Trabalho Prático 1 de Redes de Computadores

[**Rafael Sant'Ana**]

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

rafael-sant-ana@ufmg.br, 2023087834

1 Introdução

O desenvolvimento do sistema cliente-servidor para a simulação de batalhas interestelares (StarFleet Protocol) em C, utilizando sockets POSIX, apresentou desafios específicos inerentes à programação de rede de baixo nível e à implementação da lógica de jogo determinística. Este documento descreve as principais dificuldades encontradas e as estratégias adotadas para superá-las.

2 Desafios Encontrados na Implementação

Os obstáculos primários concentraram-se em três áreas: a comunicação via sockets, a serialização dos dados e a complexidade das regras de combate.

2.1 1. Gerenciamento de Sockets e Conexão TCP

A principal dificuldade inicial reside na criação e manutenção de uma comunicação TCP robusta em C, sem o uso de bibliotecas de alto nível:

- **Configuração Unificada (IPv4/IPv6):** A necessidade de suportar tanto IPv4 quanto IPv6 a partir de um argumento de linha de comando ('v4' ou 'v6') exigiu que eu usasse com muito cuidadoa fun;'ao `getaddrinfo()`. Garantir que o servidor pudesse ligar corretamente ao endereço (usando `bind()`) e que o cliente pudesse conectar-se ao endereço fornecido de forma agnóstica (`AF_UNSPEC`) foi bem dificil. Eu já tinha feito antes algumas coisas usando a biblitoeca HTTP no Node.js, mas nesse nivel de baixo nível foi novidade.
- **Ciclo de Vida da Conexão:** Gerenciar o `listen()`, `accept()` e o `close()` do *socket* de forma correta, garantindo que o servidor permanecesse ativo após a desconexão de um cliente para aguardar um novo, demandou atenção ao *loop* principal. O uso de `recv()` de forma *blocking* (bloqueante) foi essencial e bem trabalhoso, mas tive que fazer assim para seguir as especificações. Conforme dito em sala, o que me interessou sobre isso é como podemos ter threads para ter multiplas conexões, o que é bem legal.
- No entanto, depois que consegui gerenciar a conexão TCP, tive outros desafios, mas eles foram mais leves. Conectar foi realmente bem trabalhoso. No meu repositorio do github, tem commits que foram literalmente só para ajeitar a parte da conexão. Mas as video aulas do italo ajudaram muito.

2.2 2. Serialização e Transferência de Estruturas de Dados

A comunicação entre cliente e servidor depende da transferência da estrutura `BattleMessage`, o que levanta desafios de serialização:

- **Transferência de Estruturas (Structs):** Enviar e receber a estrutura `BattleMessage` de tamanho fixo (`sizeof(BattleMessage)`) através de `send()` e `recv()` é complicado. É fundamental que ambos os lados concordem *exatamente* com o tamanho e a ordem dos campos para evitar corrupção de dados ou leituras parciais (e ai eu tive que criar um protocol.h). O uso de `sizeof(BattleMessage)` em todas as chamadas foi como resolvi.
- **Limpeza de Buffer:** Em especial no lado do servidor, garantir que todos os campos da estrutura (incluindo a `char message[MSG_SIZE]`) fossem inicializados com `memset` (`0`) antes do uso foi importante para evitar o envio de residuos de dados ou lixos de memória para o cliente.

2.3 3. Implementação da Lógica de Combate (`process_turn`)

A implementação das regras de combate, que representa a lógica central (20% da avaliação), exigiu uma estrutura lógica organizada para lidar com a precedência e as anulações de dano:

- **Prioridade do Hyper Jump (4):** Garantir que o `Hyper Jump` fosse verificado e processado no *loop* principal (`main`) antes de chamar a função `process_turn` foi como eu fiz para respeitar sua regra de prioridade de encerramento imediato. mas dava pra fazer tambem colocando ela dentro do `process_turn` antes de tudo. mas acho que o codigo ficou mais bonito separado.
- **Regras de Anulação e Sobrescrita:** A maior dificuldade dentro de `process_turn` foi estruturar a lógica para que as regras defensivas (Escudos bloqueiam, Camuflagem evita Torpedo) fossem avaliadas corretamente, mas que as regras de confronto (Torpedo vs. Laser) pudessem sobrescrever o dano calculado, resultando em zero ou 20 HP, conforme o caso. O código foi limpo para processar o dano em etapas (Ataque Cliente $\rightarrow$ Ataque Servidor $\rightarrow$ Regras de Confronto). Algo legal é que os 20 de HP vem de um `# DEFINE`, assim, é facil mudar essa regra do jogo
- **Geração de Mensagens:** Concatenar corretamente o resultado textual do turno com o placar final (`Placar: Voce X x Y Inimigo`) no campo `message` da `BattleMessage` exigiu o uso de `snprintf` e `strncat` para evitar *overflow* e garantir a legibilidade. Na verdade, essa foi a seugnda parte mais trabalhosa do trabalho: Lidar com as saídas. No arquivo de documentação original, elas não estão muito bem expressas, e foi um trabalho bem extenso com os meus amigos que estavam fazendo a disciplina determinar quais deveriam ser os prints, dado que eles não batia com os dos exemplos, mas faziam sentido com a especificação. Essa dualidade foi um tanto quanto confusa. Além disso, ficar dar append nas string para gerar as mensagens foi algo que me ensinou muito sobre C. Em python, é um simples `+`, e foi legal ter que mexer isso em um nivel mais baixo e ver denovo como linguagens como Python abstraem essa parte de lidar com strings.

3 Estratégias Adotadas

Para garantir a robustez, legibilidade e manutenibilidade do código em C, foram empregadas as seguintes estratégias de implementação:

- **Modularização por Funções Auxiliares:** A lógica de I/O do socket (`send_message`) e a

finalização do jogo (`finalize_game`) foram encapsuladas em funções separadas, simplificando o *loop* principal (`main`) e o tornando focado apenas na coordenação dos turnos.

- **Definição de Protocolo Compartilhado:** A criação de um arquivo `protocol.h` centralizou as definições de `MessageType` e `BattleMessage`, garantindo que cliente e servidor operassem com as mesmas regras de dados e minimizando erros de desserialização.
- **Estrutura de Estado (GameState):** Utilizar uma estrutura `GameState` dedicada no servidor garantiu que todos os dados críticos (HP, contadores de inventário, turnos) fossem passados por referência para a função `process_turn`, facilitando a leitura e atualização atômica do estado.
- **Tratamento Seguro de Entrada do Cliente:** No lado do cliente, a função `get_client_input` foi implementada utilizando `fgets()` e `sscanf()` para validar não apenas o valor numérico (0 a 4), mas também para descartar entradas muito longas ou que contenham caracteres não numéricos, aumentando a robustez contra erros de usuário.

4 Possíveis Melhorias

Embora o sistema cumpra todos os requisitos funcionais, as seguintes melhorias poderiam ser implementadas em um desenvolvimento futuro:

- **I/O Não Bloqueante e Threads:** Embora o requisito seja para um cliente por vez, o servidor poderia ser aprimorado para I/O não bloqueante, empregando *threads* para permitir que múltiplos clientes se conectassem (e talvez esperassem em uma fila de lobby) (a ideia de threads veio de uma explicação do Flip em aula).
- **Inteligência Artificial (IA) Tática para o Servidor:** A ação do servidor é puramente aleatória (`rand() % 5`). Uma melhoria seria implementar uma lógica simples, como:
  1. Se o HP do servidor estiver abaixo de 40, aumentar a probabilidade de usar `Shields Up` ou `Cloaking`.
  2. Se o cliente usou `Torpedo` (1no turno anterior, o servidor pode priorizar `Cloaking`.
  3. Ou talvez só colocar um pequeno modelo para aprender a melhor estratégia de jogo, talvez com um Reinforcement Learning.
- **Inventário Dinâmico e Classes de Naves:** Adicionar classes de naves (e.g., Dreadnought, Scout) com diferentes valores iniciais de HP e de dano, e habilidades novas. Além disso, introduzir um limite de munição para os `Torpedos` para aumentar o desafio tático.
- **Interface de Terminal (ncurses):** Substituir a interface de terminal simples por uma baseada em *ncurses* ou allegro para criar um display mais dinâmico, exibindo o HP em barras e o histórico de ações lado a lado, melhorando a experiência do usuário.

5 Ambiente de desenvolvimento

Para testar o programa, foi utilizado:

- Ubuntu 22.04.3 LTS e 20.04.4 LTS (PC "madeira"da sala 2019 do ICEX)
- Linguagem C++
- Foi utilizado o compilador G++ na versao 11.4.0 e, no PC "madeira"da sala 2019, 9.4.0

- O primeiro computador usado para desenvolver tem como processador Amd Ryzen 5 1600x e 16gb de ram. Já no PC "madeira"da sala 2019, tem como processador Intel i7-6700 e 16gb de ram.

6 Instruções de Compilação e execução

Abaixo, segue as instruções para compilar e executar o código.

- Acesse o diretório que contém os arquivos do trabalho prático;

```
make
```
(1)

- Com esse comando, deve ser gerado um executavel com o nome `client` e um com o nome `server` em *bin*
- Para executar o servidor, basta executar `./bin/server v4 5000` . Uma vez com o servidor executando, podemos executar o cliente. para isso, basta executar `./bin/client 127.0.0.1 5000`

7 Conclusão

O projeto StarFleet Protocol foi um excelente exercício em programação de sistemas e redes em C. A correta implementação da comunicação TCP e a modelagem determinística das regras de combate em C formaram os maiores desafios, resolvidos por meio de modularização e do foco estrito na manipulação correta dos *sockets* e das estruturas de dados enviadas (definidas nos structs).

8 Referências

Beejs Guide to Network Programming

A playlist de sockets do Italo Cunha

Cormen, Thomas H. Algoritmos: Teoria e Prática, Editora Campus, v. 2 p. 296, 2002