CSCI 379

Computer Networking

Programming assignment #2

Due by 11:59pm, Sunday, May. 27th, 2018

In this project, you will implement a simple game of hangman (https://en.wikipedia.org/wiki/Hangman_(game)) using socket programming in Python or Any other language you choose to send and receive messages between a client program and a server program.

> Hangman is a paper and pencil guessing game for two or more players. One player thinks of a word, phrase or sentence and the other tries to guess it by suggesting letters or numbers, within a certain number of guesses. (via Wikipedia)

You're going to build a set of server and client such that server will generate words and determine whether the players win or lose the game, and the players will play the game on the client side that send its guess to the server. The protocol for this program will be described in this document. This project is adopted from CSE422@MSU (https://www.cse.msu.edu/%7Edennisp/cse422).

For this project, you may work as a group of two. In that case, you and your partner need to submit a copy of code and document, and in your README document, you need to list the team members(you and your partner) and their responsibility.

1. Outline of project

The project2_skeleton.zip file is provided to you. It contains:

- extra_credit/ (folder for your extra credit parts, leave it empty if not attempted)

- Project2.pdf (This document)
- README.txt
- src/
    - server.py
    - client.py
    - game.py (Don’t modify this)
    - words.txt (Don’t modify this)

Your need to complete the skeleton code in server.py and client.py to successfully play a simple game of hangman. (The hangman game logic is mostly done for you, so you need to mainly focus on sending messages back and forth using TCP and UDP sockets). You don't necessarily follow the entire structure of the skelton code, feel free to make any change on it, but you have to follow the protocols described below such that any correctly implemented client and server can interact with yours, e.g., my client is supposed to be able connect with your server program and play without any issue.

The idea is the client will contact the server by making a connection using TCP. The client will then announce itself to the server. The server responds with the UDP port the client should use for further communication. Finally, send game messages back and forth using UDP. The client repeatedly guesses letters by sending them to the server. The backend (server) handles the game logic by checking the guess and replacing the blanks with correctly guessed letters. It also decrements the number of attempts left in the case of an incorrect or already guessed letter.

Each message has 2 parts: the message type and the message text. Each message will be of the form `<message type><sp><message text>` (notice the space character in between). The message type will dictate to each program what to do next (this is outlined in Section 6 below). An example exchange of game messages is shown in Figure 1 for your reference.

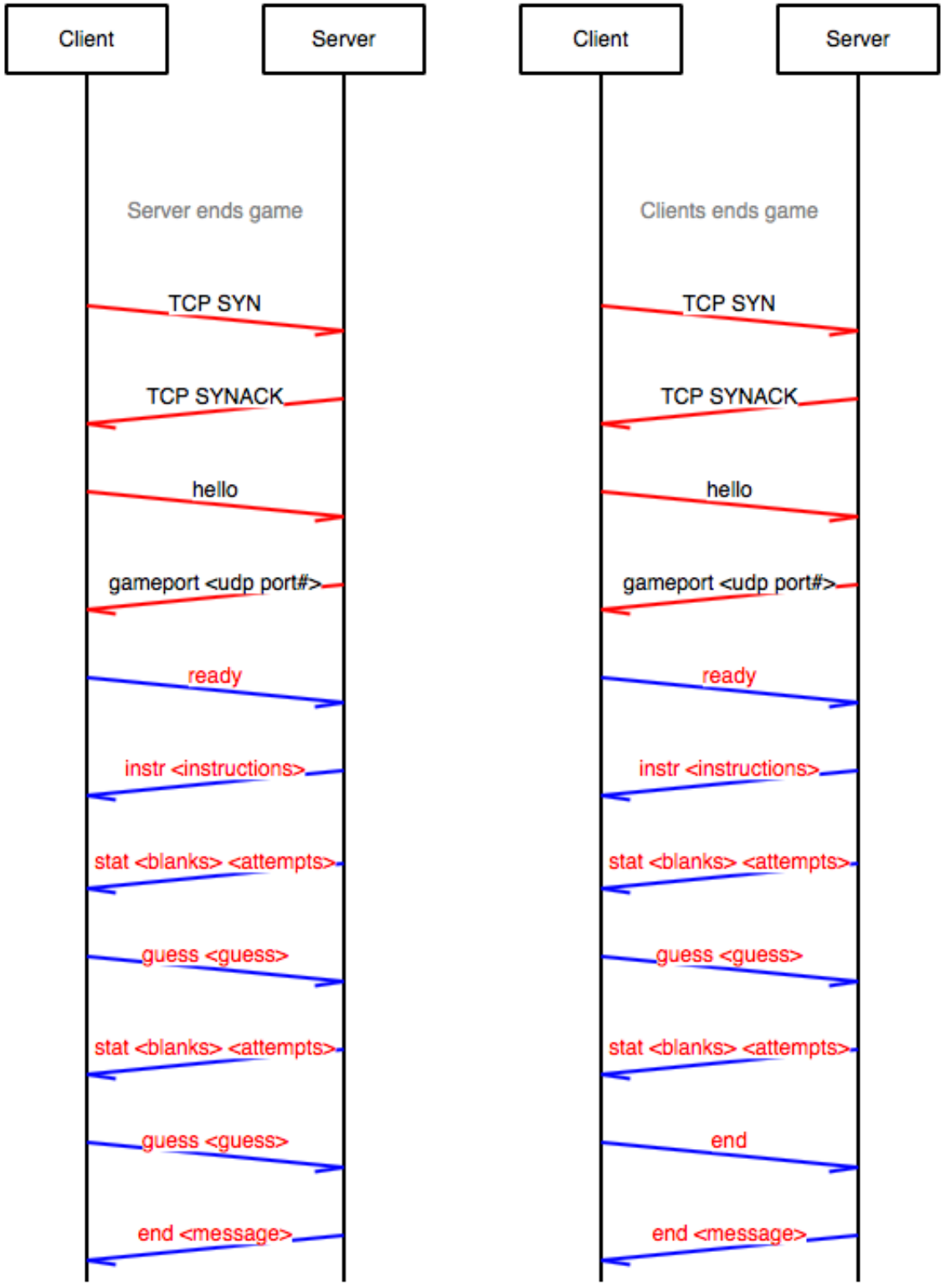

Figure 1. Game Messages Visualization(red lines indicate TCP messages, blue lines indicate UDP messages)

You should modify the README.txt with your information, a list of resources you used, additional comments, and some sample test output from your programs (both client and server). If you work as a group of two, you need to list your partner and how do you divide up the project, what are the responsibilities of each of you?

2. Client Specifications

The client should accept two command line arguments:

1. Name of the server host
2. TCP Port to be used to contact the server process

Example: `python3 client.py localhost 15678`

When the client is first started it should prompt the user for their name. After the user has entered their name, the client should create the TCP connection to the server and send the `hello` message with the user's name. Once the client receives the `gameport` message from the server, the client should open a UDP socket and then prompt the user and allow them to enter various game playing commands (these are different from the network messages). The client should support the game commands being entered in any case (upper, lower, or mixed case). All additional communication with the server should be performed using UDP messages to the port provided by the `gameport` message.

The player (user) interacts with the client program by entering game playing commands through their keyboard. The client should support the following game playing commands:

- start – Starts a new game.
- end – Ends the current game. If a game is not already in progress this command has no effect. Sends an `end` message to the server indicating that the player does not want to make any more guesses. The client then can start a new game or exit.
- guess – Used by the user to enter a guess. The user can guess either a single letter that might be part of the game word or if more than one letter is entered then the user is trying to guess the entire game word. • exit – Sends a `bye` message to the server, ends the client process.

3. Server Specifications

The server should accept a single command line argument. This argument should either be the option `-r` or a word. If the argument is `-r` then a client starts a new game and the word to be used for that game is randomly chosen from those contained in the supplied file words.txt. If the argument contains a word then that word will be used for all games started by a client. The server will only support one game at a time, but a client can play multiple games one right after another. The client can end a game at any time by sending an end message. The game also ends when the player either

accurately guesses the word or uses all their guesses. When started the server should create a TCP socket to be used for connections from clients. When creating this socket, it should have the system dynamically allocate an available port. The server should then display this port number. This is how clients know which port to use when contacting the server.

Examples:

```
python3 server.py -r
python3 server.py applejacks
```

After the client connects to the server, a UDP socket should be created and its port number should be sent to the client using the TCP connection. All further communication should be done using UDP messages. Next section describes the action which should be taken when message types are received. The server should have a timeout action on the UDP socket so that if a client disconnects or stops playing, i.e., the server does not receive a message from the client for 2 minutes (120 seconds), the server can service other clients.

4. Messages and Message Actions

Message Type	Who sends / Receives	Description
hello	Client to Server	Used by the client to initiate a session and tell the server the player's name. Example: `hello Joe`
bye	Client to Server **or** Server to Client	Used by the client to tell the server its exiting. The server should respond by sending a corresponding bye message.
gameport	Server to client	Used by the server to tell the client what UDP port should be used for all subsequent communications. The server when creating this port should use a dynamically allocated port. Example: gameport 59875
ready	Client to server	Sent by the client when the user starts a new game. This message only consists of the command word "ready". It does not have any additional command arguments
instr	Server to	Sent by the server to provide game play instructions. Client should display the

	client	text following the message type word. Example: instr This is how you play the game.
stat	Server to client	This message has two command arguments following the command word: and . is the combination of dashes and letters that represent the hidden word and is the number of attempts the user has left to guess letters before losing.
guess	Client to server	Sent by the client when the player makes a guess. There is one message argument following the message type. This is either a single character or a full word which is the guess entered by the player. Examples: `guess e` or `guess apples`
end	Server to client **or** client to server	When sent by the server to the client, this indicates that the server has ended the game. The client should display the message following the message type to the player, then prompt and wait for the next game command.When sent by the client to the server, this message only contains the message type. There are no message arguments. The server should end the game and respond with its own end message.
na	Server to client	Sent back to the client if the server receives an unknown message type (or possibly known message type like guess, but no game is currently started).

The server should do the following when receiving various message types:

- hello – The server now knows the name of the user. The server should send its UDP port number to the client.
- ready – Indicates to the server program that the client has a UDP port set up and is ready to start playing the game. Send the instr message followed by a stat message with the initial word blanks and attempts.
- guess – Includes one or more characters in its message text. Upon receiving a guess message, the server will use the provided checkGuess() function to update the word*blanks, attempts, and win variables. If ending conditions are met (a. guess was more than one character and it wasn't the correct word, b. there are no more attempts, or c. the player won the game), an end message will be sent with either a win or loss message text. Otherwise, if ending conditions are not met, a stat message will be sent with the updated word*blanks and attempts variables.
- end – Ends the current game. The server should respond with a corresponding end message containing a message to the player.
- bye – Ends the current game, sends a bye message back to the client, closes the udp port and TCP connection, waits for a new client TCP connection.

The client should do the following when receiving various message types:

- gameport – Extract and store the server program's provided UDP port number. Then prompt the player for a game command.
- instr – Simply display the message text and wait to receive the initial stat message.
- stat – Display the message text then prompt for a game command from the player.
- end – Display the message text, then prompt for a game command from the player.
- na – Display the message text and then prompt for a new game command.
- bye – Display the message text and end the program.

6. Sample Output

Example: Client Output

```
->  python3 client.py localhost 54751
Client is running...
Remote host: localhost, remote TCP port: 54751
Please enter your lovely name:CJ
The server address is 127.0.0.1:54751
Received UDP port#: 61474
>start
This is hangman. You will guess one letter at a time. If the letter is in
the hidden word, the "-" will be replaced by the correct letter. Guessing multiple letters at
a time will be considered as guessing the entire word (which will result in either a win
or loss automatically - win if correct, loss if incorrect). You win if you either guess all of
the correct letters or guess the word correctly. You lose if you run out of attempts. Attempts
will be decremented in the case of an incorrect or repeated letter guess.
Word: ---- Attempts left: 5
>guess v
Word: v--- Attempts left: 5
>guess b
Word: v--b Attempts left: 5
>guess t
Word: v--b Attempts left: 4
>guess r
Word: v-rb Attempts left: 4
>guess e
You win! Word was verb.
>start
This is hangman. You will guess one letter at a time. If the letter is in
the hidden word, the "-" will be replaced by the correct letter. Guessing multiple letters at
a time will be considered as guessing the entire word (which will result in either a win
or loss automatically - win if correct, loss if incorrect). You win if you either guess all of
the correct letters or guess the word correctly. You lose if you run out of attempts. Attempts
will be decremented in the case of an incorrect or repeated letter guess.
Word: ----- Attempts left: 5
>end
Good luck! Word was cloud.
>exit
Closing TCP and UDP sockets...
```

Server Output:

```
>python3 server.py -p
```

```
->python3 server.py -r
Server is running...
Creating TCP socket...
Server is listening on port 54751
Waiting for a client...
A new client is connected to the server!
User's name: CJ
Creating UDP socket...
UDP socket has port number: 61474
Sending UDP port number to client using TCP connection...
Hidden Word: verb
Starting game...
Guess is v
Correctly guessed char
Attempts left: 5
Win status: False
Sending message: stat Word: v--- Attempts left: 5
Guess is b
Correctly guessed char
Attempts left: 5
Win status: False
Sending message: stat Word: v--b Attempts left: 5
Guess is t
Incorrectly or already guessed char
Attempts left: 4
Win status: False
Sending message: stat Word: v--b Attempts left: 4
Guess is r
Correctly guessed char
Attempts left: 4
Win status: False
Sending message: stat Word: v-rb Attempts left: 4
Guess is e
Correctly guessed char
Attempts left: 4
Correctly guessed word
Win status: True
Sending message: end You win! Word was verb.
Hidden Word: cloud
Starting game...
User gives up.
```

```
Client exiting...
Waiting for a client...
```

5. Assignment Notes and Other Requirements

- `python 3` is recommended to be used in this project. The encode() and decode() methods should be used for converting messages between Unicode and byte representation.
- Find more information on socket `settimeout`, `bind` from Python Socket documentation (https://docs.python.org/3.6/library/socket.html)
- The Python split() method "splits" a string into a list with the space character as the default delimiter
    - Use try/except for cases where list indices may cause an IndexError
- Remember to close sockets at the end of each program
- Use try/except to catch errors whenever possible. (i.e. timeouts, ctrl-c keyboard interrupt, unexpected IndexError)

6. Grading Rubric

(90%) Program submitted shall be syntax-error-free, ready to run and implement all the features described above. Any programming language can be used, however, any network library other than socket cannot be used.

(10%) Complete the README document. Your code should be well formatted and documented. If you are not using python, you need to supply instructions on how to build and run it.

You need to finish all the work above before you work on this part.

Extra credit:

(20%) Survey some(at least two) techniques/functions that are commonly used for computer networking program to support concurrent connections and write a short essay on each of them.

(60%) In this design, the game server can only serve one player at a time, since the TCP connection is blocked after one player connected. Can you modify it to support multiple players? So that, two or more instances of your client program can be running at the same time connecting the same server and play the game simultaneously.

(10%) Briefly desribe the design of your server. What change had you made to the original one that make multi-player feasible? Which technique have you used for concurrent connections?

(10%) You code should be well formatted and documented.

Submission guidance

Things to submit: make a **zip** file named as `csci379_prj2_yourlastname.zip` (it will be `csci379_prj2_chu.zip` for me) including the everything from the skelton folder and extra credit part if attempted.

For this project, you may work as a group of two. In that case, each of you need to submit a copy of code and document, and in your submission and document please mention who you are parter with.

Submit the zip file through blackboard.