REDES DE COMPUTADORES
LAB – SOCKETS

This lab will be a bit different from the other labs we have undertaken so far. Here, we will not use Wireshark or analyze well-known protocols. Instead, our focus will be on writing some a (yet useful) network application using sockets. More specifically, you will be tasked with creating a simple protocol for an implementation of a network-based multiplayer version of the classical game *Battleship*. After that, you will use the *sockets* library for writing the specific functions that concern the network communication withing this implementation. As one final step, you will test your implementation by running the game on two different computers and playing a match.

1. THE GAME

Battleship is a very simple game. It can be played with just pen and paper, but here we will consider a computer version.

The game is played by two players. Each player has a board which is organized in a grid with a certain number of columns and rows. For this lab, we'll assume that the board has 10 columns (numbered from 0 to 9) and 10 rows (also numbered from 0 to 9). At the beginning of the game, each player will place its ships on its board, and the goal here is to achieve a placement that is as unpredictable as possible for the opponent (*i.e.*, the opponent should have difficult in guessing where are your ships). Different versions of the game work with different sets of ships, but for this lab we'll consider the following five ships:

- *Carrier*, of size 5.
- *Battleship*, of size 4.
- *Destroyer*, of size 3.
- *Submarine*, of size 3.
- *Patrol Boat*, of size 2.

Notice that each ship has a certain size, which corresponds to the number of consecutive grid positions that the ship will occupy on the board. Ships can be placed either horizontally or vertically.

After both players have positioned their ships, the game proceeds in rounds. At each round, players take turns attacking each other. An attack corresponds to a shot: the attacking player announces to the opponent the coordinates (row and column) of the shot it is currently firing; the opponent then checks if it has (a part of) a ship at those coordinates; if so, the shot is a hit (*i.e.*, it destroys that part of the ship); otherwise, it is a miss. A ship is only sunk if all the squares it occupies are hit. The game ends when all the ships of a given player have been sunk: that player is the loser, while the opponent is the winner.

2. THE IMPLEMENTATION

Because our focus on this lab is just the network communication portion, you'll not have to implement the full game. Instead, you'll find attached to this document an almost complete implementation. This implementation is divided into four modules/files:

- ***Battleship***: this contains the main program and accessory functions. It implements the logic of the game, as well as the basic user interaction.
- ***Board***: this module represents and manipulates the players' board.
- ***Term***: this module has a few helper functions that allow the program to do some special manipulations of the console (*e.g.*, print in color).
- ***Sockets***: this module handles the communication aspects of the implementation.

The only incomplete module is the ***Sockets*****:** it only contains the stubs of the functions / methods that should be implemented by you. All other modules are complete and, thus, require no modification from your part.

This implementation consists of a client-server application. One of the players should run the program as a server (indicated by the command-line option *-s*), while the other must run it in client mode (option *-c <server_address>*, where *<server_address>* should be replaced by the address of the desired server).

You should implement the following functions / methods:

- *create_server()*: this function initializes whatever you consider that must be initialized so that the server side application can receive a client.
- *wait_client()*: on the server side application, this function is called to wait for a client and establish a communication.
- *connect_server(serverAddr)*: on the client side application, this function is called to contact the server indicated by *serverAddr* and establish a communication.
- *send_ready()*: this function sends a READY message which indicates to the other side of the communication that the local player has already placed its ships on its boards.
- *wait_ready()*: this function waits for a READY message sent by the other side of the communication.
- *send_shot(x, y)*: this function sends a SHOT message, announces to the other side of the communication the attack performed by the local player (a shot to the cell at row *y* and column *x*).
- *wait_shot()*: this function waits for a SHOT message sent by the other side of the communication and returns the coordinates of the shot.
- *send_gameover(status)*: this function sends a GAMEOVER message, which informs the other side of the communication if all the ships of the local player have been sunk. Notice that this message carries a Boolean *status* that defines whether or not the game is over.

- *wait_gameover()*: this function waits for a GAMEOVER message sent by the other side of the communication and returns the status received within that message.
- *send_result(res)*: this function sends a RESULT message, which informs the other side of the communication if the previous shot was a hit (char 'H') or miss (char 'M').
- *wait_result()*: this function waits for a RESULT message sent by the other side of the communication and returns the result received on that message (char 'H' for hit or char 'M' for miss).

While you *have* to implement all those functions for the game to work properly, you *can* add any helper function that find useful (suggestion: create generic functions for sending and receiving messages, which will then be called by the *send_** and *wait_** functions above).

3. The Communication protocol

Before you start actually coding, you should first define the application protocol that your implementation will use. Notice that part of that protocol has already been defined for you, as the provided partial implementation already defines the types of messages and their semantics. However, notice that the specific format of each message is not defined. It is up to you to define those details (e.g., message format, if the protocol is binary or text-based). Another decision that you should make is whether to use TCP or UDP as a transport layer protocol.

4. Testing the Implementation

Once you have finished implementing the ***Sockets*** module, the implementation should be fully operational. To test it, you should run both server and client sides and attempt a match. While you can run both sides on a same computer, it is more interesting to test it on different computers.

To do so, connect both computers to a same network and find out the IP address of the one where the server will be executed. On windows, that can be done by running the command ***ipconfig /all*** on a console, which will show the configurations of the various network interfaces on the computer. On Linux, you can use the command ***ifconfig***.