Henrique Assumpção
Matrícula: 2020006620
henriquesoares@dcc.ufmg.br

TP1: Relatório
Redes de Computadores
Universidade Federal de Minas Gerais

2023-10-08

1 Introdução

O primeiro trabalho prático da disciplina de redes consiste na implementação de um jogo de campo minado entre em cliente e um servidor, utilizando a biblioteca de sockets para sistemas Unix.
O conexão entre cliente e servidor deve ser feita por meio do protocolo TCP, utilizando endereços do protocolo IP, tanto a versão 4 quanto a versão 6. O cliente deve ser capaz de ler entradas da entrada padrão, enviá-las para o servidor, receber uma resposta e então repetir até que o jogo termine ou até que o cliente deseje parar. O servidor por sua vez é encarregado de todo o processamento do jogo, e ele deve ser capaz de receber uma mensagem do cliente, processar a ação desejada, alterar o estado do jogo e então enviar tal estado ao cliente, até que o servidor detecte que o cliente ganhou, perdeu, ou quis se desconectar.

2 Modelagem

O programa foi desenvolvido na linguagem C, no sistema operacional Windows 11, utilizando o sistema WSL de Linux. O programa foi implementado e compilado em uma máquina com 16GB de RAM em um processador Intel Core I5-9400F.
Nesta seção, iremos fornecer uma breve descrição da solução adotada para implementar o trabalho. Os detalhes de implementação podem ser encontrados nos comentários feitos no código fonte, portanto iremos prover apenas uma discussão em alto nível.

2.1 Cliente

A implementação do cliente se deu de forma relativamente simples, de acordo com os seguintes passos:

1. Primeiro o cliente recebe os parâmetros da linha de comando, e então realiza o processamento necessário do endereço IP fornecido e do porto que será utilizado para a conexão. A função PARSE_CLT_ADDR é a que realiza todo o processamento dos parâmetros, isto é, ela identifica se o endereço é um IPv4 ou um IPv6, e então prossegue para inicializar as estruturas ADDR necessárias.

2. Após processar os parâmetros, o cliente inicializa seu socket e se conecta ao endereço do servidor, dando início a conexão.

3. O cliente então entra em um loop WHILE(1), e chama a função PARSE_INPUT que recebe a entrada do stdin. Essa função também checa se a entrada é válida, e caso não seja imprime as respectivas mensagens de erros descritas no enunciado, i.e., o tratamento de erros na entrada é feita pelo próprio cliente. Caso o cliente leia uma entrada inválida, ele ignora a mesma e passa para a próxima iteração, ou seja, somente entradas válidas são enviadas ao servidor.

4. O cliente então envia a mensagem para o servidor utilizando a função SEND_ACTION, que serializa os dados da estrutura ACTION descrita no enunciado. Após isso o cliente checa se recebeu um EXIT como entrada no stdin, e caso positivo saí do loop, fecha o socket e encerra o programa.

5. O cliente recebe a ação de volta do servidor, e então checa se venceu ou perdeu, caso positivo imprime as respectivas mensagens descritas no enunciado e então espera uma nova entrada do usuário, que deve ser um comando START ou RESET tendo em vista que o jogo anterior se encerrou.

2.2 Servidor

A implementação do servidor é o aspecto mais complexo do trabalho, e sua implementação foi de acordo com os seguintes passos:

1. Primeiro o servidor recebe os parâmetros da linha de comando, e então realiza o processamento do indicador da versão do protocolo IP recebido e do porto que será utilizado. A função PARSE_SERV_ADDR é utilizada para determinar a versão do protocolo IP que será utilizada, e também para inicializar as estruturas ADDR necessárias. O servidor então inicializa seu socket, realiza a operação de bind do socket com o endereço e então realiza a operação de listen, com um backlog de tamanho um.

2. Após isso, o servidor lê o tabuleiro do jogo do arquivo fornecido na linha de comando, e o armazena em uma matriz de inteiros.

3. O servidor então entra em um loop WHILE(1). Primeiramente ele aceita uma conexão do cliente a partir da operação de ACCEPT, e então ele inicializa o estado do cliente como uma matriz com todas as entradas igual a -2 (indicador de célula oculta).

4. Após isso, ele entra em outro loop WHILE(1). Ele primeiro recebe uma ação do cliente, checa para caso tal ação seja um EXIT, no caso positivo ele sai do loop interno e fecha a conexão com o cliente, e caso negativo continua. Após isso ele processa a ação com a função PROCESS_ACTION, que atualiza o estado do cliente de acordo com a ação recebida e as regras determinadas no enunciado. Finalmente, ele envia a ação processada ao cliente, e então checa para o caso do cliente ter vencido ou perdido, e caso positivo ele reinicia o tabuleiro do cliente e espera por uma nova ação, que deve ser do tipo START ou RESET. Note que o loop externo do servidor é eterno, ou seja, ele só para caso o programa seja manualmente interrompido.

2.3 Organização do Código

O código foi organizado da seguinte maneira:

- A pasta /SRC/ contém os arquivos CLIENT.C,SERVER.C,COMMON.C,CLT_UTILS.C,SERV_UTILS.C que implementam o cliente, o servidor, as funções comuns, as funções exclusivas do cliente, e as funções exclusivas do servidor, respectivamente.

- A pasta /INCLUDE/ contém os arquivos ACTION.H,CONSTANTS.H,COMMON.H,CLT_UTILS.H SERV_UTILS.H. Os arquivos ACTION.H,CONSTANTS.H contém a estrutura de dados action descrita no enunciado, e a definição das diversas constantes utilizadas ao longo do programa, respectivamente. Os demais arquivos são os cabeçalhos dos arquivos descritos no último item.

- A pasta /OBJ/ contém os arquivos .o obtidos após a execução do comando MAKE.

- A pasta /INPUT/ contém arquivos .txt com a entrada do tabuleiro do servidor.

- A pasta /BIN/ contém os arquivos binários do cliente e do servidor obtidos após a execução do comando MAKE.

- No diretório raiz, encontra-se o arquivo MAKEFILE que compila os demais arquivos e gera os binários.

3 Desafios e Dificuldades

Nesta seção, iremos discutir os principais desafios e dificuldades encontrados ao longo do desenvolvimento do trabalho, e também explicitar as soluções propostas para superar tais desafios.

3.1 Compatibilidade com IPv4 e IPv6

Um dos primeiros desafios encontrados na implementação do trabalho foi o de **como receber endereços que podem ser tanto em formato IPv4 quanto IPv6**? Essa diferença configura um desafio pois os formatos são altamente diferentes, e portanto o parsing das entradas deve ser diferente. Resolver tal problema para o servidor foi trivial, pois o servidor utiliza indicadores "v4" e "v6" para identificar a versão, no entanto, o cliente apenas informa o endereço. Para resolver isso, utilizamos a função INET_PTON com os tipos de interface AF_INET e AF_INET6, que nos permitem tentar converter uma string para um endereço IP de algumas das versões, e caso não seja possível a função nos avisa, e portanto assim podemos detectar de maneira dinâmica o tipo de endereço passado pelo cliente.

3.2 Envio e Recebimento de Ações

O enunciado requer que as mensagens trafegadas sejam de acordo com a estrutura action definida, isto é, um inteiro representando o tipo da ação, um array com duas posições representando as coordenadas, e uma matriz 4×4 representando o estado do cliente. Um desafio natural que segue de tal definição é a forma como essa estrutura será representada em bytes para que possa ser enviada pela rede através do protocolo TCP.
A solução para o problema foi implementada da seguinte forma:

- SEND: Para enviarmos os dados pela rede e garantirmos a compatibilidade com outros clientes, primeiro copiamos o conteúdo da ação atual para outro objeto ação que irá atuar como um buffer, e então realizamos uma chamada a função auxiliar SEND_LOOP, que realiza operações de send até que todo o buffer tenha sido enviado pela rede. Tal operação de loop é necessária dentro do contexto do protocolo TCP: uma operação única de send não tem garantia que irá enviar todos os bytes demandados, portanto devemos realizar sucessivos sends até que todos os bytes do objeto action sejam corretamente enviados. Não é feita uma checagem de se a mensagem foi enviada corretamente, deixamos isso para quem recebe.

- RECEIVE: Para recebermos os dados pela rede, inicializamos um objeto de tipo action que irá atuar como um buffer e então chamamos a função auxiliar RECV_LOOP, que lê bytes do socket até que o tamanho do buffer seja atingido ou até que um erro seja detectado. Essa operação é essencial no contexto do TCP: assim como no send, não há garantia que todos os bytes irão chegar em um único receive, portanto devemos realizar sucessivos receives até que o buffer esteja cheio, isto é, até que todos os bytes da ação tenham sido lidos. Se ocorreu algum erro, armazenamos o valor constante ERROR_TYPE_ID – que é definido no arquivo CONSTANTS.H – no atributo type da ação, e deixamos que o cliente ou servidor lide com tal problema. Caso não tenha erro, copiamos os valores do buffer de tipo action para a ação passada como parâmetro.

3.3 Detecção de erros

Tanto o cliente quanto o servidor devem ser capazes de detectar erros. A detecção por parte do cliente é trivial: basta checar se a entrada fornecida pelo stdin é compatível com as regras do enunciado. Caso contrário, armazenamos o valor ERROR_TYPE_ID no atributo type da ação do cliente. No loop while, o cliente checa se após o parsing da entrada ele recebeu um ERROR_TYPE_ID, e caso

positivo ele imprime a respectiva mensagem de erro e pula para a próxima iteração, pois de acordo com o enunciado o cliente deve ser capaz de continuar jogando caso receba algum comando inválido de entrada. Após receber a mensagem do servidor, o cliente também checa se ocorreu erro na transmissão, e no caso positivo encerra a conexão com o servidor. Esse caso é altamente improvável, mas como ele é possível e não foi abrangido no enunciado, optamos pela solução discutida baseado no seguinte princípio: um erro de transmissão pode significar um erro no código do servidor ou uma instabilidade na rede, e em ambos os casos é melhor cessar a comunicação.
A princípio, um servidor que assume que o cliente implementa tal processo de detecção de erros não precisaria se preocupar com ações inválidas, no entanto, **optamos por desenvolver um servidor que não assume o cliente segue essa regra**. O servidor checa por erros em duas etapas: no processo de recebimento de mensagem, e após processar a ação recebida. Caso a ação do cliente não seja uma das possíveis, ele armazena o valor ERROR_TYPE_ID no atributo type da ação e no loop interno checa por possíveis erros. Nesse caso, existem dois cenários possíveis de erro: o cliente é defeituoso e envia uma ação inválida, ou ocorreu um erro na transmissão da mensagem pela rede. Em ambos os casos, como não foi especificado no enunciado o que deve ser feito, optamos por encerrar a conexão com o cliente. Essa escolha é baseada no seguinte princípio: se o cliente é defeituoso, é melhor não comunicar com ele, e se a ocorreu algum erro de transmissão, pode ser tanto porque a rede não é confiável ou porque ocorreu algum erro no cliente, e em ambos os casos é melhor cessar a comunicação.
Um exemplo de uma situação de erro pode ser o seguinte: o servidor aceita a conexão do cliente, mas antes de enviar qualquer mensagem o programa do cliente encerra abruptamente. Nesse caso, o servidor iria ler 0 bytes do socket, e caso ele prosseguisse de forma normal, o processamento da ação poderia levar a algum erro inesperado.
O comportamento do servidor e do cliente quanto a erros pode ser sumarizado da seguinte forma:

- **Cliente**: No loop while, faz o parse da entrada, e caso a entrada seja inválida ele a ignora e passa para a próxima iteração. Envia a ação para o servidor, e depois recebe de volta a ação atualizada. Se ocorreu algum erro de transmissão, encerra a conexão com o servidor.

- **Servidor** Aceita a conexão do cliente e entra no loop while interno. Recebe a ação do cliente, e caso ocorra erro de transmissão, sai do loop interno e encerra a conexão com o cliente. Processa a ação, e caso ela seja inválida, i.e., o cliente é defeituoso, também sai do loop interno e encerra a conexão com o cliente.

4 Conclusão

O primeiro trabalho prático da disciplina foi uma experiência desafiadora, mas no entanto extremamente gratificante. A experiência prática adquirida ao longo do trabalho agrega ao conhecimento fornecido pelo professor em sala de aula, fazendo com que o aprendizado seja mais completo. De modo geral, acreditamos que fornecemos uma implementação completa dentro do contexto do enunciado, que não só executa as ações esperadas mas também lida com alguns cenários de erro de forma criativa e consistente com o trabalho.