Project 1: Socket Basics

Get started writing code using network sockets

This project is due at 11:59pm on Monday, January 22, 2024.

Description

This assignment is intended to familiarize you with writing simple network code. You will implement a client program that plays a variant of the recently-popular game Wordle. Your program will make guesses for the secret word, and the server will give you information about how close your guess is. Once your client correctly guesses the word, the server will return a *secret flag* that is unique for each student. If you receive the *secret flag*, then you know that your program has run successfully.

Your client must support TLS encrypted sockets. The server will return different secret flags depending on whether your client communicates with or without TLS. To receive full credit on this project, you must turn in **both** secret flags: the one retrieved via a non-encrypted socket, and the one retrieved via a TLS encrypted socket.

Language

You can write your code in whatever language you choose, as long as your code compiles and runs on Gradescope. Similarly, your code must compile and run on the command line. You may use IDEs (e.g., Eclipse) during development, but do not turn in your project without a Makefile. Make sure you code has **no dependencies** on your IDE.

Protocol

The server runs on the machine `proj1.3700.network` and listens for non-encrypted requests on a TCP socket bound to port `27993` (it also listens for TLS encrypted requests on a TCP socket bound to port `27994`). This exercise has four types of messages: `hello`, `start`, `guess`, `retry`, and `bye`. Each message is an ASCII string consisting of multiple fields separated by spaces (`0x20`) and terminated with a line feed (`0x0A`, `\n`). Messages are case sensitive.

All messages in the game, both from the client and the server look like:

```
<JSON formatted data>\n
```

The data is therefore simply a JSON-formatted object, followed by a `\n` character. To simplify the protocol, you can assume that no `\n` character will appear inside of the JSON data. However, the order of fields in the JSON data may differ from what you see below; as long as the JSON data is correctly formatted and has all of the fields—and only the fields—specified in the protocol, it will be accepted by the server.

```
{"type": "hello", "northeastern_username": <your-My.Northeastern-username>}
```

In your client, you should replace `<your-My.Northeastern-username>` with your Northeastern username as a string (thus it will be in double-quotes [`"`]). For example, if your Northeastern email address was `a.mislove@northeastern.edu`, your Northeastern username would be `a.mislove`. You must supply your username so the server can look up the appropriate *secret flag* for you. The server will reply with a `start` message. The format of the `start` message is:

```
{"type": "start", "id": <string>}\n
```

In this message, the `<string>` in `id` field is a random string selected by the server that represents this game ID; your client needs to remember this ID for future messages. Upon starting a game, the server will select a new, random word as the secret word. At this point, your client can start making guesses for the secret word with `guess` messages. The `guess` message looks like

```
{"type": "guess", "id": <string>, "word": <string>}\n
```

Where the `<string>` in the ID field is *exactly the same* string as was provided to you in the `start` message. The `<string>` in the `word` field is your client's guess. **Note: Your client can *only* guess words that appear on the official word list.** The server will then respond with either a `retry` or `bye` message. If your guess was incorrect, the server will respond with a `retry` message, which will look like:

```
{"type": "retry", "id": <string>, "guesses": [{ "word": <string>, "marks":
```

The `<string>` in the `id` field is exactly the same as was provided in the `start` message. The `guesses` field is an array representing a history of all of your guesses, along with the `marks` of each. Each `marks` field is *itself* an array of length 5, with each element containing one of three values:

Value	Meaning
0	Letter does not appear in the secret word
1	Letter appears in the secret word, but not in this position
2	Letter appears in the secret word in this position

in other positions unless that letter appears in the word multiple times.
- Guesses are case-sensitive, and the word list has only lowercase words.
- After your client has made too many incorrect guesses in a single game (500), the server will send an error and close the connection.

If your guess was correct, the server will respond with a `bye` message, which will look like:

```
{"type": "bye", "id": <string>, "flag": <string>}\n
```

Once your program has received the `bye` message, it can close the connection to the server. The `<string>` in the `flag` field is your client's secret flag. Write this flag value down, since you need to turn it in along with your code.

If, at any time, the server determines that your client has sent an illegal message, it will respond with

```
{"type": "error", "message": <string>}\n
```

where the `<string>` in the `message` field describes the nature of the error. In particular, if the error message is `Unknown username`, that means it did not recognize the Northeastern username that you supplied in the `hello` message.

Your client program

Your client program must execute on the command line using the following command.

```
$ ./client <-p port> <-s> <hostname> <Northeastern-username>
```

Your program must follow this command line syntax **exactly**, i.e., your program must be called *client* and it must accept these two optional and two required parameters in exactly this order. If you cannot name your program *client* (e.g., your program is in Java and you can only generate *client.class*) then you **must** include a script called *client* in your submission that accepts these parameters and then executes your actual program. Keep in mind that all of your submissions will be evaluated by grading scripts; if your program does not conform **exactly** to the specification then the grading scripts may fail, which will result in loss of points.

- The `-p port` parameter is optional; it specifies the TCP port that the server is listening on. If this parameter is not supplied on the command line, your program must assume that the port is 27993.
- The `-s` flag is optional; if given, the client should use an TLS encrypted socket connection. If this parameter is supplied on the command line and `-p` is not specified, your program must assume that the port is 27994.
- The `hostname` parameter is required, and specifies the name of the server (either a DNS name or an IP address in dotted notation).
- The `Northeastern-username` parameter is required.

Your program should print **exactly one line of output** containing **only the *secret flag*** from the server's BYE message. If your program encounters an error, it may print an error message before terminating. Your program **should not write any files to disk**, especially to the *secret_flags* file!

Example transcript

To give you a better idea of what the exchange between the client and server might look like, please consider the following message exchange, between the client `C` and server `S`, where the secret word is `steam`:

```
C -> S: {"type": "hello", "northeastern_username": "a.mislove"}\n
S -> C: {"type": "start", "id": "foo"}\n
C -> S: {"type": "guess", "id": "foo", "word": "treat"}\n
S -> C: {"type": "retry", "id": "foo", "guesses": [{ "word": "treat", "mark
C -> S: {"type": "guess", "id": "foo", "word": "sweat"}\n
S -> C: {"type": "retry", "id": "foo", "guesses": [{ "word": "treat", "mark
C -> S: {"type": "guess", "id": "foo", "word": "steam"}\n
S -> C: {"type": "bye", "id": "foo", "flag": "sndk83nb5ks&*dk*SKDFHGk"}\n
```

Implementing your program

Below are a few pointers that may be useful while you are implementing your program.

Word list

We will be using this word list for our server; you can assume that any word on this list is a valid guess (including proper nouns), and that your client's guesses can only be the words on this list. You will note that all words at exactly 5 characters long, and all words are lowercase; you can assume these two facts in your code.

Parsing JSON

Most modern programming languages come with libraries that will both parse and encode JSON for you. Your are *strongly encouraged* to use those libraries and *not* implement JSON encoding/decoding yourself.

Performance

To complete the project you are **not** required to worry about performance. Your client's strategy for guessing can be extremely simplistic if you wish: as long as your client runs correctly and you have found both of your flags, your program will receive full credit. Do note that the server will cut your client off after 500 guesses in a single game.

That said, you may find that non-intelligent strategies may take a long time to find your flags, and you are encouraged to think about ways to implement smarter strategies. Particularly creative strategies, as documented in your `README.md` file, may be awarded a small amount of extra credit by the instructors.

to remember all of your guesses). You are welcome to take advantage of this when implementing your client to make things easier.

Encrypted Communication with TLS

In addition to supporting the unencrypted version of the protocol specified above, your client program must also support an encrypted version of the protocol. To accomplish this, you must modify your client such that it supports TLS connections. If the `-s` parameter is given to your program, it should connect to the server using an encrypted TLS socket and then complete the protocol normally (i.e., `hello`, `start`, `guess`, `retry`, and `bye`). You may assume that the server's TLS port is `27994`, unless the port is overridden on the command line using the *-p* option.

All modern programming languages have support for TLS encrypted sockets. You may use libraries, modules, etc. to facilitate adding this functionality to your client program.

When you successfully run your TLS-enabled client against the TLS version of the server (using port `27994`), you will receive a new secret flag (that is different from the normal secret flag). You must add this TLS secret flags into the `secret_flags` file when you turn in your project (i.e., your secret flags file will eventually contain two flags).

Other Considerations

You may test your client code with our server as many times as you like. Your client should conform to the protocol described above, otherwise the server will terminate the connection with an error message. Your client program should verify the validity of messages by checking their format. Remember that network-facing code should be written defensively.

Submitting Your Project

To turn-in your project, you should submit the following four things:

1. Your thoroughly documented source code that implements your client program.
2. A `Makefile` that compiles your code. **You must turn-in a Makefile**, even if your code does not need to be compiled (e.g., your code is in Python or Ruby). You can create a Makefile that does nothing in this case.
3. A plain-text (no Word or PDF) `README.md` file. In this file, you should briefly describe your high-level approach, any challenges you faced, the guessing strategy that your client implements, and an overview of how you tested your code.
4. A file called `secret_flags`. This file should contain both of your *secret flags*, one per line, in plain ASCII.

Your `README.md`, `Makefile`, `secret_flags` file, source code, etc. should all be placed in the root directory of a compressed archive (i.e., a .zip file) and then uploaded to Gradescope. Do not use the archive or compress Finder option on a Mac—Gradescope does not parse these archives. Instead, use the `zip` program on the command line. Alternatively, you can check **all** these items into Github, download a zip file from Github, and submit that to Gradescope. Do not submit code binaries or object files.

Grading

Item	Percentage of Grade
Correct secret flags	25%
Program correctness	60%
Style and documentation	15%

If your program compiles, runs correctly, and you successfully submit both *secret flags*, then you will receive the full points for *Program correctness* and *Correct secret flags*. The *Style and documentation* portion of the grade is an evaluation of how well the strategy is documented in your `README.md`, if the code is readable and well-documented, and the robustness of the submitted code. All student code will be scanned by plagiarism detection software to ensure that students are not copying code from the Internet or each other.

FAQ

Here are a few common questions that get asked about this project:

- *Question: does the [`Makefile` , `README.md` , `secret_flags` , `client`] file need be named exactly that? Can I turn in [README.txt, client.py, secret_flags.whatever, etc.] instead?* **NO.** The files need to be named exactly what we have specified in this document. If you don't follow the specification exactly, you will lose points.
- *Question: Why do we need a `Makefile` ?* This is the consequence of letting you write your program in whatever language you want. Since you can turn-in whatever crazy source code you want, we need to set a couple of ground rules so that we can compile and run your code. Those ground rules are (1) everyone turns in a `Makefile` , and (2) everyone's program must be named `client` .
- *Question: Sometimes when I `socket.read()` , I don't receive a complete message from the server. Why isn't the server sending me complete messages?* You are probably not reading from the socket until you receive a newline (`\n`), which indicates you have read the entire message.

← Schedule

Project 2: FTP Client →