

**UNIVERSIDADE ESTADUAL DE MATO GROSSO DO SUL**  
**CÂMPUS DOURADOS**  
**CIÊNCIA DA COMPUTAÇÃO**



**LUIS EDUARDO SOUZA VASCONCELOS**

**RELATÓRIO TRABALHO 1**  
**(Jogo de Batalha Naval em linguagem C usando Sockets)**

**DOURADOS – MS**  
**2021**

## 1. SUMARIO DO PROBLEMA A SER TRATATO

Um dos principais problemas encontrados na implementação deste trabalho, foi o entendimento de como funciona a API sockets e também a sincronização de suas operações com os demais clientes. Desde que a implementação de servidor com múltiplos clientes foi utilizado a recurso das threads, também foi preciso entender e resolver os problemas das "Corridas", para isso foi utilizado o recurso de mutex, que foi essencial para o funcionamento do programa.

Para a realização deste trabalho, foi preciso pesquisar a fundo o funcionamento dos sockets e também buscar muitas bases para conseguir utilizar as threads e fazer elas funcionarem de forma correta.

## 2. DESCRIÇÕES DOS ALGORITMOS, TIPOS ABSTRATOS DE DADOS, PRINCIPAIS FUNÇÕES E DECISÕES DE IMPLEMENTAÇÃO

Para a implementação de um servidor que aceita vários clientes foi utilizado threads. Optei por fazer desse modo, pois fica mais fácil trabalhar quando cada cliente possui a sua própria thread, ou seja, paralelismo. E também a facilidade de debugar é notável, por mais que não seja muito elegante. Com cada cliente tendo a sua própria thread rodando, a comunicação com o servidor se torna única para cada um, desse modo é muito simples a troca de mensagem. Em outras palavras, é como se cada cliente tivesse o seu próprio servidor, então fica fácil para o servidor saber para qual cliente ele está mandando a mensagem.

### 2.1 Servidor

A principal estrutura do servidor que guarda os dados de cada cliente é:

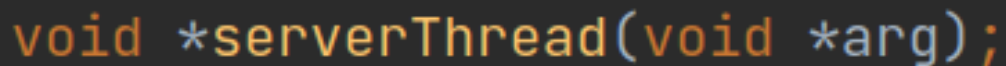
```
/*Estrutura que guarda os dados dos clientes*/
typedef struct {
    struct sockaddr_in address;
    int socket;
    int ID;
    int port;
    int inGame;
    char name[50];
} Clients;
```

**Imagem 1** - Estrutura que guarda os dados dos clientes.

Nessa estrutura é guardado os principais dados desse cliente, o nome de cada dado é autoexplicativo, mas vale notar que cada cliente terá a sua própria porta única, que

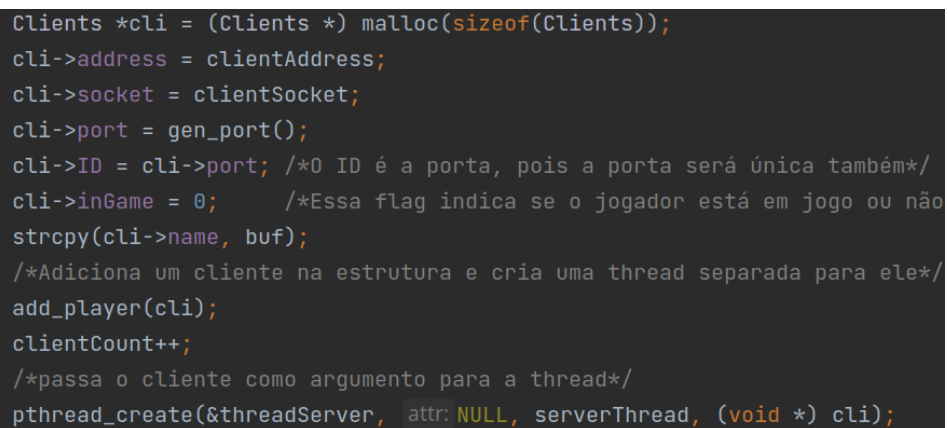
posteriormente será notado que a porta também é o ID de cada cliente. A geração dessa porta é feita pela função `int gen_port()` que gera um número pseudoaleatório entre 10.000 e 11.999, além de verificar se nenhum outro cliente possui essa porta, caso possua, é gerado outro número. A explicação do motivo de cada cliente ter sua porta será abordada mais adiante.

Quando é aceita uma conexão no servidor, imediatamente ele aloca uma estrutura cliente atribui todos seus valores e cria uma thread separada para esse cliente, onde será feita as trocas de mensagens.



```
void *serverThread(void *arg);
```

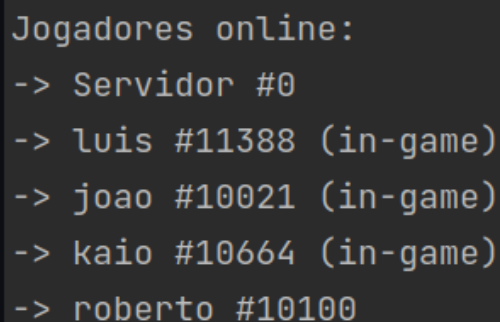
**Imagem 2** - Protótipo do thread que faz a comunicação cliente/servidor.



```
Clients *cli = (Clients *) malloc(sizeof(Clients));
cli->address = clientAddress;
cli->socket = clientSocket;
cli->port = gen_port();
cli->ID = cli->port; /*0 ID é a porta, pois a porta será única também*/
cli->inGame = 0; /*Essa flag indica se o jogador está em jogo ou não*/
strcpy(cli->name, buf);
/*Adiciona um cliente na estrutura e cria uma thread separada para ele*/
add_player(cli);
clientCount++;
/*passa o cliente como argumento para a thread*/
pthread_create(&threadServer, attr: NULL, serverThread, (void *) cli);
```

**Imagem 3** - Alocando novo cliente para a estrutura de dados.

Após criar essa thread, o servidor envia imediatamente para o cliente o valor de sua porta. Nessa thread o cliente terá duas opções de comando que fará a comunicação direto com o servidor, que são: listar, desafiar {player id}. Quando enviado "listar" o servidor enviará uma lista com os jogadores online no servidor, como mostrado abaixo:



```
Jogadores online:
-> Servidor #0
-> luis #11388 (in-game)
-> joao #10021 (in-game)
-> kaio #10664 (in-game)
-> roberto #10100
```

**Imagem 4** - Lista de jogadores após o comando "listar".

Caso o cliente inicie uma batalha contra o servidor, na própria thread do cliente, o servidor chamará a função `void playBattleShip(Clients *cli, char *buffer, int playerID);`

```
/**
 * Lógica do jogo batalha naval, utilizado aqui caso o cliente deseje jogar contra o servidor
 * @param cli cliente que deseja jogar contra o servidor
 * @param buffer buffer para a troca de mensagens
 * @param playerID id do jogador que no caso determina quem será o primeiro a atacar,
 * no caso o servidor sempre será o segundo
 */
void playBattleShip(Clients *cli, char *buffer, int playerID);
```

**Imagem 5** - Protótipo do procedimento onde irá ocorrer a lógica do batalha naval (lado servidor).

Nessa função está definida toda a lógica do jogo Batalha Naval e também será onde vai ocorrer a maior troca de mensagens entre o cliente e o servidor.

## 2.2 Cliente

O programa cliente é mais complexo do que o programa servidor. Ele possui três partes. A parte que iniciará uma conexão, outra parte receberá uma conexão e também a parte em que manterá a conexão com o servidor. Todas serão explicadas resumidamente a seguir.

### 2.2.1 Mantém conexão com o servidor

Para manter a conexão com o servidor, é criado uma thread separada que vai tratar dos comandos que o cliente pode realizar.

```
/*Thread que trata da comunicação com o servidor. Comandos por exemplo*/
void *server_client_thread();
```

**Imagem 6** - Thread que faz a comunicação direta com o servidor.

Não é passado nenhum argumento para ela, pois a maior parte das variáveis no programa cliente, são variáveis globais.

Nessa thread o cliente terá quatro opções de comandos: listar, desafiar, limpar e sair. Sendo que só listar e desafiar que trocará dados com o servidor. Caso ele desafie outro jogador, será enviado para o servidor o nome e o ID do jogador, que verificará se o jogador existe e retornará o endereço e a porta para quem requisitou o desafio.

### 2.2.2 Iniciar uma conexão com outro cliente.

Após enviar o comando desafio para o servidor, a thread `void *server_client_thread();` é cancelada, para iniciar outra thread:

```
/*Thread usada quando desafio alguém,  
*nela é criado outro socket que abre  
*a comunicação com o oponente*/  
void *enemy_thread_handle(void *arg);
```

**Imagem 7** - Protótipo da thread que abre uma comunicação com outro cliente.

Aqui será explicado o porquê de cada cliente ter sua própria porta. Acontece que quando um cliente tentava iniciar a conexão com outro cliente, o endereço e a porta eram as mesmas, pois os testes são feitos na mesma máquina, e por se tratar de thread, ocorria que quando enviava o desafio para outro jogador, o desafio às vezes não chegava no outro jogador e sim para o mesmo que fez o desafio. E deixar uma porta diferente para cada cliente resolveu esse problema.

Após receber o endereço e a porta do jogador que deseja desafiar, é iniciado outra conexão TCP com esse jogador, que, caso aceitar o desafio, é iniciado a Batalha Naval pela função `void playBattleShip(int socket, char *buffer, int playerID)` que será detalhada mais a seguir.

Então, ao fim da função `playBattleShip()` é recriado a `void *server_client_thread()` que faz a comunicação com o servidor e é destruída a thread `void *enemy_thread_handle(void *arg)`.

### 2.2.3 Recebe uma conexão de um cliente.

Logo após ser feita a thread que trata a comunicação direta com servidor, o programa cliente cria uma parte "servidor", é feita toda a estrutura de um servidor para que o cliente esteja pronto para receber uma conexão. Quando receber uma conexão, isso quer dizer que o jogador está recebendo um desafio de outro jogador. Caso aceite o desafio, ambos os jogadores irão para o procedimento `playBattleShip()`.

```

/**AQUI É A PARTE QUANDO O CLIENTE RECEBE UM DESAFIO DE OUTRO CLIENTE**/
/* Quando é convidado para um desafio*/
/*Lado servidor do cliente*/
struct sockaddr_in challengerAddr; //Endereço do oponente que enviou o desafio
struct sockaddr_in clientServ; //Endereço de quem receber o desafio

socklen_t socksize = sizeof(challengerAddr);
memset(&clientServ, 0, sizeof(clientServ));

clientServ.sin_family = AF_INET;
clientServ.sin_addr.s_addr = INADDR_ANY;
clientServ.sin_port = htons((u_short) myPort);

if ((serverSideSocket = socket(AF_INET, type: SOCK_STREAM, protocol->p_proto)) < 0) {
    perror( s: "\n[!]ERRO: Erro ao criar socket para comunicacao com desafiante\n");
    closeSockets();
    exit( status: 1);
}

if (bind(serverSideSocket, (struct sockaddr *) &clientServ, sizeof(clientServ)) < 0) {
    perror( s: "\n[!]ERRO: Falha ao bindar socket que faz comunicacao com desafiante\n");
    closeSockets();
    exit( status: 1);
}

listen(serverSideSocket, n: 1); /*Só aceita a comunicacao com o desafiante*/

```

**Imagem 8** - Parte do código onde é criado o socket para aceitar uma conexão de outro cliente.

## 2.2.4 Procedimento playBattleShip().

Esse é o protótipo do procedimento que irá realizar toda a troca de mensagem da Batalha Naval, ou seja, receber ataque e enviar ataque.

```

/**
 * Função responsável pela lógica do jogo batalha naval em só
 * @param socket é o descritor do oponente
 * @param buffer utilizado para troca de mensagens. Coordenadas de ataque por exemplo.
 * @param playerId identificador que define quem será o primeiro ou segundo jogador
 *      1 - Primeiro a fazer o ataque
 *      2 - Primeiro a receber o ataque
 */
void playBattleShip(int socket, char *buffer, int playerId);

```

**Imagem 9** - Protótipo do procedimento onde irá ocorrer a lógica do batalha naval (lado cliente).

Dentro desse procedimento é feito a leitura do arquivo predefinido de barcos. Para a leitura ser correta, é importante que o nome do arquivo seja "barcos.txt". Caso ocorra qualquer erro durante o processo de leitura do arquivo, o posicionamento dos barcos será iniciado aleatoriamente.

```

FILE *ships; /*arquivo que vai guardar os barcos*/
ships = fopen( filename: "barcos", modes: "rw");
if (ships == NULL) {
    perror(s: "\n[!]ERRO: Falha ao carregar arquivo de barcos!"
        "\nIniciando tabuleiro aleatoriamente!\n");
    randomPositionShips(&myBoard);
    fclose(ships);
} else {
    setShips(ships);
    if (!positionShips(&myBoard)) {
        fprintf(stderr,
            format: "\n[!]ERRO: Formato incorreto no arquivo de barcos!"
                "\nIniciando tabuleiro aleatoriamente!\n");
        bootUpBoard(&myBoard);
        randomPositionShips(&myBoard);
    } else {
        fprintf(stdout, format: "\n[!]ATENCAO: Arquivo de barcos lido com sucesso!\n");
    }
    fclose(ships);
}
}

```

**Imagem 10** - Parte do código em "playBattleShip( )" onde ocorre a leitura do arquivo de barcos.

No procedimento também serão encontrados dois laços, o mais externo é loop principal, que continuará até ter um vencedor no jogo ou caso um jogador se desconecte durante a partida.

```

/*0 loop é encerrado caso tive um vencedor,
  *ou caso o cliente fechar conexão, nesse caso len será < 0*/
while (winner == 0 && len > 0) {

```

**Imagem 11** - Laço mais externo dentro do playBattleShip

O laço mais interno, que é um “do while”, ficará repetindo enquanto o jogador requisita o tabuleiro, ou enquanto ele não faz a entrada correta dos ataques.

```

/*Enquanto n != 2, quer dizer que os argumentos passados
  *não foram lidos corretamente, ou seja, teve entrada inválida.
  *Se o len < 0, quer dizer que algum jogador se desconectou.*/
} while (n != 2 && len > 0);

```

**Imagem 12** - Laço mais inteiro dentro do playBattleShip

No final de cada ataque realizado, é verificado que há um vencedor, caso houver, a variável winner não será mais 0 e o jogo se encerrará.

```

/*Verifica se o jogo acabou em algum dos dois tabuleiros.
 * Caso tenha acabado winner recebe o valor de player, que é a vez de determinado jogador
 *Um exemplo: Jogador 1 (playerID = 1), Jogador 2 (playerID = 2)
 *caso player = 2, winner = 2, isso quer dizer que o ID do vencedor
 *é 2, no caso, o Jogador 2*/
if (gameOver(myBoard) || gameOver(enemyBoard)) {
    winner = player;
}

```

Imagem 13 - Parte do código onde é verificado se há algum vencedor.

## 2.3 BattleShip.h BattleShip.c

O uso no geral da estrutura BattleShip é simples, porém a sua implementação é um pouco complexa, em outras palavras mal implementada, porém aparentemente funcional.

A estrutura do tabuleiro apresentada é essa:

```

/**
 * Estrutura do tabuleiro
 * board é tabuleiro em si
 * nAttack é o número de attacks feito.
 * nHits é o numero de acertos feito.
 */
typedef struct board_ship {
    int board[TAM][TAM];
    int nAttacks;
    int nHits;
} Board;

```

Imagem 14 - Estrutura do tabuleiro da Batalha Naval

A inicialização do board possui uma ordem específica para funcionar. Sendo ela:

**void bootUpBoard(Board \*b);** nesse procedimento será iniciado o tabuleiro, todos os índices da matriz será inicia com **ocean**, e **nAttacks** e **nHits** serão iniciados com 0, depois é preciso ler o arquivo de barcos com o procedimento **void setShips(FILE\* arqShipsPositions);** nesse procedimento tudo o que for lido será atribuído na matriz "shipsMatrix" que se encontra em BattleShip.c e finalmente **bool positionShips(Board \*b);** que irá de fato posicionar os navios no tabuleiro, e retornará true (1) se todos navios foram posicionados corretamente ou false (0) se houve algum erro. Porém caso queira iniciar automaticamente, só é preciso chamar os procedimentos **void bootUpBoard(Board \*b);** e **void randomPositionShips(Board \*b);**

Na função **bool positionShips(Board \*b);** série de verificações são feitas para garantir a integridade do arquivo, sendo as mais importantes:



```
bool inputVerify(int ship, int startRow, int endRow,
                int startCollum, int endCollum, int dir) {
```

**Imagem 15** - Função que verifica integridade do arquivo de barcos.

```
bool place(Board *b, int ship, int direction,
           int way, int startRow, int startCollum) {
```

**Imagem 16** - Função que verifica integridade do arquivo de barcos.

A primeira resumidamente verifica a coesão dos argumentos passados, por exemplo, se o início e fim da linha correspondem ao tamanho de um navio, e a segunda verifica se o navio vai passar em cima de outro navio, ou se o navio passará dos limites do tabuleiro. Também vale ressaltar o *switch case* que recebe o primeiro argumento que são os barcos.

```
/*Dependendo do navio lido, atribui em ship o tamanho do navio*/
switch (ship) {
    case 'P':
        ship = aircraftCarrier;
        break;
    case 'N':
        ship = tankShip;
        break;
    case 'T':
        ship = torpedoBoat;
        break;
    case 'S':
        ship = submarine;
        break;
    default:
        /*Se não foi lido nenhum desses navios, então a entrada é inválida*/
        return false;
}
```

**Imagem 17** - Switch case que verifica a entrada dos navios no arquivo de barcos

O tabuleiro pode possuir apenas três estados nos seus índices:

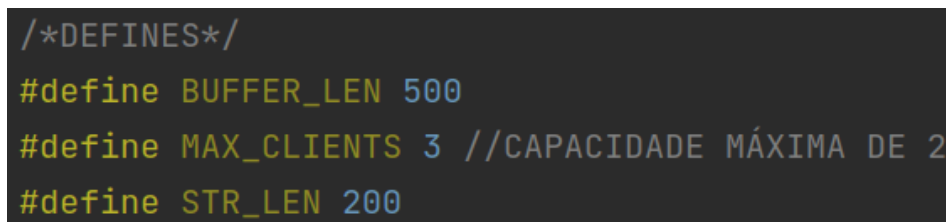
```
/**
 * Estados do tabuleiro.
 * Representação gráfica:
 * ocean = '~' oceano
 * missHit = '*' errou o tiro
 * strike = 'X' acertou o tiro
 */
enum boardStates {
    ocean = -1, missHit, strike
};
```

**Imagem 18** – Possíveis estados do tabuleiro

### 3. DECISÕES DE IMPLEMENTAÇÃO

Apesar de nas especificações do trabalho estar dizendo que o servidor poderá aceitar duas conexões, eu fiz algo um pouco diferente. O servidor poderá aceitar quantas conexões desejar, e funcionará como um lobby com vários jogadores conectados, e também o servidor poderá batalhar com vários jogadores simultaneamente.

A quantidade de jogadores que poderão se conectar ao servidor é controlada por um *define*.



```
/*DEFINES*/
#define BUFFER_LEN 500
#define MAX_CLIENTS 3 //CAPACIDADE MÁXIMA DE 2
#define STR_LEN 200
```

**Imagem 19** – Constantes que definem a capacidade máxima de usuários conectados no servidor.

Diz que a capacidade máxima é 2, pois o servidor também conta como um jogador, portanto apenas 2 outros clientes poderão se conectar. Então aumentando `MAX_CLIENTS`, aumentará a capacidade de jogadores no servidor. Mas vale ressaltar que se deve tomar cuidado com o tamanho do `BUFFER_LEN` e `STR_LEN` caso for aumentar a capacidade máxima de conexões no servidor.

### 4. RETRANSMISSÃO DE MENSAGENS

A retransmissão de mensagens com o servidor se dá de duas formas diferentes:

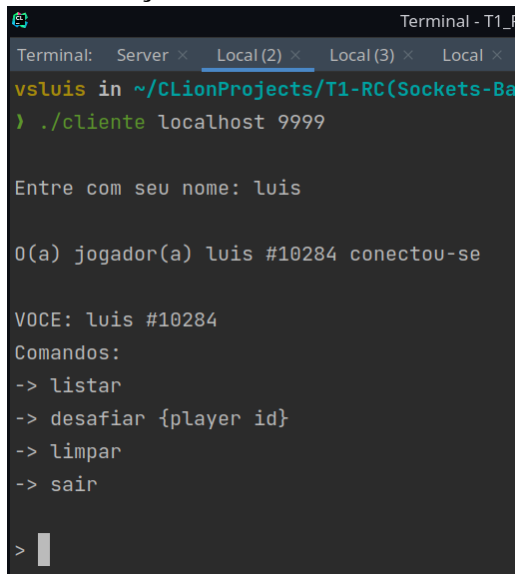
- **Um cliente está no *lobby***
  - Quando um cliente está no *lobby* os comandos são separados em argumentos. Por exemplo, quando o cliente desafia um jogador: "desafiar fulano 10638". Essa string será quebrada em 3 argumentos: "desafiar", "fulano" e "10638" e nesse caso, o servidor retornará o endereço e a porta desse jogador. Outro exemplo é o comando "listar", o servidor enviará a lista de jogadores online. Nesse caso esse comando só precisa de um argumento, ou outros serão ignorados.
- **Um cliente está em uma partida**
  - No caso de *Player vs Player*, a troca de mensagens se dá por enviar as coordenadas de ataque que é separado em dois argumento, por exemplo "15 a" é separado em "15" e "a". Após enviar o ataque, imediatamente recebe a mensagem se acertou ou errou o ataque feito, e passa a aguardar a mensagem de ataque do outro oponente, que ao receber, imediatamente

envia a flag se ele acertou ou errou o tiro. Também é possível enviar "M" ao invés do ataque, nesse caso o *player* responderá enviando o tabuleiro dele, que será exibido. Caso você receba "M", será avisado que o jogador requisitou o mapa. Se um jogador fizer um ataque inválido, por exemplo: "20-a" ou "1-a", ele perderá a vez.

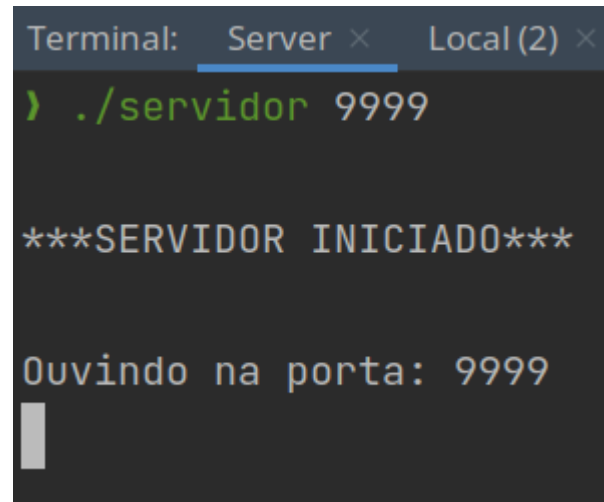
- No caso *Player* vs Servidor, a troca de mensagens não diferencia muito de *Player* vs *Player*. A maior diferença é que o Servidor não irá requisitar o mapa, e seus ataques serão imediatos, pseudoaleatórios e sempre válidos.

## 5. TESTES MOSTRANDO QUE O PROGRAMA ESTÁ FUNCIONANDO DE ACORDO COM AS ESPECIFICAÇÕES

### 5.1 Execução correta

A terminal window titled 'Terminal - T1\_F' with tabs for 'Server', 'Local (2)', 'Local (3)', and 'Local'. The 'Local (2)' tab is active. The prompt is 'vsluis in ~/CLionProjects/T1-RC(Sockets-Ba)'. The user enters './cliente localhost 9999'. The output shows: 'Entre com seu nome: luis', '0(a) jogador(a) luis #10284 conectou-se', 'VOCE: luis #10284', 'Comandos:', and a list of commands: '-> listar', '-> desafiar {player id}', '-> limpar', '-> sair'. The prompt returns to '> '.

**Imagem 20** – Precisa ser passados dois argumentos (ip/nome e número da porta) para o cliente executar.

A terminal window titled 'Terminal: Server' with tabs for 'Server' and 'Local (2)'. The 'Local (2)' tab is active. The prompt is '> ./servidor 9999'. The output shows: '\*\*\*SERVIDOR INICIADO\*\*\*' and 'Ouvindo na porta: 9999'. A cursor is visible on the line 'Ouvindo na porta: 9999'.

**Imagem 21** – Precisa ser passado um argumento (número da porta) para o servidor executar.

## 5.2 Execução incorreta

```
) ./servidor
Uso: ./servidor "numero da porta"
vsluis in ~/CLionProjects/T1-RC(Sockets-BattleShip)
) ./servidor 1

[!]ERRO: Falha no bind do servidor!
: Permission denied
vsluis in ~/CLionProjects/T1-RC(Sockets-BattleShip)
) ./servidor -1

Numero de porta invalido: -1
vsluis in ~/CLionProjects/T1-RC(Sockets-BattleShip)
)
```

**Imagem 22** – Executando sem o argumento (porta), passando uma porta que já é usada por um protocolo e passando uma porta inválida (negativa)

```
) ./cliente
Uso: ./cliente "ip/nome" "porta"
vsluis in ~/CLionProjects/T1-RC(Sockets-BattleShip)
) ./cliente localhost
Uso: ./cliente "ip/nome" "porta"
vsluis in ~/CLionProjects/T1-RC(Sockets-BattleShip)
) ./cliente localhost -1
[!]ERRO: Numero de porta invalido! (-1)
vsluis in ~/CLionProjects/T1-RC(Sockets-BattleShip)
) ./cliente localhost 1

Entre com seu nome: luis

[!]ERRO: Falha ao se conectar com o servidor...
: Connection refused
vsluis in ~/CLionProjects/T1-RC(Sockets-BattleShip) took 2s
) ./cliente 127.0.0.1 9999

Entre com seu nome: j

0(a) jogador(a) j #11848 conectou-se
```

**Imagem 23** – Executando sem os parâmetros corretos, passando porta inválida e passando porta que já é usada por outro protocolo.

## 5.3 Compilação sem warnings ou erros

```
) make
gcc -c BattleShip.c
gcc BattleShip.o cliente.c -o cliente -lpthread -w -Wall -Wextra -pedantic
gcc BattleShip.o server.c -o servidor -lpthread -w -Wall -Wextra -pedantic
vsluis in ~/CLionProjects/T1-RC(Sockets-BattleShip)
)
```

**Imagem 24** – Compilação com -Wall -Wextra -pedantic sem warnings ou erros.

## 5.4 Arquivo de barcos correto

```
P 1-a 5-a
N 1-b 4-b
N 1-c 4-c
T 1-d 3-d
T 1-e 3-e
T 1-f 3-f
S 1-g 2-g
S 1-h 2-h
S 1-i 2-i
S 1-j 2-j
```

**Imagem 25** – Arquivo de barcos com entradas corretas.

```
[!]ATENCAO: Arquivo de barcos lido com sucesso!
```

**Imagem 26** – Após iniciar uma partida, essa mensagem é mostrada no todo, indicando que o arquivo de barcos não teve erros durante a leitura ou posicionamento dos barcos.

## 5.5 Arquivo de barcos incorreto

```
P 1-a 10-a
N 1-b 4-b
N 1-c 4-c
T 1-d 3-d
```

**Imagem 27** – O tamanho de P não condiz com as parâmetros passados

```
[!]ERRO: Formato incorreto no arquivo de barcos!
Iniciando tabuleiro aleatoriamente!
```

**Imagem 28** – Mensagem dizendo que houve algum erro durante o processo de leitura ou posicionamento dos barcos.

## 5.6 Máximo de duas conexões no servidor

```
***SERVIDOR INICIADO***

Ouvindo na porta: 9999

[LOG]: 0(a) jogador(a) luis #10837 conectou-se

[LOG]: 0(a) jogador(a) pedro #11664 conectou-se

[!]ERRO: Capacidade maxima atingida. Conexao rejeitada: 127.0.0.1::43972
```

**Imagem 29** – O Log mostra que dois jogadores se conectaram e ainda estão conectados. Quando outro jogador tenta se conectar, ele é aceito e desconectado imediatamente, pois a capacidade máxima já foi atingida.

## 6. PRINTS MOSTRANDO O CORRETO FUNCIONAMENTO DO CLIENTE E SERVIDOR

### 6.1 Comando listar

```
> listar

Jogadores online:
-> Servidor #0
-> luis #10284
-> fulano #11539

>
```

**Imagem 30** – Mensagem do servidor após o comando listar.

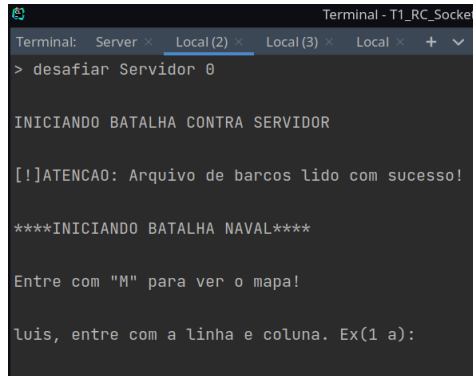
```
> listar

Jogadores online:
-> Servidor #0
-> luis #10284 (in-game)
-> fulano #11539

>
```

**Imagem 31** – O mesmo comando, porém é notável que o servidor também retorna quando algum cliente já está em jogo.

## 6.2 Comando desafiar



```
Terminal - T1_RC_Socket
Terminal: Server x Local(2) x Local(3) x Local x + v
> desafiar Servidor 0

INICIANDO BATALHA CONTRA SERVIDOR

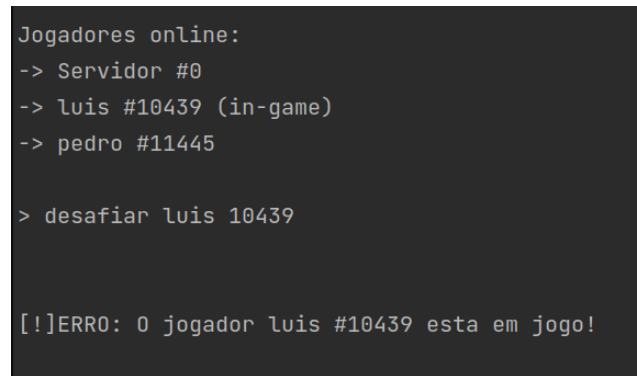
[!]ATENCAO: Arquivo de barcos lido com sucesso!

****INICIANDO BATALHA NAVAL****

Entre com "M" para ver o mapa!

luis, entre com a linha e coluna. Ex(1 a):
```

**Imagem 32** – Após desafiar o servidor, automaticamente entra em batalha.

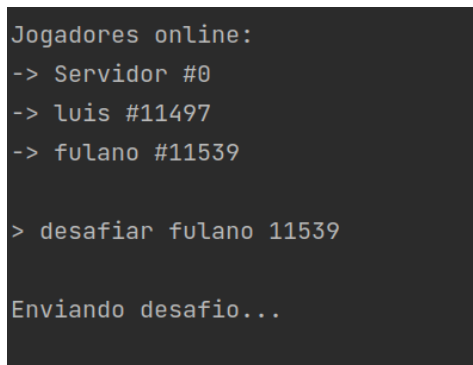


```
Jogadores online:
-> Servidor #0
-> luis #10439 (in-game)
-> pedro #11445

> desafiar luis 10439

[!]ERRO: O jogador luis #10439 esta em jogo!
```

**Imagem 33** – O servidor não retorna o endereço e a porta do jogador quando ele já está em jogo, ao invés, manda uma mensagem dizendo que o player já está em jogo.

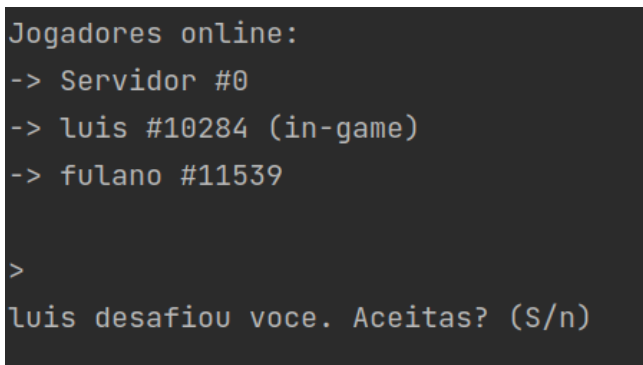


```
Jogadores online:
-> Servidor #0
-> luis #11497
-> fulano #11539

> desafiar fulano 11539

Enviando desafio...
```

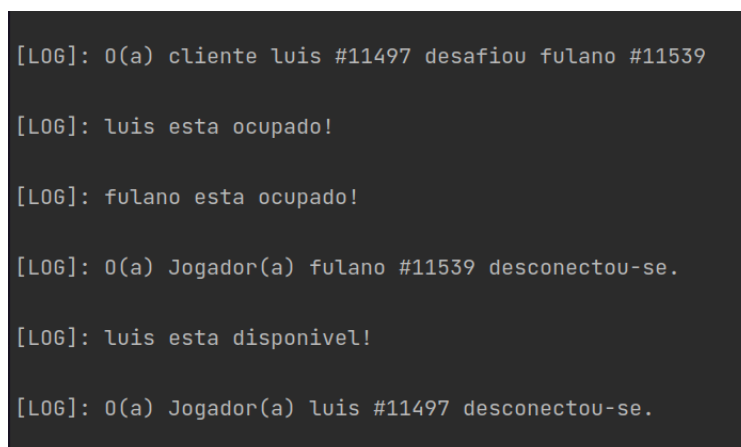
**Imagem 34** – Quando é enviado um desafio para um player, precisa aguardar ele aceitar ou recusar.



```
Jogadores online:
-> Servidor #0
-> luis #10284 (in-game)
-> fulano #11539

>
luis desafiou voce. Aceitas? (S/n)
```

**Imagem 35** – Mensagem que apareceu quando recebe um desafio.



```
[LOG]: 0(a) cliente luis #11497 desafiou fulano #11539

[LOG]: luis esta ocupado!

[LOG]: fulano esta ocupado!

[LOG]: 0(a) Jogador(a) fulano #11539 desconectou-se.

[LOG]: luis esta disponivel!

[LOG]: 0(a) Jogador(a) luis #11497 desconectou-se.
```

**Imagem 36** – Logs do servidor mostrando quando um cliente desafia outro, quando um cliente está ocupado (entra e jogo) e disponível (acaba o jogo) e também quando algum cliente se desconecta.

## 6.3 Em partida

### 6.3.1 Batalha contra o servidor

```
luis, entre com a linha e coluna. Ex(1 a): 1 a

[!ATENCAO!]: Enviando ataque para o(a) Servidor fazendo o ataque

[!ATENCAO!]: Voce ERROU o ataque(1-a) no(a) jogador(a) Servidor!
flag se acertou ou errou
Esperando por Servidor...

[!ATENCAO!]: O(a) jogador(a) Servidor atacou em 9-a ataque do servidor

[!ATENCAO!]: O(a) jogador(a) Servidor ERROU o ataque(9-a)!
indicando q o servidor errou
Entre com "M" para ver o mapa!

luis, entre com a linha e coluna. Ex(1 a):
```

**Imagem 37** – Fazendo um ataque correto contra o servidor. Após o ataque, uma resposta imediata é recebida, indicando se você acertou ou errou o ataque.

```
luis, entre com a linha e coluna. Ex(1 a): 1-a

[!ATENCAO!]: Enviando ataque para o(a) Servidor
[!ATENCAO!]: ATAQUE INVALIDO, PERDEU A VEZ >:(

Esperando por Servidor...
```

**Imagem 38** – Mensagem ao tentar realizar um ataque com a formatação inválida ou coordenadas inválida.

```
luis, entre com a linha e coluna. Ex(1 a): M

TABULEIRO DO(A) Servidor:

  a   b   c   d   e   f   g   h   i   j   k   l   m   n   o
1  *   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
2  ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
3  ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
4  ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
5  ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
6  ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
7  ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
8  ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
9  ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
10 ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
11 ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
12 ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
13 ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
14 ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
15 ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~   ~
```

**Imagem 39** – Requisitando o tabuleiro do servidor. Note que o ataque que foi realizado acima já aparece marcado no tabuleiro.

### 6.3.2 Batalha contra Player

```
luis, entre com a linha e coluna. Ex(1 a): 3 a

[!ATENCAO!]: Enviando ataque para o(a) fulano

[!ATENCAO!]: Voce ACERTOU o ataque(3-a) no(a) jogador(a) fulano!

[!ATENCAO!]: Parabens luis, VOCE EH O(a) VENCEDOR(a)!

Sua precisao de acerto: 100.00% (3/3)
```

**Imagem 40** – Realizando um ataque válido contra um player. Nota-se que esse foi o último ataque do jogo, pois teve um vencedor. (Nesse caso diminui a quantidade de acertos para 3, para o jogo acabar rápido).

```
Esperando por luis...

[!ATENCAO!]: O(a) jogador(a) luis atacou em 3-a

[!ATENCAO!]: O(a) jogador(a) luis ACERTOU o ataque(3-a)!

[!ATENCAO!]: luis ganhou dessa vez...

Sua precisao de acerto: 100.00% (2/2)
```

**Imagem 41** – Mensagem após receber um ataque. Nota-se que na primeira mensagem o jogador fica aguardando receber as coordenadas de ataque. E também esse foi o último ataque do jogo, pois teve um vencedor.

```
***INICIANDO BATALHA NAVAL***

Esperando por luis...

[!ATENCAO!]: luis requisitou o tabuleiro. Enviando...

Esperando por luis...
```

**Imagem 42** – Mensagem mostrada após o jogador requisitar o tabuleiro.

## 7. CONCLUSÃO E REFERÊNCIAS BIBLIOGRÁFICAS

O desenvolvimento desse projeto foi crucial para o aprendizado da API sockets, e também no paradigma cliente/servidor. Além disso, por se tratar de programar na linguagem C acarretou que também houve um aprendizado a mais da linguagem, o que sempre é bem-vindo.



Uma das maiores aquisições nesse projeto, foi a utilização de threads, o que estranhamente facilitou bastante o desenvolver do projeto, e também serviu como uma motivação de fazer algo maior. Fiquei orgulhoso por conseguir utilizar as threads, creio que a mecânica do servidor funcionar como um *lobby* ficou muito interessante, pois da caminho para outros projetos, outros tipos de jogos por exemplo. Todavia a implementação não foi uma das mais elegantes, principalmente na parte da biblioteca da Batalha Naval “BattleShip.h”

Enfim, esse foi um projeto interessante e também motivador. Trabalhos desse estilo deveriam ser mais explorado pelos educadores, pois o aprendizado que traz ao alunos é imensurável.

## REFERÊNCIAS BIBLIOGRÁFICAS

Referência para o uso da ferramenta Make

<https://cs.colby.edu/maxwell/courses/tutorials/maketutor/>

[https://github.com/ahockersten/makefile\\_tutorial](https://github.com/ahockersten/makefile_tutorial)

Referência que serviu de apoio para o desenvolvimento do trabalho

[https://man7.org/linux/man-pages/man3/inet\\_ntop.3.html](https://man7.org/linux/man-pages/man3/inet_ntop.3.html)

[https://man7.org/linux/man-pages/man3/pthread\\_detach.3.html](https://man7.org/linux/man-pages/man3/pthread_detach.3.html)

<https://www.gta.ufrj.br/ensino/eel878/sockets/htonsman.html>

<https://github.com/nikhilroxtomar/Chatroom-in-C>