

Documentação do Código: Servidor e Cliente de Labirinto

Introdução e Problema Resolvido

Este projeto implementa um jogo de exploração de labirintos utilizando comunicação via sockets em C. A solução envolve um servidor que mantém o estado do jogo e um cliente que interage com o usuário, permitindo que ele se mova e explore o labirinto até encontrar a saída. O objetivo principal é testar habilidades de programação em rede, comunicação cliente-servidor, e manipulação de estruturas complexas.

No jogo, o cliente se conecta ao servidor, envia comandos para se mover e o servidor responde com as informações de movimento válidas e atualizações sobre o estado do jogo. O servidor mantém o mapa completo do labirinto, enquanto o cliente só pode ver áreas já exploradas.

Como Rodar o Projeto

Este projeto utiliza um `Makefile` para compilar os arquivos do servidor e do cliente. Para facilitar o uso, os binários são criados dentro de uma pasta chamada `bin` na raiz do projeto. Siga as etapas abaixo para compilar e rodar o projeto:

1. **Compilar o Projeto**

- Execute o comando `make` na raiz do projeto para compilar o servidor e o cliente. Isso criará os executáveis `server` e `client` na pasta `bin`.

- Comando:

```
...
```

```
make
```

```
...
```

2. **Executar o Servidor**

Documentação do Código: Servidor e Cliente de Labirinto

- Para iniciar o servidor, utilize o comando abaixo. O servidor requer que você especifique a versão do IP (`v4` ou `v6`), a porta de conexão, e o caminho para o arquivo de entrada que contém o labirinto.

- Comando:

...

```
./bin/server v4 51511 -i input/in.txt
```

...

- Isso iniciará o servidor que ficará esperando por conexões na porta especificada.

3. **Executar o Cliente**

- Para conectar o cliente ao servidor, utilize o comando abaixo, especificando o endereço IP do servidor e o número da porta.

- Comando:

...

```
./bin/client 127.0.0.1 51511
```

...

- Uma vez conectado, o cliente poderá enviar comandos para explorar o labirinto, como `start`, `move`, `map`, `reset`, entre outros.

Funções do Servidor (`server.c`)

1. `main`: Configura o servidor e aceita conexões de clientes. Inicializa o estado do jogo e chama a função `handle_client` para lidar com a comunicação.

2. `read_matrix_from_file`: Lê a matriz do labirinto a partir de um arquivo de entrada. Essa matriz representa o labirinto e inclui a posição inicial do jogador e a posição de saída.

3. `initialize_game`: Inicializa o estado do jogo. Configura o labirinto, define o jogador na posição inicial e marca

Documentação do Código: Servidor e Cliente de Labirinto

todas as posições não descobertas como zero.

4. `set_matrix_descoberto_to_zeros`: Reseta a matriz de "descobertas" para zeros, indicando que nenhuma posição foi explorada.

5. `mark_positions_around_player`: Marca posições ao redor do jogador como descobertas para simular a "visão" do jogador.

6. `handle_client`: Lida com a comunicação do cliente. Recebe comandos e processa de acordo com o tipo da ação.

7. `reset_game`: Restaura o estado do jogo para os valores iniciais, permitindo que o jogador recomece o labirinto.

8. `process_action`: Processa a ação recebida do cliente, seja de início (`START`), movimento (`MOVE`), visualização do mapa (`MAP`), reset (`RESET`), ou outras. Atualiza o estado do jogo e envia as informações necessárias de volta ao cliente.

9. `send_action`: Serializa e envia a estrutura de ação (`action`) ao cliente via socket.

10. `serialize_action` e `deserialize_action`: Convertendo os valores da estrutura `action` para a ordem de bytes de rede (big-endian) e vice-versa, garantindo que a comunicação seja compreendida pelo cliente.

11. `move_player`: Move o jogador na direção indicada, se for válida. Atualiza a posição no labirinto e determina se o jogo foi vencido.

12. `calculate_possible_moves`: Calcula e retorna os movimentos possíveis do jogador, considerando as paredes e os limites do labirinto.

13. `copy_board_to_action`: Copia o estado do tabuleiro para a estrutura de ação que será enviada ao cliente.

14. `fill_unreachable_positions`: Marca posições que não estão no campo de visão do jogador, representando-as como "não descobertas".

15. `init_game_state`: Inicializa a estrutura `GameState`, definindo todos os valores para iniciar o jogo.

16. `fill_possible_moves`: Preenche a estrutura de ação com os movimentos possíveis a partir da posição atual do jogador.

17. `build_error`: Constrói uma mensagem de erro para ser enviada ao cliente quando uma ação inválida é solicitada.

Documentação do Código: Servidor e Cliente de Labirinto

18. `handle_start`: Inicia o jogo, inicializando o estado com a matriz de entrada. Envia uma resposta de atualização ao cliente com os movimentos possíveis.
19. `handle_move`: Executa o movimento do jogador conforme a direção especificada pelo cliente. Atualiza o estado do jogo e envia uma resposta de atualização ou uma mensagem de vitória se o jogador alcançar a saída.
20. `handle_map`: Envia ao cliente o mapa parcial, marcando as posições descobertas pelo jogador e aquelas que estão fora do alcance como "não visíveis".
21. `handle_reset`: Reinicia o jogo para os valores iniciais e envia uma confirmação ao cliente com o tipo `UPDATE`.
22. `handle_exit`: Lida com o pedido de desconexão do cliente, enviando uma mensagem de atualização e fechando a conexão.
23. `handle_default`: Envia uma mensagem de erro ao cliente para comandos inválidos.
24. `handle_game_not_initialized`: Envia uma mensagem de erro indicando que o jogo ainda não foi iniciado e que o cliente deve usar o comando `START` primeiro.
25. `handle_game_over`: Lida com ações recebidas após o jogo ter terminado. Permite ao cliente resetar ou sair, enquanto outros comandos são ignorados.

Funções do Cliente (`client.c`)

1. `main`: Conecta ao servidor usando sockets e gerencia a interação do usuário, enviando comandos como `START`, `MOVE`, `MAP`, etc. Lida com a entrada do usuário e processa a resposta do servidor.
2. `send_action` e `receive_action`: Envia a estrutura `action` ao servidor e recebe respostas. Utiliza a serialização para garantir a compatibilidade dos dados.
3. `serialize_action` e `deserialize_action`: Convertendo valores de rede para ordem local e vice-versa para assegurar a correta comunicação entre cliente e servidor.
4. `handle_move`: Lida com a resposta de movimento do servidor, imprimindo os movimentos possíveis.

Documentação do Código: Servidor e Cliente de Labirinto

5. `handle_reset`: Lida com o comando de reset enviado ao servidor, imprimindo os movimentos possíveis após o reset.
6. `handle_error`: Imprime mensagens de erro recebidas do servidor.
7. `handle_start`: Inicializa o jogo e imprime os movimentos possíveis que o jogador pode fazer.
8. `print_possible_moves`: Imprime os movimentos possíveis a partir da posição atual do jogador.
9. `handle_map`: Exibe o mapa parcial recebido do servidor, mostrando apenas as posições descobertas pelo jogador.
10. `encontrar_dimensoes`: Encontra as dimensões reais do labirinto, considerando posições válidas, para uma melhor visualização no terminal.
11. `print_board`: Imprime o estado atual do tabuleiro no terminal, utilizando caracteres para melhor visualização das posições (muro, caminho livre, jogador, etc.).

Conclusão

Este projeto envolve a implementação de um servidor e um cliente que se comunicam via sockets para explorar um labirinto. O servidor é responsável por manter o estado completo do labirinto, enquanto o cliente apenas exibe a parte já descoberta. Este projeto testa habilidades de rede e comunicação em C, incluindo o uso de sockets e gerenciamento de estruturas complexas de dados.

A implementação de funções auxiliares, como a serialização/deserialização e o cálculo de movimentos possíveis, garante a continuidade e a integridade da comunicação entre cliente e servidor. Além disso, o gerenciamento dos estados do jogo permite que o usuário explore o labirinto, visualize o mapa, receba dicas, e até mesmo reinicie ou saia do jogo conforme desejado, mantendo uma experiência de jogo fluida e interativa.