Trabalho Prático 1

Redes de Computadores

Starfleet Protocol

Arthur Araujo Rabelo

2022091552

1. Desafios encontrados

Para implementar o programa, surgiram alguns desafios, os quais estão listados a seguir:

- Familiarizar-se com os métodos e estruturas das bibliotecas de *socket*.
- Organizar os "*sends*" e "*recvs*" para que o cliente e o servidor estejam organizados e coerentes entre si, garantindo o funcionamento correto da comunicação entre eles.
- Definir como a combinação dos ataques e seus resultados serão implementados.
- Definir como a mensagem será montada a cada combinação de ataques.
- Definir quais informações, além da mensagem, serão enviadas pelo *socket*.
- Por fim, não ficou muito claro na documentação se toda a *struct* deveria ser enviada ou se somente o atributo *message*. Além disso, também poderia ter sido melhor esclarecido se o tipo da struct deveria continuar como *int*, porque os tipos usados nas funções de conversão *htonl/htons* e *ntohl/htons* são *uint32_t* e *uint16_t*, e isso poderia ocasionar problemas.

2. Estratégias adotadas

Para os desafios encontrados acima, foram adotadas as seguintes estratégias, respectivamente:

- Ler o Beej's Guide to Network Programming para entender como funciona a programação com *sockets*. Esse material foi muito proveitoso, principalmente por causa dos exemplos de código e das informações

muito úteis que ele disponibiliza. Para citar um exemplo, usei *AF_UNSPEC* no cliente, de maneira que ele aceite tanto endereços em IPv4 quanto em IPv6.

- Para organizar a comunicação entre o cliente e o servidor, utilizei algumas variáveis de controle dentro do laço de comunicação. Uma delas foi a *loop* que conta as iterações dentro do laço de comunicação, e serve especificamente para controlar as mensagens de início do jogo, que são enviadas nas duas primeiras iterações. A segunda informação de controle foi o *type* da mensagem, que serve para finalizar o laço se algum dos dois escapou ou se o jogo terminou. No servidor, é utilizada também a variável *game_over*, que termina o laço caso alguma das naves esteja destruída. Fora do laço, são enviadas as mensagens finais: *MSG_GAME_OVER* e *MSG_INVENTORY*.
- A combinação dos ataques foi implementada a partir do *scruct Combination*. Essa estrutura possui 3 atributos: *client_damage*, que diz quanto de dano o cliente vai sofrer; *server_damage,* que diz quanto de dano o servidor vai sofrer; por fim, *msg* é a mensagem resultante da combinação. Foi declarada então uma matriz 5x5 que define os valores desses atributos para uma combinação *ij*, onde *i* é a ação do cliente e *j* é a ação do servidor.
- As mensagens são montadas a partir da *struct BattleMessage* e da função *update_message*, que recebe um objeto da referida estrutura e de acordo com o *type*, definido pelo *enum MessageType*, constrói a mensagem e a atualiza no objeto. Foram criadas algumas funções auxiliares para ela.
- Além do texto da mensagem, o servidor também envia o *type*, visto que o cliente precisar estar ciente do tipo de mensagem para controlar a comunicação com o servidor. Se o tipo for *MSG_GAME_OVER* ou *MSG_ESCAPE*, por exemplo, o cliente precisa terminar o loop de comunicação e receber as mensagens finais.

- Além disso, configurei o *socket* para que a porta usada pelo servidor seja possível de ser reutilizada assim que a conexão encerrar, sem que seja necessário aguardar. Para tanto, basta configurar o *socket* com *SO_REUSEADDR*.
- De início, implementei a comunicação enviando somente o atributo *message* do servidor para o cliente, deixando somente o servidor manipular o *BattleMessage*. Depois de uma dúvida sanada no fórum, soube que a *struct* inteira deveria ser enviada. Essa mudança foi no entanto muito tranquila, já que todo o fluxo de comunicação já estava bem estruturado, bastando adicionar a conversão dos atributos de *BattleMessage* e alterar os meus métodos que chamam o *send* e o *recv*.
- Por fim, decidi deixar a struct com seus atributos de tipo *int* mesmo, assim como foi passado no enunciado, mas ainda reforço que isso poderia ocasionar problemas.

3. Possíveis melhorias

Uma melhoria que pode ser feita é possibilitar que o servidor receba vários clientes. Isso pode ser feito com um *fork*, alocando cada cliente para um conexão própria, cada uma com seu *file descriptor*.

Outra melhoria seria um controle melhor das ações escolhidas pelo servidor: a geração de números aleatórios pelo *rand()* é ruim, apesar de simples. O servidor foge muitas vezes e é difícil terminar um jogo sem que isso aconteça. O número da ação poderia ser calculado de acordo com um peso para cada tipo de ação. Ações maiores, escudos e escape, teriam uma probabilidade menor de ocorrer. Ataques aconteceriam com maior frequência.

A gestão de probabilidades acima abriria uma brecha para outra melhoria: escolher um nível de jogo (de 1 a 5, por exemplo). Quanto maior o nível, mais o

servidor se defende e ataca estrategicamente, com base em um cálculo de probabilidades feito a partir das rodadas anteriores. Quanto mais fácil, o servidor foge mais e ataca pouco.

Alguns outros detalhes poderiam ser implementados futuramente, mas que já não são tão impactantes como os acima. Por exemplo, alocar dinamicamente o tamanho mensagem (fixo em 256), possibilitando assim mensagens maiores e maior flexibilidade e personalização do código.