UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Trabalho Prático 1
Introdução a Sistemas Computacionais - TISC
Engenharia de Sistemas 2025/01 - Professor Marcos Augusto Menezes Vieira

Aluna: Raissa Gonçalves Diniz - 2022055823

1. Introdução:

Tendo em vista o atual cenário de tensão entre as potências globais, o trabalho consistiu em implementar um sistema cliente-servidor para a realização de partidas de Jokenpo expandido com 5 opções (Ataque Nuclear, Interceptação de Mísseis, Ciberataque, Bombardeio com Drones e Emprego de Armas Biológicas, cada um com seus domínios de vitória e derrota) entre um jogador humano (cliente) e o computador (servidor, que toma decisões aleatórias). O cliente escolhe manualmente suas jogadas, e a partida continua até que ele decida não jogar novamente. Em caso de empate, novas rodadas são realizadas. O sistema opera tanto em redes IPv4 quanto IPv6, conforme passado no terminal. No final da partida, o servidor retorna o placar.

2. Implementação:

O código é composto por quatro arquivos principais — server.C, client.C, common.C e common.H — além de um Makefile para compilação.

Os arquivos common.C e common.H concentram os elementos compartilhados entre cliente e servidor, como as structs MessageType e GameMessage, fornecidas no enunciado, e funções auxiliares para manipulação de endereços e tratamento de erros. Essas funções incluem: logexit, que imprime mensagens de erro e encerra o programa; addrparse, que converte strings de IP e porta em estruturas de socket; addrtostr, que transforma endereços de socket em strings legíveis; e server_sockaddr, que inicializa uma estrutura de endereço para o servidor escutar conexões em IPv4 ou IPv6.

O arquivo server.C implementa toda a lógica do servidor, que é o principal responsável por controlar o fluxo da aplicação. Cabe a ele iniciar a comunicação com o cliente, enviando a mensagem que solicita a escolha da jogada. Após receber a resposta, o servidor valida a jogada, sorteia sua própria escolha de forma

aleatória, compara os resultados e envia ao cliente o desfecho da rodada. Em caso de empate, o servidor reinicia automaticamente a partida. Ao final de cada rodada, também é o servidor quem pergunta ao cliente se ele deseja continuar jogando. Se o cliente escolher não jogar, a sessão é encerrada do seu lado, mas o servidor ainda fica aberto, pronto para se conectar com outro cliente (sempre um de cada vez). Toda a lógica de controle do jogo, sequência de mensagens e regras está concentrada no lado do servidor.

O arquivo client.C recebe as mensagens do servidor, as exibe no terminal e envia de volta as respostas do usuário, tanto a jogada quanto a decisão de continuar jogando. Embora a maior parte da lógica de validação esteja centralizada no servidor, o cliente realiza uma verificação local simples para evitar o envio de entradas inválidas que não possam ser corretamente interpretadas, que é melhor explicada na próxima seção. A interface com o usuário também está concentrada do lado do cliente, mas sempre orientada pelas mensagens recebidas do servidor.

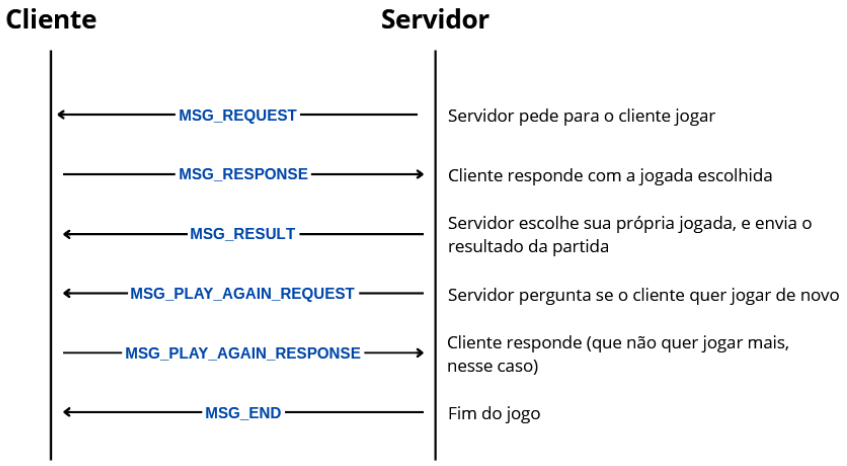

Figura 1: Fluxo de Mensagens. Fonte: Autoria própria.

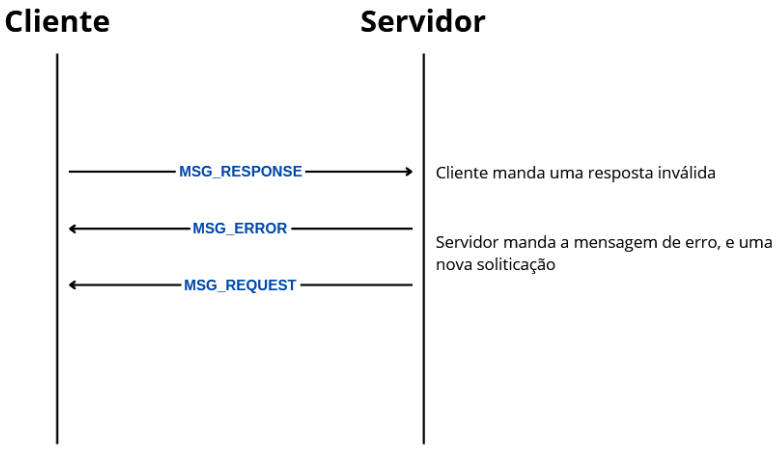

Figura 2: Fluxo de mensagens em caso de erro. Fonte: Autoria própria.

Após compilação do código usando o comando “Make”, a execução do servidor é feita da seguinte maneira:

- ./bin/server <protocol> <port>

O protocolo pode ser v4 (IPv4) ou v6 (IPv6).

Para executar o cliente:

- ./bin/client <server_ip> <port>

3. Discussão:

Uma das maiores dificuldades na realização do trabalho foi em relação a utilização dos structs e a implementação de demais especificações providenciados no enunciado do trabalho, por conta da maneira como o código foi feito inicialmente. A base do código seguiu o tutorial providenciado pelo professor Ítalo Cunha no Youtube, e a lógica do jogo foi gradualmente implementada em cima disso. Quando o código estava praticamente funcionando, foi quando decidi adequá-lo para os usos dos structs, o que originou vários problemas, tanto para seguir todos os requisitos, quanto para fazer essa adaptação em relação ao que já havia sido criado antes. Além disso, a implementação seguiu, inicialmente, apenas os padrões dos resultados providenciados na seção 7 do enunciado, sem grande atenção a algumas especificações previamente mencionadas em relação à ordem das mensagens. Por conta disso, grande parte da lógica de validação de mensagens do cliente havia sido feita do seu próprio lado, e não do servidor, o que teve que ser gradualmente alterado. De início, o cliente quem também enviava a mensagem, com base na pergunta que aparecia no próprio terminal, e não o servidor quem começava a comunicação. Tais mudanças foram incrementalmente feitas e o código passou a funcionar, apesar das dificuldades.

Outra questão não mencionada no enunciado, era um pequeno erro que estava acontecendo quando o cliente apertava a tecla “enter” ao invés de selecionar alguma das opções válidas de jogada. Opções não válidas já estavam sendo corretamente validadas, mas por conta da natureza da função atoi (que converte um caractere em um inteiro), ela estava convertendo o caractere vazio para o valor 0, o que é uma jogada válida para o servidor. Isso foi corrigido com uma validação do lado do próprio cliente, uma vez que o valor que estava chegando ao servidor era válido. Algo semelhante aconteceu com a validação da resposta do cliente à pergunta se ele gostaria de jogar novamente. Ambos os desafios foram superados, esperando que o usuário escrevesse um valor válido diferente de um caractere vazio para continuar com o fluxo normal do jogo.

Apesar da sugestão do professor de usar scanf para ler a entrada do cliente, essa abordagem apresenta algumas desvantagens em comparação com a maneira previamente mencionada, que usa fgets. O scanf com %d é menos robusto porque, ao encontrar uma entrada não numérica, ele frequentemente retorna 0, o que é problemático, pois 0 é uma jogada válida no jogo. Além disso, o comportamento do scanf pode ser inconsistente quando o usuário simplesmente pressiona Enter, podendo deixar caracteres indesejados no buffer de entrada. Já o fgets lê a linha inteira, facilitando a validação precisa do que o cliente realmente digitou, como detectar uma entrada vazia ou não numérica de forma explícita, tornando o tratamento de erros mais robusto.

Alguns outros pormenores também foram solucionados, como quando o servidor terminava a sessão e o cliente continuava conectado, ocasionando uma série de erros, e vice-versa. Inicialmente, o servidor também encerrava quando o cliente saía, mas agora ele continua ligado, esperando um novo cliente, o que foi resolvido com um loop a mais no código.

4. Conclusão:

Com o trabalho, foi possível observar o funcionamento de uma comunicação em rede utilizando o protocolo TCP/IP. O jogo utilizou tanto o servidor para armazenar e validar as informações quanto o cliente para uma interface de comunicação com o usuário (inputs por meio do CMD).

É interessante observar na prática os conceitos estudados em sala de aula, como o protocolo para a conexão de duas pontas, o fechamento de conexão, e os padrões da internet para endereçamento de máquinas. O fluxo temporal de envio e recebimento de mensagens também pôde ser exemplificado com a comunicação entre o cliente e o servidor, com um protocolo que foi construído a partir das orientações do enunciado.

Neste trabalho, o servidor aceitava a conexão de apenas um cliente, e conseguia se comunicar com eficácia com este. Em trabalhos futuros, será interessante ver como é o funcionamento de um servidor com múltiplas conexões.