Instructions

Due: 9/18/16 11:59PM

Complete the following assignment in pairs. Submit your work into the Dropbox on D2L into the "Programming Assignment 1" folder. Both partners will submit the same solution and we will only grade one solution for each group.

Objectives

In this lab you will learn:

- To write a client-server application.
- To communicate using HTTP.
- To design a messaging standard.

Assignment

In this lab you will write a distributed implementation of the Battleship game. We will use the standard 10x10 variation of the game. Here is an online implementation of the Battleship game. *Note that the ships in that implementation have slightly different names.*

Our implementation will be based on a symmetric client server architecture, where each player has both a server and a client. The server keeps an internal state of the game and issues replies to the other player's client.

Board Setup

The first step before the game begins is the setup of the board, according to the rules of the game. We will represent the board with a character array, where '_' represents water and Carrier, Battleship, cRuiser, Submarine, and Destroyer fields are represented by '`C`,' '`B`,' '`R`,' '`S`,' '`D`' respectively. For example, your board might be set up as follows:

```
CCCCC_____
BBBB______
RRR_______
SSS_______
D_________
D_________
__________
__________
__________
__________
```

You will save your board as `board.txt`. *Note that the character to represent water is '_' – the underscore.*

Messages

In class we designed a set of messages to be exchanged between the client and the server. The 'fire' message needs to communicate the grid location of a salvo. The 'result' message needs to communicate whether the salvo was a hit, a sink, or a miss. Standardization of these messages will allow you to connect your code with that of other groups.

Based on the class discussion, here is the format of messages we will use. The 'fire' message will be represented as an `HTTP POST` request. The content of the fire message will include the targeted coordinates as a URL formatted string for Web forms, for example 5 and 7, as: `x=5&y=7`. Assume that coordinates are 0-indexed.

The 'result' message will be formatted as an HTTP response. For a correctly formatted 'fire' request your reply will be an `HTTP OK` message with `hit=` followed by 1 (hit), or 0 (miss). If the hit results in a sink, then the response will also include `sink=` followed by a letter code (`C, B, R, S, D`) of the sunk ship. An example of such a reply is `hit=1&sink=D`.

If the fire message includes coordinates that are out of bounds, the response will be `HTTP Not Found`. If the fire message includes coordinates that have been already fired opon, the response will be `HTTP Gone`. Finally, if the fire message is not formatted correctly, the response will be `HTTP Bad Request`. For your reference here's a link to the different HTTP response status codes.

Program Invocation

Your server process should accept a port parameter, on which a client can connect, and the file containing the setup of your board, eg.

`python server.py 5000 board.txt`.

Your client process should accept the IP address, the port of the server process, and the X and Y coordinates onto which to fire, eg.

`python client.py 128.111.52.245 5000 5 7`.

The client will be invoked multiple times during the game.

Internal State Representation

Following each `fire` message the server should update the state of the player's board (whether a player's ship has been hit and where). Following each `result` message the client should update the record of the player's shots onto the opponent's board. A player should be able able to visually inspect their own board and their record of opponent's board on `http://localhost:5000/own_board.html` and `http://localhost:5000/opponent_board.html` respectively. It is up to you how you visually represent the state of each board, however, I will award **one bonus point** to the group with the most visually appealing representation.

BONUS

I will also award **one bonus point** to any group that implements the Version 1 rules of the Battleship: Advanced Missions variant of the game.

I will also award **one bonus point** to any group that implements the client and server FlatBuffers, or Protocol Buffers.

What to Submit

1. [2 points] Find a partner. Submit `partners.txt` with your partner's, or partners' first and last name.

2. [3 points] `message_format.txt` – A text file describing the message formats you are using in your implementation. Let us know if you are using FlatBuffers, or Protocol Buffers for your fire/result messages.

3. [10 points] `server.py` – your working Python implementation of your server process. `client.py` – your working Python implementation of your client process. Alternatively you may also implement this assignment in C/C++. Note: code that does not compile, or crashes will receive zero credit.

4. [10 points] A link to a one to two minute YouTube video showing a narrated execution of your code.

5. [1 point] (BONUS) `client_am.py` and `server_am.py` – implementing the Advanced Missions rules of the Battleship game. Please also include `BONUS_README.txt` that explains any changes into how the client program should be invoked.