connection with that client.
    - Game ends when client guesses **6** incorrect letters or if all correct letters for the word have been guessed.
- The server should choose a word at random for each game. Please choose **15** total possible words for your server to choose from.
    - The words must range in size from a **minimum length of 3 letters to a max length of 8**.
    - You must choose words of **at least three different lengths**. For example, five 4-letter words, five 5-letter words, and five 8-letter words
    - Each word must have an equal chance of being chosen by your server for a new game.
- If a client (player) tries starting a 4th game, the server should send “*server-overloaded*” message (please follow the server message format in §2.3) then gracefully end the connection.
- After ending a game with a client the server should be able to accept a new connection from a client until the limit(3) is reached again and so on.

2.2 Client - Player (Single player)

- Use server IP and Port to connect to a game server
- Once connected ask user if they are ready to start the game as follows:
  $ Ready to start game? (y/n):
    - If the user says no terminate client otherwise send an empty message to the server with msg length = 0. (more on this below) signaling game start.
- The server will then respond with it’s first game control packet which the client should properly print out.
- The client will then ask the player for a guess exactly as follows:
  $ Letter to guess:
- Client then sends the users guess to the server. **Make sure that the user guess is formatted correctly**.
    - If the user guesses something invalid like a number or more than one letter please respond with an error then ask for input again as follows:

      $ Letter to guess: 5
      $ Error! Please guess one letter.

$ Letter to guess: aa
$ Error! Please guess one letter.
$ Letter to guess:

- If the user guesses something that has been guessed before (i.e., some duplicate letter), please respond with an error then ask for input again as follows:

  $ Letter to guess: a # a has been guessed before
  $ Error! Letter A has been guessed before, please guess another letter.
  $ Letter to guess:

- The player keeps guessing letters while the server responds with updated game information.
- Continue this process until the game word is either correctly guessed or the guess limit is exceeded (6 incorrect guesses).
- Client should terminate gracefully when the game is over.
- Client should be able to accept lower or uppercase letters as a guess but must make sure to convert all guesses to lowercase before sending to the server.

2.3 Server Message header

When sending messages to a client the server must use the following format. **Up to 2-point penalty, if failing to strictly follow this format**.

Msg flag	Word length	Num Incorrect	Data

Msg flag (1 byte) is used when the server would like to send a string message to a client. If the server is sending a message to the client the Msg flag will be set to the length of the message otherwise Msg flag will be zero. If the message flag is set then there are no Word length or Num Incorrect fields the entire packet minus the message flag is message data. When a client receives a message from the server it should directly print it out. The client should print messages out in the order they are received.

-An example of a message could be “Game Over!”, “You Win!”, “You Lose :(“

If the message flag is zero then we have a normal game control packet.

Word length (1 byte) is the length of the word the user is trying to guess.The first n bytes in the data segment of the packet where n is the value of Word Length will be used to store the positions of the letters that have been guessed correctly so far.

Num Incorrect (1 byte) is the number of incorrect guesses a client has made. The incorrect guessed letters will be after the game word in the data segment.

Two example data packets can be see below. In the example each box is one byte.
This packet would be a game control packet with the game word being “green” and the user having guessed g, and e correctly. The user has guessed a, b, and c incorrectly.

Msg flag	Word length	Num incorrect	Data [1]	Data [2]	Data [3]	Data [4]	Data [5]	Data [6]	Data [7]	Data [8]
0	5	3	g	_	e	e	_	a	b	c

This packet is an example of a message packet. When the client receives a message packet it should directly print out the contents.

Msg flag	Data [1]	Data [2]	Data [3]	Data [4]	Data [5]	Data [6]	Data [7]	Data [8]
8	“Y”	“o”	“u”	“ “	“W”	“i”	“n”	“!”

2.4 Client Message Header

This defines the message the client will send to the server. **Up to 1-point penalty, if failing to strictly follow this format**.

Msg Length	Data

An example packet could be the following when a client is guessing “a”.

Msg Length	Data
1	“a”

2.5 Example Program flow

The following is an example game output from a client connecting to a game server at the address 127.0.0.1 and on port 8080. The server chose the game word “dog”.

```
$ ./client 127.0.0.1 8080                  # Connect to server
$ Ready to start game? (y/n): y    # Client asks if player is ready to start and sends empty
                                   # message (“0”) to server signaling game start
$ _ _ _                            # Server responds with first game control packet
$ Incorrect Guesses:
$                                  # Print new line after Incorrect Guesses for readability
$ Letter to guess: a               # Client asks for input and user types “a”
$ _ _ _                            # Server responds
$ Incorrect Guesses: a
```

```
$
$ Letter to guess: d                # Client asks for input and user types “d”
$ d _ _                             # Server responds
$ Incorrect Guesses: a
$
$ Letter to guess: g                # Client asks for input and user types “g”
$ d _ g                             # Server responds
$ Incorrect Guesses: a
$
$ Letter to guess: o                # Client asks for input and user types “o”
$ d o g                             # Server responds
$ Incorrect Guesses: a
$ You Win!
$ Game Over!
```

If the player instead guesses 6 incorrect letters and loses the game replace “You Win!” with “You Lose :(“

2.6 Multiplayer Support

- Finally you will add support for a multiplayer version of the hangman game.
- The limit of 3 games still stands here - so you should be able to handle up to 6 clients connected to your server at once.
- When the client first starts instead of asking if the player is ready to start the game ask if the the game will be 2 player.

  $ Two Player? (y/n):
- If the player responds no continue as normal by sending server “0”
- If the player says yes send server “2”
- The server will then respond to the client right away with a message packet saying “Waiting for other player!”
  - The first client to request a 2 player game will be player 1
- The server will wait for another client to request a two player game.
  - This player will be player 2
- When the server receives a second request it will initialize a game as normal and will send a message to both players containing “Game Starting!”
- The server will then send a message to player 1 saying “Your Turn!” followed by a normal game control packet and the client will ask for a letter.
- Once player 1 guesses the server will respond with a string message containing either “Incorrect!” or “Correct!” depending on if the letter guessed was correct. An example of what player 1 would see follows. Let’s say the game word is “green”.

  $ Game Starting!
  $ Your Turn!
  $ _ _ _ _ _
  $ Incorrect Guesses:

$
$ Letter to guess: e
$ Correct!
$ Waiting on Player 2...

- Player 2 is then up. Player 2 would have seen the following.

$ Game Starting!
$ Waiting on Player 1...
$ Your Turn!
$ _ _ e e _
$ Incorrect Guesses:
$
$ Letter to guess:

- The game then follows this pattern until the two players reach a combine **6** incorrect guesses or they correctly guess the word. The game will then end similar to a single player game.

3. Logistics

Students can work in **groups of 2 max** for this project.

You need to implement two separate C/C++ (or Python, Java) programs: one for the server (**server.x**) and another for the client (**client.x**). Note that, here "**server.x**" could be **server.c**, **server.cpp**, **server.py**, or **server.java**, it depends on which language you choose. Same rule applies to "**client.x**".

- Server program should accept one **required** input parameter: the port number. For example, if we want the server to listen to port number 2017, we will start your program by doing "./server 2017".
    - You may add the name of a file containing your 15 game words as an second **optional** input param
- Client program should accept two required input parameters: the server ip and port number. For example, if the server ip is 127.0.0.1 and port number is 2017, then we will start your program by doing "./server 127.0.0.1 2017"

3.1 Deliverables

- You need to submit one compressed folder containing
    - Server.x
    - Client.x
    - Makefile (if you use C/C++/Java)
    - README
    - Sample output of a game

- In the README please give a high level description of your implementation ideas.

    - Please include the names of both team members and an explanation of how the work was divided between the team members (*i.e.* who did what).
    - You should explain clearly how to use your code.
    - You should explain how to install dependent packages for being able to test your code.
- Note that, We will compile your program by just doing “make”, if you use C/C++/Java.
- Please comment your source code.

3.2 Grading

This is not a performance-oriented project; we will test the functionality only. The Rubric will be:

1. Server functionality (single player) - 6 points (**lose up to 2 points if failing to follow the message format**)
2. Client functionality (single player) - 2 points (**lose up to 1 points if failing to follow the message format**)
3. README + Makefile (if using C/C++/Java) - 2 points (**lose up to 6 points, if no Makefile and README**)
4. Multiplayer mode - 2 points

3.3 Implementation Tips

Here is an example of how to support multiple clients connecting to the same server:
Server: geeksforgeeks
You can see a very simple client example here: stackoverflow

The server side code uses select to support multiple clients connecting to the same server. Note that, this is not the only solution or the best solution, but maybe the simplest solution if you are not familiar with multi-threading.

If you have experience or want to learn about multi-thread programming, you can use multiple threads to handle your clients for this project. If going this route you may want to check out the pthreads library and skimming through this guide may be helpful as well. There are tons of other online resources on how to approach this so as always Google is your friend here!