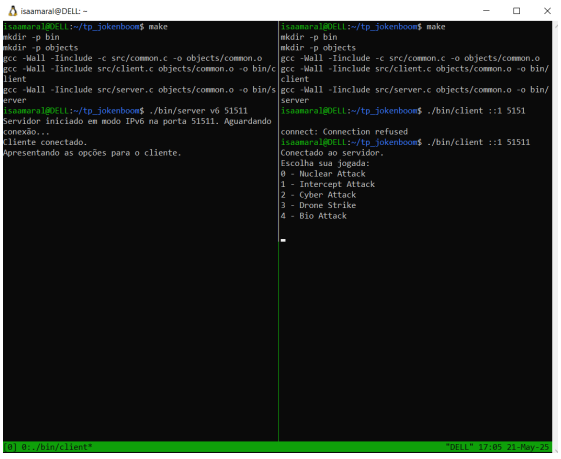
TRABALHO PRÁTICO 1 - JOKENBOOM
Isabella de Paula Amaral - 2023039023

Dificuldades Técnicas

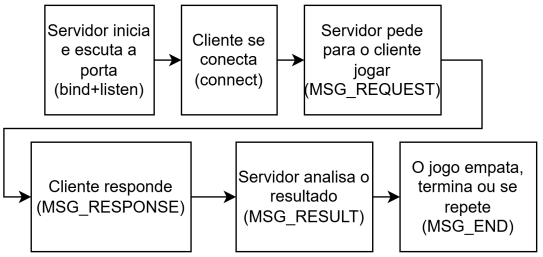
Inicialmente, o primeiro problema enfrentado é conseguir realizar a comunicação entre um cliente e um servidor utilizando sockets, protocolo TCP e conseguir lidar com IPv4 e IPv6. Creio que todas as dificuldades foram muito facilitadas com as videoaulas do professor Ítalo Cunha. Sem elas, o tempo de realização do TP teria sido bem mais desgastante e demorado. Mesmo com as aulas, não estava muito acostumada com as nomenclaturas e entender o código foi trabalhoso, mas ao mesmo tempo recompensador ao ver a comunicação funcionando bem.

Além dessa comunicação, eu realizei o trabalho através do WSL (minha máquina não suportaria dual boot), então tive que configurar todo o ambiente de uma forma mais improvisada do que no Linux, utilizando tmux e tendo dificuldade em manipular os arquivos por estar no Windows e estar acostumada com ele. Apesar disso, não tive muitos problemas e os resolvi com ajuda de experiências passadas das aulas de programação e desenvolvimento de software.



Dificuldades de Lógica

Assim que a parte de comunicação estava pronta, a próxima etapa foi pensar como relacionar a lógica de conexão com a lógica do jogo, como elas iriam conversar e como o programa iniciaria, lidaria com erros, quais seriam os loops e o que finalizaria loops infinitos. Para essa parte, o que me ajudou a iniciar foram as estruturas de mensagem que deveriam seguir à risca o que a documentação do TP exigia e ter essa base do GameMessage me ajudou a organizar a sequência de ações da figura abaixo.



Funções Criadas – Com a minha comunicação funcionando e uma ideia geral de como ficaria o resto, criei no arquivo do servidor funções que com certeza deveriam existir na lógica principal do jogo, sendo elas a action_name() e a winner(). Elas seriam úteis porque o código entenderia as funções do jogo através de números (0 a 4) mas era preciso “printar” os nomes das ações de fato e também para verificar o ganhador seguindo as regras do jogo, respectivamente.

Para criar a lógica do ganhador, criei inicialmente várias sequências de “else if” que funcionavam, porém tinham um código muito extenso e difícil de entender. Por isso, alterei a

lógica para uma matriz pensando em linhas como o jogador e colunas como o cliente.

```c
int winner(int player, int server) {

    int win[5][5] = {
        {-1, 0, 1, 1, 0}, // -1 empata, 1 jogador vence e 0 servidor ver
        {1, -1, 0, 0, 1},
        {0, 1, -1, 1, 0},
        {0, 1, 0, -1, 1},
        {1, 0, 1, 0, -1}
    };

    return win[player][server];
}
```

Além dessas funções, era necessária uma função que lidasse com a lógica central do jogo, o que viria a ser a função game(), que se relacionaria com a função principal de conexão através do socket do cliente e com as mensagens através do Game Message.

Na lógica de game(), variáveis foram criadas para armazenar valores que precisam ser exibidos no final do jogo e durante (pontuações dos dois jogadores) e o loop que continua ativo enquanto o cliente quiser jogar. No loop, o servidor altera o tipo da mensagem e envia mensagens no terminal, controlando o que acontece dependendo da mensagem e tratando erros.

No código do cliente, percebi que foi necessário criar a função clean_n() porque quando o cliente escrevia algo que não era um número, esse caractere ficava armazenado e causava erros imprevisíveis. Ela resolveu o problema sendo sempre chamada depois de “scanf” para limpar esse lixo.

Além dessa função, o loop da main do cliente envia mensagens diferentes para o servidor com base no tipo (mudado pelo servidor) da mensagem, terminando de resolver tudo o que era necessário para a comunicação funcionar. Um exemplo de aplicação dependendo do tipo de mensagem se encontra na imagem abaixo.

```c
switch (msg.type) {
    case MSG_REQUEST:
        printf("%s\n", msg.message);
        msg.type = MSG_RESPONSE;
        if (scanf("%d", &msg.client_action) != 1) {
            msg.client_action = -1;
        }
        clean_n();
        send(s, &msg, sizeof(msg), 0);
        break;
```

Dificuldade Durante Testes

Durante os testes que eu realizei o código gerou alguns problemas que não tinham sido percebidos durante a criação dele. Uma das problemáticas que mais demandou tempo foi arrumar todas as mensagens que deveriam aparecer no terminal, algumas não tinham sido colocadas no começo, outras estavam aparecendo em locais errados e também não estavam com pontuação correta. Em sua maioria, eu esqueci de apresentar para o servidor o que estava acontecendo no jogo, só estava mostrando os acontecimentos para o cliente. Exemplo de parte do código em que avisos são enviados para ambos:

```c
msg.type = MSG_REQUEST;
snprintf(msg.message, MSG_SIZE, "Escolha sua jogada:\n0 - Nuclear Attack\n1 - Intercept
Attack\n2 - Cyber Attack\n3 - Drone Strike\n4 - Bio Attack\n");
printf("Apresentando as opções para o cliente.\n");
send(csock, &msg, sizeof(msg), 0);

if (recv(csock, &msg, sizeof(msg), 0) <= 0) {
    break;
}

printf("Cliente escolheu %d.\n", msg.client_action);
```

Ainda sobre mensagens no terminal, inicialmente o programa estava utilizando sprintf para formatar as strings que deveriam ser enviadas. Porém, o código deu bastante erro já

que eu deixei de alterar o tamanho da mensagem que estava 1024 para 256, então resolvi mudar para snprintf, o que no final acabou sendo mais seguro já que isso previne também mensagens muito grandes de entrada ou que o placar atinja um valor muito grande.

Além das mensagens, o fluxo estava errado quando aparecia algum erro de digitação na hora do cliente escolher se quer jogar novamente. Se ele digitasse um erro, o cliente encerrava automaticamente, mas o servidor deveria perguntar outra vez. Isso acontecia porque eu zerava a variável keep_playing quando um erro aparecia, o que deveria acontecer só quando o cliente respondesse com zero. O ocorrido apareceu também quando o cliente desconecta e o servidor desconecta em conjunto, o que não deveria acontecer e foi corrigido.

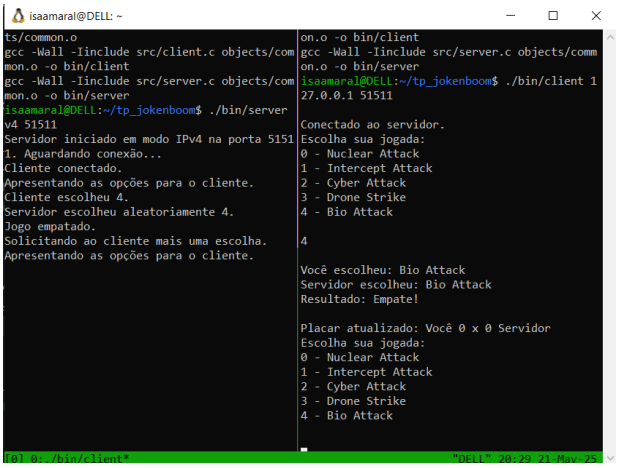
```
if (!valid_response) break;

keep_playing = msg.client_action;
```

Saindo dos erros menores, uma dificuldade enfrentada foi rodar os arquivos todas as vezes em terminais diferentes sempre que era necessário testar. O Makefile poderia resolver esse problema, mas não consegui fazer os comandos de tmux funcionarem dessa forma. No final, o Makefile ficou dessa forma:

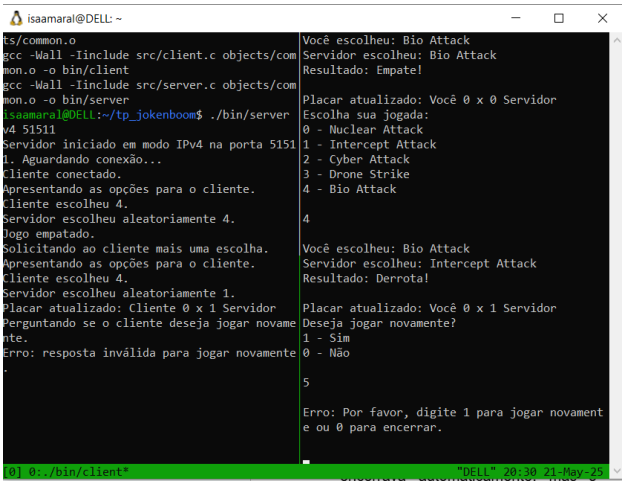
```
Makefile
all:
	mkdir -p bin
	mkdir -p objects
	gcc -Wall -Iinclude -c src/common.c -o objects/common.o
	gcc -Wall -Iinclude src/client.c objects/common.o -o bin/client
	gcc -Wall -Iinclude src/server.c objects/common.o -o bin/server

```

Imagens do Terminal

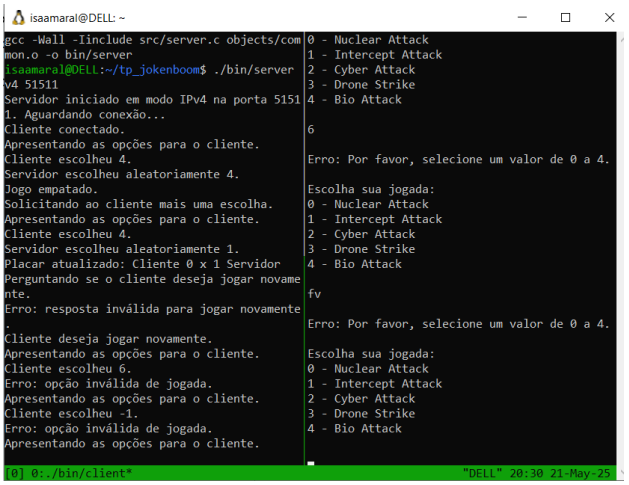
Jogada normal:



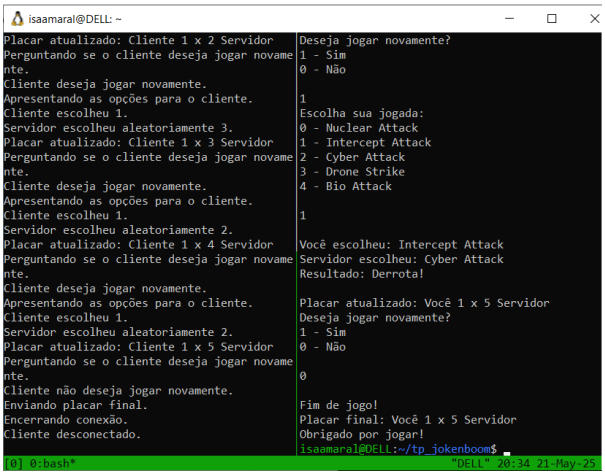
Com erro em tentar novamente:



Com erro no número da jogada:



Fim de jogo:



Código do projeto no GitHub: https://github.com/lsaamaral/TP1-Jokenboom