

Battleship

A TCP Socket Implementation

Felipe M. Megale¹, Isabelle H. A. Langkammer²

¹Pontifícia Universidade Católica de Minas Gerais (PUCMINAS)
30535-901 – Belo Horizonte – MG – Brazil

²Instituto de Ciências Exatas e Informática

`{fmegale, isabelle.langkammer}@sga.pucminas.br`

Abstract. *This paper aims to describe an implementation of the Battleship game, using the Python programming language to code two programs that exchange messages between each other using the TCP protocol [Cerf and Kahn 1974] in the client / server model.*

Resumo. *Este artigo tem como objetivo descrever uma implementação do jogo Batalha Naval, Battleship, em inglês, utilizando a linguagem de programação Python para codificar dois programas que trocam mensagens entre si utilizando o protocolo TCP [Cerf and Kahn 1974] no modelo cliente / servidor.*

1. Summary

A Battleship game is made of two players, each with it's own 10x10 board, and a few ships. The game consists of each player placing their ships on the desired location. Afterwards, each player then chooses a position on the board, denoted by a letter of the alphabet from A to J, which represents the rows of the board, and a number from 1 to 10, which represents the board's columns to attempt an attack on the opponent's ships. Should the attacker hit, that is, guess a position on the opponent's board that is occupied by a ship, the attacked player must state so, and vice versa.

The problem involved in this simple game is the game needs to be played by two people on two separate computers, and one of the players is a machine (the server). There must be a connection placed between the server and the program the human player is using, and that is accomplished with socket programming.

2. Implementation

2.1. server.py

The first piece of code we are going to explain is the server side of the application. The server is responsible for automatic, unique, random attacks. We have defined the connection type, by instantiating a socket, which accepts IPv4 addresses, and uses the TCP protocol to communicate with the outer world. Since all connections need to go through a port, our socket has been bound to a hardcoded port, 1234. Finally, due to software requirements, we have also forced the server to accept one connection at a time, using the `listen()` command.

When a client connects to the server, a while loop is triggered and the first action the server takes is generate a new, random board for itself, by instantiating a new object of type Board, which we are going to talk about in a later section of this paper. After the board is generated, the server waits for the client to make the first move. This was a decision made on how the game would work. It seemed reasonable that the client should make the first move.

When the server receives the first message from the client, it verifies whether the client has hit a ship, generates a random attack, and sends the response plus the attack position back to the client, in a CSV fashion. On subsequent server attacks, a similar logic is applied, except for when the server hits a client's ship. In this case, the server must attack the same row as the last time, but a column to the right of the last attack. However, the server can't attack to the right forever, since the client's board has a 10 column width. Then, the position the server attacks has to be verified. If it can attack to the right, so it does. If not, i.e. the server aimed at the tenth column, it must recalculate a new random position of the board to attack.

2.2. client.py

The client code starts off by setting up the connection similarly to what is done with the server. The same IP family is defined, as well as the transport protocol. Then, the client has to place its ships on the board. This is done by reading a text file located on the same folder as the source code for the client. With the board assembled, the client then connects to the server.

When the connection has been established, the CLI presented to the client will contain a simple help guide, stating what the client can type in in order to obtain some information. He can either ask to see the help message again, or ask to see what he's learned from the server so far, as well as see the current status of his own board, i.e. where the server has attacked.

On the first attack, the client sends only the position of the attack (e.g. A,2). The attack must be written as showed here, due to the logic used in both server and client sides message processing. The client then receives a message from the server stating whether he's hit a server's ship, alongside with the server's current attack, in a CSV fashion. The client is free to invoke the help message, or the board status at any time.

2.3. Board.py

This class is responsible for the random assembly of the server's board. It places the carrier horizontally, the tankers vertically, destroyers horizontally, and submarines vertically. The first ship placed is the carrier, since it's the largest one on the fleet. Subsequent ships are placed randomly by first choosing a starting point. If this position on the board is empty, then the algorithm checks whether it can place the ship up or down, if its kind requires vertical orientation, or if it can place the ship to the left or to the right, if its kind requires horizontal orientation. This class also has a function, which is used in the client side, which is a print function to see the server's board.

2.4. Data Structures

Boards are 10x10 2D arrays, which, in Python, can be read as a list of lists. Ships are placed on the board by simply accessing the position on the 2D array and setting a new

value for that position. Regarding the client's view of the server board, the client marks where they've attacked with an X. All the other positions remain in an unknown status, depicted by a question mark.

Each file has the functions and script lines that apply only for that specific file. Server and client connections with their respective lines. The server side has a list that stores its previous attacks, preventing it to try to attack the same position twice or more.

On the other hand, the client code has a class named client, which is responsible for reading the text file containing the input ship positions, as well as implementing a print function to print the client's board. Since the board is referenced with both letters and numbers, it seemed reasonable to abstract the mathematics and conversions from the client. Hence the parts of the code subtracting 65 and 1 from a read value. Also, the client has a 2D array representing the server's board, initialized with question marks, as well as a string called hit_or_miss, which records the status of the server's last attack.

It is also worth noting the game ends when a player has sunk all the other player's ships. This is the verification carried out by the while loops in both client and server codes. Each player starts with 30 positions marked on the board. As the game goes and someone hits a position that contained a ship, the ship counter decrements.

3. Tests

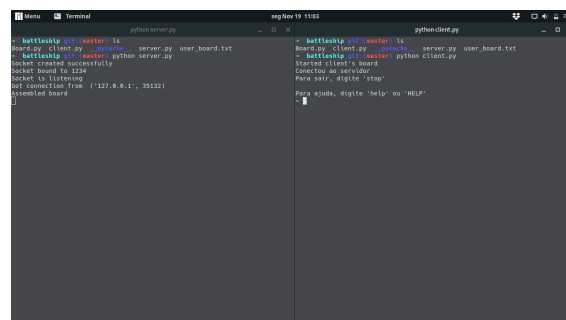
In this section, some tests conducted are going to be shown and illustrated by print screens of the two terminals used to run the application.

When the server is up and a client connects, the server logs that a client connected, and that its board has been assembled.

When the client asks the program for help, the information in Figure 2 is displayed.

Another option presented to the client is to print its board and the server's board, by using the p command, as depicted in Figure 3.

Figure 4 shows a few of the client's attack, alongside the server's response, and the current status of the boards.



```
python server.py
Battleship v1.0 (master) is
Board.py client.py ...python... server.py user board.txt
Battleship v1.0 (master) python server.py
Socket created successfully
Socket bound to 1234
Socket is listening
Got connection from
assembled board

python client.py
Battleship v1.0 (master) is
Board.py client.py ...python... server.py user board.txt
Battleship v1.0 (master) python client.py
Started client's board
Construct an server
Para sair, digite 'stop'
Para ajuda, digite 'help' ou 'HELP'
```

Figure 1. Connection result

4. Conclusion

The software described in this paper used the TCP protocol and the Python programming language to implement a simple Battleship game. The expected delay due to the overhead

```
pythonclient.py
battleship g1:(master) ls
board.py client.py _pycache_ server.py user_board.txt
battleship g1:(master) python client.py
Started client's board
Conectou ao servidor
Para sair, digite 'stop'

Para ajuda, digite 'help' ou 'HELP'
= help
Entre 'p' para ver o que aprendeu do servidor
Entre 'stop' para sair
Para atacar, informe <linha><coluna>, E.g. A,1
p
```

Figure 2. Help displayed

```
pythonclient.py
battleship g1:(master) ls
board.py client.py _pycache_ server.py user_board.txt
battleship g1:(master) python client.py
Started client's board
Conectou ao servidor
Para sair, digite 'stop'

Para ajuda, digite 'help' ou 'HELP'
= help
Entre 'p' para ver o que aprendeu do servidor
Entre 'stop' para sair
Para atacar, informe <linha><coluna>, E.g. A,1
p
My Board:
X S S e e T T T
X e e e e e e e
X e D e e C e S
X e D e e C e e e
X e D e e C e D e e
X e e e C e D e e
X S e e C e D e e
X e e e S S e D
X T T T e e e D
X e e e e e e D

Server's Board:
? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ?
```

Figure 3. Print command used

in the TCP protocol could not be measured, as the exchanged messages were of small size.

References

Cerf, V. and Kahn, R. (1974). A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):637–648.

```
python client.py
- A,1
You miss! I attack on position (D,2)
- A,2
You miss! I attack on position (C,9)
- A,3
You miss! I attack on position (C,10)
- A,4
You miss! I attack on position (F,9)
- A,5
You miss! I attack on position (H,5)
- D,7
You miss! I attack on position (I,7)
- G,9
You miss! I attack on position (F,1)
- B
My Board:
e o s e e e T T T I
e e e e e e e e e
e o D e e C e e X X
e o D e e C e e e e
e o D e e C e D o e
e e e e C e D o e
e s s e e C e D o e
e e e X e S s e D
e T T T e X e e D
e e e e e e e D

Server's Board:
m m m m ? ? ? ? ?
? ? ? ? ? m ? m ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
? ? ? ? ? ? ? ? ?
```

Figure 4. Attacks and print displayed