Programming Project 3 - 20% - Part A

Due Date – 18.1.15 23:55

Online Battleship Game

(Bad illustration: children don't actually enjoy the game)

Assignment Overview

Part A: You are going to build a game that allows 2 players (*humans only*) to play an Online Multiplayer game of Battleship (צוללות). This is quite a challenging task, as you'll need to communicate between server side and client side (see next), but you can relax as all the technical network related issues are given to you, including server side and client side which connect to each other. At this stage you probably look like:

Part B *(Bonus)*: You'll write a short program which solves one of Google riddles for programmers, during which you'll implement a simple, yet efficient primality testing algorithm (Miller-Rabin). (This part was uploaded separately to the course website). **This is worth 5 points.**

Grade: Part A is 100% of the grade. The second part is very short, and it is offered to you so you can compensate any point loss in the first part. The maximal grade still remains **100.**

Battleship:

Background

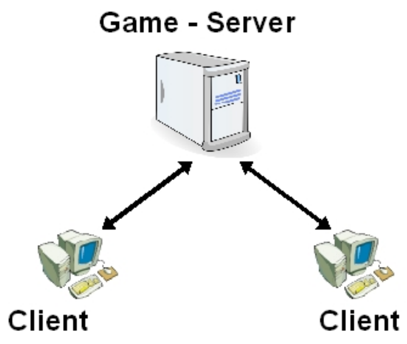
It is the simplest battleship game you've all played when you grew up. If you're not familiar with the game rules, look at: http://en.wikipedia.org/wiki/Battleship_%28game%29.

Notes:

1. We will support 2 players game only.
2. The actual rules of the game say that if a player hits his/hers opponent's ship, he/she gets another turn. For simplicity reasons – we **won't** play like this. Instead, we switch turn after each move a player makes.

Online Multiplayer:

In order to play the game online, we will need to build a server – this is a program that waits for players to connect, and manages the game (i.e. sends each player with his opponent's moves).
In addition, we'll need to create a client – this is another program (which executes separately from the server), that handles a player currently playing the game.
Each player connects **only** to the server (and **not** to each other).
The idea is to have something like this:



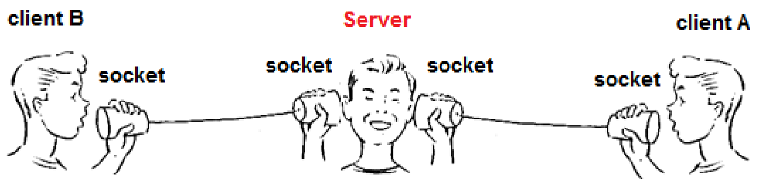
(Bad illustration, client side isn't this small)

As this can be quite a difficult task, we will have to make a lot of simplifying assumptions. In addition, some of you might not be familiar with the concepts of server-client model – this is why everything that is pure network related **is given to you**, so you can focus on the gameplay and Python, without having to worry about any of these. You'd still need to understand few basic concepts which we'll explain next, so you'd know how to use what we gave you. We encourage you to gain better understanding of these concept as there's a good chance that you'll encounter them again sometime in your career. The idea of this exercise is to show you that opening a very simple server and client sides is not an impossible task, and although most of this logic is given to you, this are good practices to know.

Server-Client:

On Unix (and Windows and Mac) based systems, 2 machines can communicate with each other using something called 'socket'. A 'socket' is just a tool one machine uses in order to send information to another machine on platforms like the Internet (or locally as well). For this to work, both sides (server and client) need to hold a socket, and the 2 sockets should be connected to each other.

You can think of a socket as a cone a client speaks to, and through the magic of the Internet, the server can hear his message. This works vice versa as well, the server can speak to the same cone, and the client can hear the message.
To sum up, the server listens for requests from clients, and sends responses back. In addition, not only do we want to listen to connections, but we also want to 'listen' to user input (the keyboard). **This was also done for you.**
Note: A pair of a server and a client needs 2 sockets to communicate. This means that if 2 clients are present (like in our case), we'll need total of 4 sockets, see illustration below.



(Bad illustration: A server isn't actually a man holding 2 plastic cups. Clients are, though)

Supplied files:

We supply you with 3 files: **Server.py, Client.py and Protocol.py** which handle all the network related issues for you, connect clients, greet them with welcome message and print the boards.

Server.py – contains all the logic (and code) that sets up a server that listens for new connections and also responses to keyboard inputs (You'll see next that a server should only handle the input 'exit'). This program takes **2 argumets:**

- **machine_name –** the machine name on which the server is running.
- **port_number** –

You can get your machine name using **socket.gethostname()** or use "**127.0.0.1**" to run the server locally. The port is just a number for that matter; you don't have to fully understand what it is used for (when you test your program use numbers anywhere between 1024 to 65535).
Client.py - contains all the logic for a client to connect to the server described above, and also supports user input. This program takes 4 arguments:

- **machine_name** - the machine name is the machine on which **the server** is running
- **port_number –** the same you used when opening a server
- **player_name –** the name of the player
- ***.ship file path –** player ships (**see Ship File** section), currently not in use, but you'll need this when you implement the game logic.

Protocol.py – contains 2 methods: **send_all**, and **recv_all**. These are the methods you should use in order to communicate between the clients and the server. These methods handle the case of partially sent messages or exceptions that may be raised. These methods were documented thoroughly so you'd know how to use them. **Every time** you use the methods 'send_all' and 'recv_all' you **have** to check for errors (as I/O calls tend to fail for no reason). The right error will be returned to the caller, and he should handle them properly (see **Error Handling** section).

If done properly, apart from playing against each other on the same machine, you should be able to compete from 2 different computers.

*** You can edit these files as much as you like (except for specific parts which are documented as "DO NOT CHANGE").**

Example how to use these files:

On one terminal, go to where **Server.py** is located and type: **python Server.py 127.0.0.1 8888**. You've now opened a server (locally).
On a **different** terminal, go to where **Client.py** is located and type: **python Client.py 127.0.0.1 8888 Iosi Ships/aa.ship**. This will open a client named Iosi and will connect him to the server you've just opened.
On yet another different terminal, and type: **python Client.py 127.0.0.1 8888 Eti Ships/bb.ship.**
You've now finished connecting the 2 clients to the server – everything you'll write in one client's terminal will be sent to the server which will send it to the other client connected.
You can run a server on one computer and connect each client from a different computer, just make sure to replace "127.0.0.1" with the machine-name the server is running on.

Game Logic:

Now we actually move to describe how to implement the game: Let's describe in general lines what the logic is:

1. Open a server
2. Open client A and wait for other players to join
3. Open client B
4. Server informs players: Game starts

5. While no player has won or quit: - this is where you come in
    6. Player $i \in \{0,1\}$ makes a move and sends it to the server
        6.1 Server 'parses' the move and informs the player with the result of his move
        6.2 Server sends **the other** player the move just played by his opponent
        6.3 Both players (clients) update their board accordingly
        6.4 We switch turns: $turn = 1 - i$

7. Winning player is informed he has won.
8. Losing player is informed he has lost.
9. Close client A.
10. Close client B.
11. Shut-down server.

Prints format:

Below you'll find details about each part of the game. Since we leave the design mostly up to you we can't really give you all the prints as you'll use them. The prints should be exactly as ours, but it is up to you how to create these strings (generate in server or in client etc.). We uploaded a set of output examples from which you can learn exactly what should be typed in each stage in case it won't be clear from this description.
That being said, within **Client.py** you'll find a method named '**print_board**()', this is the method you should use when you print the player (and his opponent) boards. You'll need to figure out how to use this method and how to add minor changes to it so it will fit our purpose – but the formats of the boards are given to you.

Ship file:

Before the game can actually start, each player must locate his/her ships on his/her board. To avoid the excruciating pain of placing each ship, we'll parse a simple file called <something>.ship (this is one of the parameters a client receives). This file consists of **X lines**, each describing **exactly one ship**. For instance, consider a file containing the following lines:
A1,A2,A3
C5
E10,E9,D9

This means this player has a ship of size 3 in cells A1 to A3, a ship of size 1 in cell C5, and a ship of size 3 in cells E10E9D9.

Notes:

- You can assume legality of the file. It will always include at least one ship. No 2 ships will overlap or will be in diagonal orientation, and there will always be at least one cell between 2 ships (just like the original game)
- No comments nor empty lines nor white spaces will be present. We've already dealt with that in the previous exercise, and there is no need to do this again.
- Note that a ship isn't necessarily a straight line (like E10E9D9) – at first glance, this seems irrelevant, but this will make additional effort once a ship is sunk (see next).
- There could be any combination of different sized ships in this file.

Start Game:

The game will start only when 2 clients are connected to the server. On this point, each client is presented with a welcome message along with the name of the player he plays against. Next, each player prints the boards – on this stage each player should see his board in front of him, as well as a yet-to-be-revealed opponent's board (just like the actual game). **The starting player is always the client who connects first** – all this is done for you.

Move:

At his turn, a player will make a move. A move is defined as of a letter followed by a number, separated by a single whitespace. For instance: **A 1**. The move a player makes is the position he wants to attack on his opponent's board. Once a player makes a move, he sends it to the server. Next you should determine the result of this move: **miss, hit** or **sink.** After that, the server needs to inform the other player that his opponent has just attacked in **<letter> <number>** . Now that both players are aware of the attack and its result (you decide exactly what will be sent from the server back to the clients), they should each update their boards accordingly. Let's say that this was a miss, then the attacking player should add '**X**' to his opponent's board, and the defending player should also add '**X**' but to **his** board. (It's best if you now look at the examples we gave you to fully understand our meaning).

a. When a move results in a miss –you should only mark with '**X**'.
b. When a move results in a hit –again, you should only mark with '**H**'.
c. If a player was able to sink his opponent's ship, this calls for a special handle. In this case, not only that we mark the entire ship with hits ('H's), but we also need to mark the ship's surrounding with '**X**'s. This can be quite challenging when you don't know the structure of the sinking ship in advance. You might consider implementing a simple BFS algorithm here, but

you don't have to. Of course this should be done in both the attacking player side, and the defending player side.

Notes:

- You can assume a move will always follow the given format and will always be legal – i.e. within the boundaries of the board.
- You can assume that each player would only play on his turn (except for one case – see Quit section).

Win/Lose:

A player wins when he manages to sink **all** his opponent ships. A player loses when **all** his ships were sunk. At which point, the winning side should be declared as victorious (this is simply a print), and the losing side should be declared as the loser (again, a simple print).

Notes:

- If a player's opponent quits the game (see Quit Game section), this players is granted with a technical win.

Quit Game:

At each stage of the game the server can be shut-down. This is done by typing '**exit**' on the server terminal. Before a server can shut-down, it needs to close all connections (sockets) it currently holds, print some informative message (see example) telling that the server is down, and exit (end the program, using **exit(0)**) .

In addition, at **each stage** of the game either one of the players (**whether it's his turn or not**) can leave the game by typing '**exit**'. What happens next is this client closes the connection to the server, prints something (see example) and exits (using **exit(0))**. At this stage the server detects this client has disconnected, and so it informs **the other** client his opponent has quit and thus he gets a technical win. The other client will then close its connection with the server and exit. Finally, the server will shut-down and exit as well.

Error Handling:

We continue our one-time tradition of assuming legal input wherever, whenever possible. **However**, unlike the previous exercise, this exercise involves using a lot of system calls (such as when we open a server and a client, when we form connections when we send message), which can fail for no reason at all as mentioned before. The code we gave you **already handles all the cases a system call can fail**, and since we wrapped the message sending mechanism in the **Protocol.py** file, all that's left for you is to check the returning value every time you use **Protocol.send_all()** and **Protocol.recv_all()**. (And as you'll learn to realize, you'll use these methods constantly).

The rule is as follows: If and when a call to send_all() or recv_all() fails (you'd know by the returning value), you should close the calling process, only after you properly closed all the sockets.

If an error occurs, the server / client should print the error message (you get this from Protocol) to the **stderr** (using **sys.stderr.write(msg)**).

Note: If a client / server disconnected, and thus we need to informs the remove ends to this socket - this is <u>**not**</u> considered as an error.

Important Note: You can assume that all calls would be done in a **synchronous matter**. This means that you can assume that once a client sends a message to the server, the server won't accidently miss it because "it wasn't listening at the time", and No 2 clients would send message at the same time.

If the next passage means nothing to you – this is ok, since you don't have to worry about any of these in this exercise. It is given to you as extra information that might help you.
As some of you might not know the difference between Blocking and Non-Blocking I/O operations, we'll try to give a short explanation. If this issue is still not clear – don't be afraid to ask us via Moodle or Email or personally.
Very simple definition: An I/O call (like reading from a file) is considered **Blocking** if it stops the execution of the process until the I/O call returns (or fails). For instance, let's say we try to read a very large file – if this is a blocking operation, then our program won't continue its execution until this operation is done. However, if this is non-blocking operations, then we send a request to read from a file, meanwhile our program will continue running, and sometime in the future, and only after the reading is done, our program will return to someplace with the file data in hand. (A popular mechanism to achieve this is using threads or via callbacks.

Tips and additional notes:

a. You might consider building a Player() class which handles all the logic for the actual game, leaving network related issues to the Client(). You can build any other classes if you like (perhaps a Board()?)
b. We deliberately don't force you to send messages in a specific way, nor do we force you to follow a specific protocol. It is up to you to decide how and what is the content of the messages you send. For instance – let's say a player has connected; then apart from his name, you should decide if the server (and maybe the other client) should get additional information from this client. We don't limit you in any way. On the one hand, this could be easier since you're free to do whatever you like, on the other hand you might need to take some time to carefully design your implementation before you actually start coding.
c. The files we gave you should only work on school computers. This is because the **select.select()** system call we use does not support listening to both standard input and sockets simultaneously on Windows OS. If you wish to work from home, you can temporarily disable listening to sys.stdin (see in our code where this is done), but be sure to test your solution on school computers.
d. Start by writing the game logic without any network support. This will also enable you to work from Mac or Windows. Once you have that done, start adding the network support. Remember that most of this was done for you, and you should only utilize the methods we gave you.

Submission
Submit **a tar archive** (not just the .py file!) named proj03.tar containing **Server.py, Client.py** and **Protocol.py** (and any other files you used).

Late Submission:
5 points will be deducted for **each** day submitted late. After the 5th day, the exercise will be graded 0.
No extensions will be given unless in case of miluim or illness.

Finally:
The number of lines of code you'll need to write in order to implement this exercise is smaller than in previous exercise. However, this exercise in harder conceptually – **so** **start** **early**, and **don't** postpone it for the last week.
Start by examining the code we gave you, it will make things a lot clearer. Hopefully by this stage, you look something like:

Good Luck!