Trabalho Prático 1 - Redes de computadores

Maria Luiza Leão Silva - 2020100953
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
marialuizaleao@dcc.ufmg.br

1. Introdução

Este documento aborda o desenvolvimento de uma versão de Campo Minado que permite a interação entre dois computadores remotos. O objetivo principal deste projeto é criar tanto um servidor quanto um cliente que viabilizem a jogabilidade do Campo Minado à distância. No servidor, reside a responsabilidade de manter e atualizar o estado atual do jogo, enquanto o cliente tem a função de enviar as coordenadas do campo escolhido e receber as informações atualizadas do servidor sobre o estado do jogo. Este projeto proporcionou uma oportunidade para aplicar conhecimentos de programação em um cenário de rede, contribuindo para uma compreensão mais sólida dos conceitos envolvidos.

Na próxima seção, vamos adentrar na parte de modelagem computacional do problema, onde examinaremos em detalhes como o tabuleiro do jogo é estruturado e como as ações dos jogadores são comunicadas entre o cliente e o servidor. Em seguida, na seção 3, vamos explorar os obstáculos que surgiram durante o processo, seguidos por um guia que explica como compilar e executar o programa.

2. Modelagem computacional do problema

A modelagem computacional do problema envolve a definição dos principais componentes e requisitos do sistema Campo Minado remoto. Nesta seção, exploraremos os aspectos fundamentais do projeto, incluindo os atores envolvidos, as regras do jogo, as estruturas de dados e as funcionalidades essenciais.

2.1 Server/Client e sua comunicação:

2.1.1. Servidor(server.h e server.c): O servidor é responsável por manter e gerenciar o estado atual do jogo de Campo Minado. Ele recebe as ações dos clientes, atualiza o tabuleiro conforme as jogadas e envia as informações atualizadas de volta aos clientes. Além disso, o servidor controla o início e o término de partidas e gerencia a conexão de clientes.

2.1.2. Cliente(client.h e client.c): Os clientes são os jogadores que se conectam ao servidor para jogar Campo Minado remotamente. Eles interagem com o servidor, enviando coordenadas dos campos que desejam revelar ou marcar com bandeiras. Os clientes recebem informações sobre o estado atual do jogo do servidor e exibem o tabuleiro.

Em primeiro lugar, foi necessário estabelecer um `struct sockaddr_storage` para ambos `client` e `server` para armazenar o endereço do socket. A escolha dessa estrutura foi motivada pela flexibilidade e portabilidade que ela fornece para lidar com diferentes tipos de endereços(IPv4 e IPv6) em um único contexto de código. Para isso, foram desenvolvidas as funcões `'clientSockaddrInit'` e `'serverSockaddrInit'` que inicializam uma estrutura `sockaddr_storage` com os valores apropriados para um protocolo de IP e número de porta fornecidos. No entanto, `clientSockaddrInit` é usado para configurar o endereço IP do servidor remoto ao qual o cliente se conectará, enquanto `serverSockaddrInit` é usado para configurar o socket do servidor para escutar em um endereço IP e porta específicos, dependendo do protocolo de IP escolhido.

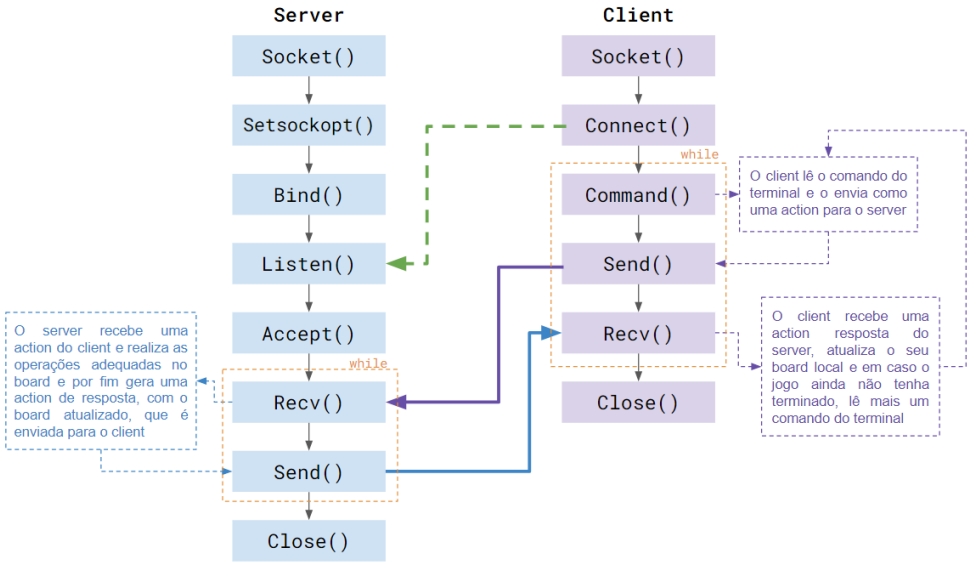

Figura 1: fluxo geral do server e client com base nas funções do socket

1. **`socket()`:** usada para criar um novo socket. Os sockets usados são do tipo SOCK_STREAM (para comunicação TCP) tanto para o `server` como para o `client` que usam essa função para criar seus próprios sockets.
2. **`bind()`:** usada para associar um endereço IP e uma porta a um socket. Isso permite que o servidor escute em uma porta específica e aguarde conexões de clientes nessa porta. A porta que será usada é passada pela linha de comando para o `server`. Essa mesma porta e IP também são passados para o client, que os usará ao chamar o `connect()`.
3. **`listen()`:** usada para colocar o socket em um estado passivo, pronto para aceitar conexões de clients. O argumento backlog especifica o número máximo de conexões pendentes na fila.
4. **`accept()`:** usada no lado do server para aceitar uma conexão de um `client`. Ela cria um novo socket conectado que pode ser usado para trocar dados com o `client`.
5. **`connect()`:** usada para estabelecer uma conexão com o `server`. O `client` deve especificar o endereço IP e a porta do servidor para se conectar. Esses parâmetros são passados a ele pela linha de comando.
6. **`send()` e `recv()`:** são usadas para enviar e receber dados pela conexão estabelecida. O `server` e o `client` podem trocar informações usando essas funções.
7. **`close()`:** usada para encerrar a conexão de um socket após a conclusão da comunicação. Tanto o `server` quanto o `client` fecham os sockets após o uso para liberar recursos.

O `client` tem sua principal função executada dentro de um loop `while`, mostrado de amarelo na figura 1. Seu fluxo de execução pode ser observado na figura 2.

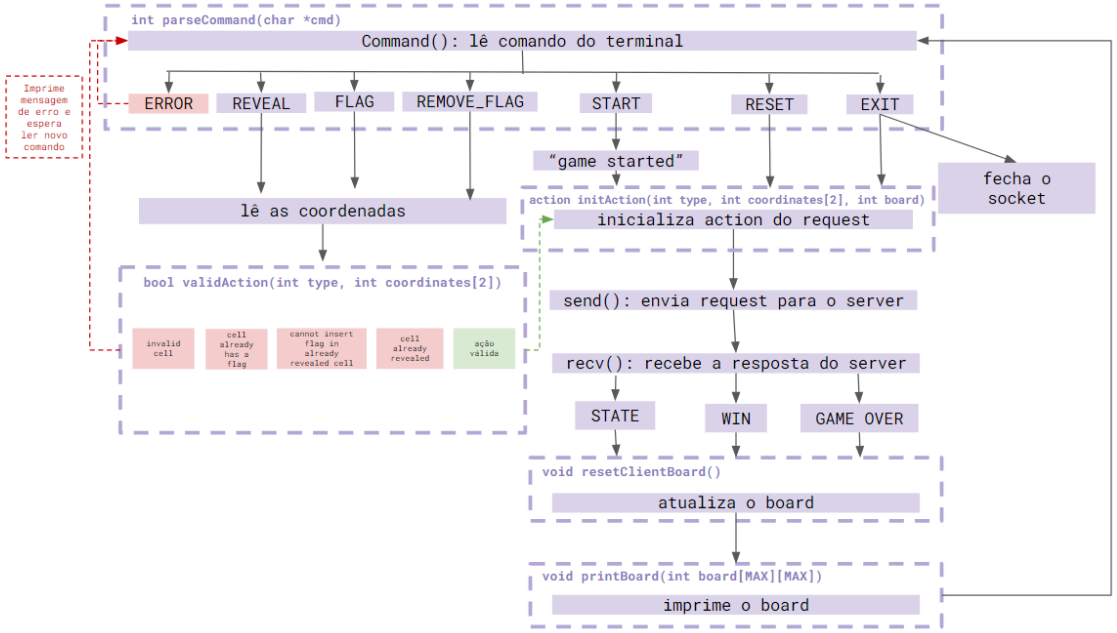

Figura 2: fluxo de execução do loop while do client mostrado na figura 1

O `server` também tem sua principal função executada dentro de um loop `while`. Seu fluxo de execução pode ser observado na figura 2.

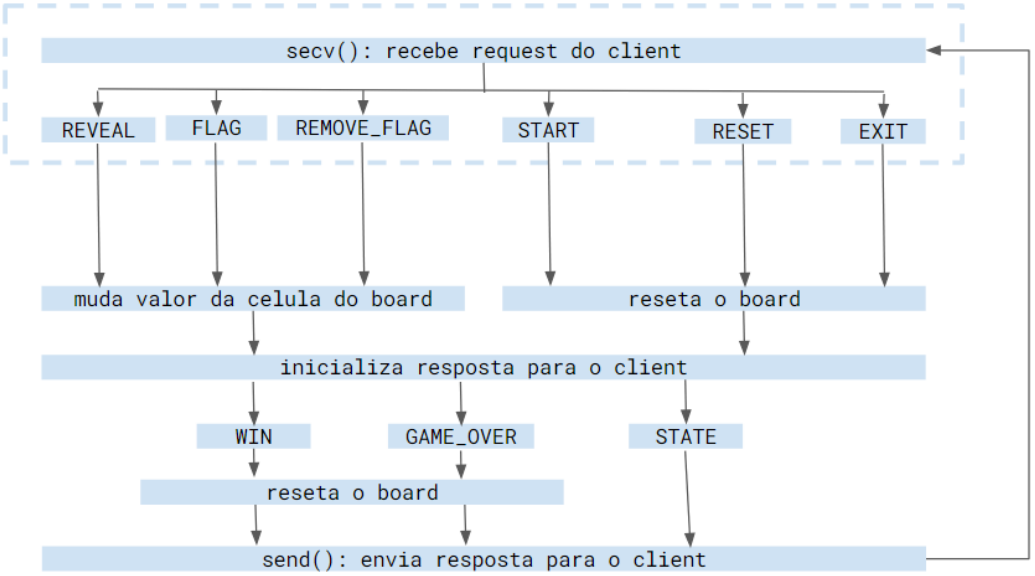

Figura 3: fluxo de execução do loop while do server mostrado na figura 1

2.2 Representação do tabuleiro e das ações dos jogadores:

O tabuleiro foi representado como uma matriz de inteiros e, para melhorar a legibilidade do código, foram usadas macros e constantes para representar os possíveis estados das células do tabuleiro e as diferentes ações, `struct action`, passadas entre o server e o cliente, junto com uma função `actionInit` para iniciar os valores dessa estrutura.

O `server` possui dois tabuleiros, o primeiro, `answerBoard`, é o tabuleiro lido do arquivo de entrada e é utilizado como um gabarito. Quando o `client` pede um `REVEAL x,y`, esse será o tabuleiro consultado para descobrir o valor da célula revelada. Esse tabuleiro não é alterado durante a execução do programa, uma vez que ele é lido do arquivo, ele se mantém sempre o mesmo. Um segundo board, `clientBoard`, é usado para guardar qual o estado do tabuleiro do `client`. Ao longo da execução, essa é a variável que será alterada. Ele começa com todas suas células como `HIDDEN` e, ao longo do jogo, pode ter seus valores revelados ou marcados como `FLAG`, até que todas as células que não são bombas sejam reveladas(`WIN`) ou até que uma bomba seja revelada(`GAME_OVER`). Para facilitar a verificação de vitória, foi criada uma variável `amountOfNotBombsCells` para guardar quantas células ainda precisam ser reveladas e assim evitar que todo o tabuleiro tenha que ser percorrido na hora de verificar o estado do jogo após cada pedido de action do `client` recebido pelo `server`. A cada `REVEAL x,y`, se o valor revelado é uma bomba, uma resposta `GAME_OVER` será enviada para o `client`, junto com o `answerBoard`. Caso contrário, `amountOfNotBombsCells` é decrementada e, se seu valor chegar a 0, temos que uma resposta `WIN` deve ser enviada ao `client` junto com o `answerBoard`. Se não, atualizamos somente o valor da célula revelada, `clientBoard[x][y] = answerBoard[x][y]`, e enviamos uma resposta `STATE` para o `client` junto com o tabuleiro atualizado.

Já o `client` possui somente um tabuleiro para guardar o tabuleiro da resposta do `server` ao seu pedido. Ao final de um `recv()` seu tabuleiro é atualizado, `board = responseFromServer.board`(ação realizada pela função `copyBoard(int board[][])`), e seu valor é impresso no terminal do `client`, `printBoard(int board[][])`.

3. Desafios

Durante minha jornada de implementação do jogo, me deparei com vários desafios e obstáculos que precisei superar. Primeiramente, uma das dificuldades iniciais foi a definição clara das responsabilidades entre o cliente e o servidor. Inicialmente, não tinha um entendimento completo sobre como o estado do jogo deveria ser mantido, como as informações deveriam fluir entre as partes e como os comandos deveriam ser tratados em cada lado. No entanto, por meio de uma análise das especificações do projeto e exemplos disponíveis, consegui estabelecer o fluxo de dados e as ações esperadas de cada componente.

Outro desafio significativo foi lidar com a configuração e o gerenciamento dos sockets para estabelecer a comunicação entre o cliente e o servidor. Foi necessário aprender a configurar detalhes como endereços IP, portas, tipo de conexão e versão do endereçamento. Para superar essa complexidade, dediquei tempo ao estudo da biblioteca de sockets em C, consultando os recursos disponíveis, a fim de garantir a configuração correta.

Uma questão importante foi a complexidade de tempo e espaço, especialmente ao considerar a possibilidade de expansão para tabuleiros maiores. Garantir um desempenho eficiente e um consumo de recursos controlado exigiu a implementação de algoritmos otimizados e estruturas de dados eficazes, independentemente do tamanho do tabuleiro utilizado.

Outra dificuldade notável durante o projeto foi a necessidade de lidar com todos os casos de borda no código do cliente e do servidor, garantindo que o jogo funcionasse sem erros mesmo em situações excepcionais. Isso exigiu uma atenção aos detalhes e a implementação de tratamentos de erro, ao mesmo tempo em que se buscava manter a clareza e a legibilidade do código. Encontrar o equilíbrio entre a gestão de casos extremos e a simplicidade da implementação foi um desafio constante, mas fundamental para a entrega de um jogo funcional com um código enxuto.

Além disso, durante a fase final da implementação, enfrentei o desafio de garantir que meu código se mantivesse claro e organizado. Isso envolveu a separação do código em vários arquivos para modularizar as funcionalidades, a redução do tamanho da função main() e a revisão e refatoração constante do programa. Essas medidas foram essenciais para melhorar a legibilidade, facilitar a manutenção e promover boas práticas de programação, tornando o código mais compreensível. Finalmente, após a conclusão do código, revisei a documentação da biblioteca de sockets para garantir que diferentes implementações do jogo pudessem interoperar adequadamente e que as conexões funcionassem conforme o esperado.

4. Instruções para compilação e execução

1 – Extraia o arquivo .zip na pasta desejada.
2 – Execute o comando ‘make’ no terminal.
3 – Execute os programas client e server da pasta bin passando um arquivo de texto com a entrada para o servidor e os argumentos necessários pela linha de comando.

- Funcionamento do cliente: ./bin/client <endereço ip> <porta>
- Funcionamento do servidor: ./bin/server <v4|v6> <porta> -i <caminho do arquivo de entrada>