UNIVERSIDADE FEDERAL
DE MINAS GERAIS

CARLOS HENRIQUE BRITO MALTA LEÃO
MATRÍCULA: 2021039794

TRABALHO PRÁTICO 01

Redes de Computadores

Belo Horizonte
2024

INTRODUÇÃO

Este trabalho prático visa explorar a comunicação entre clientes e servidores em redes de computadores, utilizando a linguagem de programação C. Inspirado pelo funcionamento do aplicativo Uber, uma plataforma que conecta usuários a motoristas parceiros para serviços de transporte, foi desenvolvida uma aplicação console que simula esse processo básico. Por meio dessa simulação, é objetivado não apenas exercitar, mas também compreender os fundamentos práticos da comunicação cliente-servidor usando conexão TCP.

Além de exercitar a implementação de sistemas distribuídos em C, os objetivos específicos deste projeto incluem familiarizar-se com o uso de sockets para comunicação entre processos, simular um sistema de solicitação e resposta em tempo real e explorar os detalhes da conexão TCP para garantir confiabilidade e ordenação na troca de mensagens.

Descrição do Projeto

A aplicação console replica o fluxo básico de interação entre usuários e motoristas, como visto no aplicativo Uber. Os clientes desempenham o papel dos usuários que solicitam serviços de transporte, enquanto os servidores atuam como os motoristas que atendem essas solicitações. A interação é estabelecida por meio de uma comunicação cliente-servidor baseada em TCP, permitindo uma troca confiável e ordenada de mensagens. Além disso, a aplicação aceita conexões IPV4 ou IPV6, basta informar durante a execução.

A implementação da comunicação cliente-servidor neste projeto baseia-se no uso de sockets. Os sockets são mecanismos que permitem a comunicação entre processos, sejam eles locais ou remotos. No contexto deste projeto, eles são essenciais para estabelecer a conexão entre o cliente e o servidor, facilitando a troca de mensagens em tempo real.

O uso de sockets com conexão TCP proporciona confiabilidade e garantia de entrega ordenada de dados. Enquanto o TCP oferece uma abordagem orientada à conexão, garantindo a integridade e a sequência das mensagens, o UDP, por outro lado, poderia ser mais adequado para aplicações que exigem uma comunicação mais leve e não confiável.

Nesse âmbito, o processo de comunicação entre cliente e servidor segue um fluxo de comunicação bem definido. Primeiramente, o servidor realiza uma abertura passiva, em que identifica-se e fica aberto a receber conexões. Em seguida, é possível que o cliente realize uma abertura ativa, em que o processo cliente sabe alcançar o processo servidor. Com isso, se o servidor aceitar a conexão com o cliente, ocorre a abertura completa, possibilitando a comunicação entre os processos. Por fim, a conexão é finalizada por ambos os processos.

DESCRIÇÃO DO CÓDIGO

O código fonte desse projeto foi desenvolvido na linguagem de programação C e utiliza bibliotecas padrões da linguagem compatíveis com o sistema operacional Linux. Os códigos do Servidor e do Cliente estão implementados em arquivos separados, respectivamente *server.c* e *client.c*. Também foram criados os arquivos *utils.h* e *utils.c* para implementar algumas funções e constantes comuns tanto para o servidor quanto para o cliente. O projeto também apresenta um arquivo Makefile, que gerencia a compilação dos arquivos fonte e a geração dos executáveis para o servidor e o cliente. Para compilar o programa, basta acessar a pasta raiz do projeto e executar o comando *make*.

Utils

O arquivo utils.c contém funções utilitárias essenciais para o funcionamento do projeto e comuns entre o Servidor e o Cliente.

A função *exitWithMessage* é responsável por imprimir mensagens de erro, junto com detalhes opcionais, e encerrar o programa. Ela recebe dois parâmetros: msg, que é a mensagem de erro principal, e detail, que são detalhes adicionais sobre o erro, se fornecidos.

A função *getServerAddressStructure* cria e retorna uma estrutura de endereço para o servidor, dependendo do tipo de IP especificado (IPv4 ou IPv6). Ela recebe dois parâmetros: *ipType*, que indica o tipo de endereço IP (IPv4 ou IPv6), e *serverPort*, que é a porta do servidor. Retorna uma estrutura de endereço para o servidor.

Servidor

O servidor deve receber alguns parâmetros para realizar suas configurações de conexão. Para isso, durante a execução, é preciso informar dois parâmetros. Primeiramente, é necessário especificar o tipo de IP que o servidor utilizará. O segundo parâmetro deverá ser a porta utilizada. Seguem exemplos de comandos de execução:

- ./server ipv4 50501
- ./server ipv6 50501

Com os parâmetros informados, o cliente pode definir o tipo de IP a ser utilizado e também a porta do servidor. Isso é possível graças à função utilitária *getServerAddressStructure*, que cria a estrutura de endereço para o servidor, dependendo do tipo de IP especificado. Essa função configura a família de endereços (IPv4 ou IPv6), a interface de entrada e a porta local.

Com essa estrutura definida, é possível criar o socket do servidor utilizando a função socket. Essa função cria um ponto final de comunicação e recebe três parâmetros: a família

de protocolos (já definida anteriormente), o tipo de comunicação (no nosso caso, SOCK_STREAM para TCP) e o protocolo (TCP).

Em seguida, com o socket criado, é possível dar continuidade no fluxo de comunicação, associando o socket do servidor ao endereço IP e porta. Para isso, utiliza-se a função bind recebendo como parâmetro a referência ao socket previamente criado, a especificação do endereço a ser associado (a estrutura de endereço criada anteriormente) e o tamanho da estrutura apontada pelo endereço, em bytes.

Após essa associação, é iniciado o processo de abertura passiva, em que, primeiramente, é preciso configurar o socket do servidor para aguardar as conexões de entrada. Para isso, é utilizada a função listen, que define o socket como passivo. Esta função recebe apenas dois parâmetros: o próprio socket e o tamanho máximo da fila de conexões pendentes. Como nossa aplicação é simples, o servidor é limitado a 10 conexões.

Com a abertura passiva feita, o servidor está pronto para receber mensagens dos clientes, para isso, existe um loop infinito que aguarda a solicitação de algum cliente. Para aceitar uma conexão no socket passivo, é utilizada a função accept. Essa função recebe o socket do servidor, uma estrutura para armazenar o endereço do cliente e o tamanho em bytes dessa estrutura. A função retorna o identificador do socket do cliente. Com esse identificador definido, é possível prosseguir para a comunicação propriamente dita.

A real comunicação do servidor para o cliente ocorre dentro da função *handleServerClientComunication*, que recebe o identificador do socket do cliente. Através da função recv, é possível receber a mensagem que foi enviada pelo socket do cliente, que contém as suas coordenadas. Essa função recebe como parâmetros o identificador do cliente, um buffer onde a mensagem é armazenada, o tamanho da mensagem recebida em bytes e possíveis flags (no nosso caso 0). Como a mensagem é recebida em forma de string, é preciso realizar uma conversão para a struct Coordinates, para acessar mais facilmente a latitude e longitude do cliente.

Após receber as coordenas, o servidor pode escolher entre aceitar ou não a corrida, basta escrever, respectivamente, 1 ou 0 no console e pressionar a tecla *Enter*. Caso o motorista rejeite a corrida, a conexão é fechada e o servidor aguarda outra solicitação de corrida. Caso a corrida seja aceita, o processo de comunicação estende-se em um loop que itera até o motorista chegar na localização do cliente. Dessa forma, primeiramente é calculado a distância entre o motorista e o cliente usando a fórmula de *Haversine*. Em seguida, para simular o deslocamento do motorista, é decrementado 400 metros, de 2 em 2 segundos.

Além disso, de 2 em 2 segundos é enviada uma mensagem para o socket do cliente usando o método send, que recebe parâmetros semelhantes ao método recv, com a diferença que o buffer deve conter a mensagem que será enviada. No caso do presente

projeto, é enviada a distância entre o motorista e o cliente. Por fim, quando a distância chega a zero, o servidor utiliza a função send novamente para enviar um código sinalizando que o motorista chegou na localização do cliente. Com isso, o servidor fecha a conexão com o cliente utilizando a função close, que fecha o descritor de arquivo. Essa função recebe como parâmetro apenas o identificador do socket do cliente. Quando a conexão com um cliente termina, o servidor continua o seu loop principal de execução, aguardando por uma solicitação de um próximo cliente.

Cliente

O cliente é responsável por solicitar uma corrida ao servidor, fornecendo suas coordenadas de localização. Durante a execução, o cliente deve receber três parâmetros na linha de comando: o tipo de IP a ser utilizado, o endereço IP do servidor e a porta utilizada pelo servidor. Abaixo estão exemplos de comandos de execução:

- ./client ipv4 127.0.0.1 50501
- ./client ipv6 ::1 50501

Assim que a execução do cliente começa, ele pode escolher “0 - Sair” ou “1 - Solicitar Corrida”. Ao escolher sair, a execução do programa é interrompida. Caso escolha solicitar uma corrida, é iniciado o fluxo de comunicação. Em diversas partes, a utilização de funções e o processo seguido pelo cliente é semelhante ao do servidor, dessa forma, essas partes não serão tão detalhadas para evitar uma repetição desnecessária.

Primeiramente, é criado um socket confiável e de fluxo usando TCP, para isso, é utilizada a função socket. Em seguida, o cliente também utiliza a função utilitária *getServerAddressStructure* para contruir a estrutura de endereço do servidor baseada no tipo de IP selecionado.

Em seguida, é necessário obter o endereço do servidor em uma representação binária de 32 bits. Para isso, é utilizada a função inet_pton, que recebe a família de endereçamento, o endereço do servidor representado em string e sua interface de entrada. Com o endereço do servidor definido, o próximo passo é realizar a abertura ativa, realizada através da função connect, que estabelece a conexão com o servidor. Nessa função é utilizado o socket do cliente, a estrutura de endereço do servidor e seu tamanho.

Após a conexão com o servidor bem sucedida, o cliente utiliza a função send para enviar uma mensagem com suas coordenadas para o socket do servidor. Para realizar esse envio, as coordenadas são convertidas para uma string que é armazenada no buffer a ser enviado. Após esse primeiro envio, o programa entra no loop principal que itera até que o motorista chegue na localização do cliente. Dessa forma, a cada iteração, é utilizado um buffer e a função recv para receber as mensagens do servidor que informam a nova posição

do motorista em relação ao cliente. Esse loop é iterado até que o cliente recebe um dos seguintes códigos:

- NO_DRIVER_FOUND: Quando o cliente recebe essa mensagem, significa que o motorista não aceitou a corrida. Assim, é informado ao cliente que nenhum motorista foi encontrado e o menu inicial é impresso na tela novamente, aguardando a entrada do usuário.
- DRIVER_ARRIVED: Quando o cliente recebe essa mensagem, significa que o motorista chegou em sua localização. Assim, é informado ao cliente que o motorista chegou e o programa encerra-se.
- Qualquer outra mensagem que chega representa a distância do motorista até o usuário. Dessa forma, é impresso para o usuário “Motorista a Xm”.

Se o motorista recusar, ou o motorista chegar, a conexão entre o cliente e o servidor é fechada. Isso é feito através da função close, que recebe o socket do cliente como parâmetro e finaliza a conexão. Se o motorista recusou, o cliente pode optar por solicitar outra corrida, assim, a conexão é estabelecida e o processo é executado novamente.

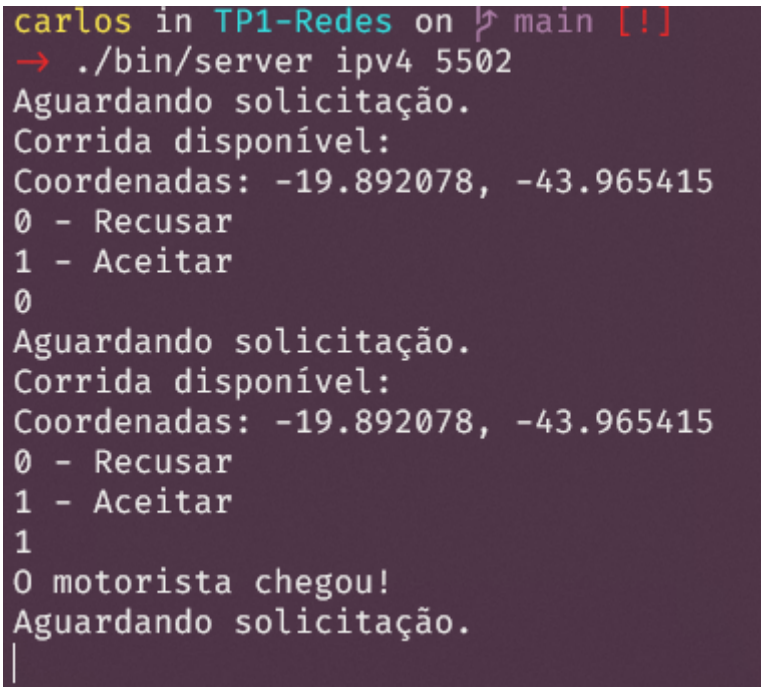
EXEMPLOS DE USO

Nesta seção, serão apresentados exemplos de uso do programa em funcionamento, demonstrando sua interação entre cliente e servidor. Por meio de exemplos práticos, será ilustrado como executar o programa, fornecer os parâmetros necessários e acompanhar a comunicação entre o cliente e o servidor durante a solicitação de uma corrida. Cada exemplo será acompanhado por prints que destacam as principais etapas da execução, proporcionando uma compreensão visual do processo. Esses exemplos ajudarão os usuários a entenderem melhor como utilizar e interagir com a aplicação em diferentes cenários de uso.

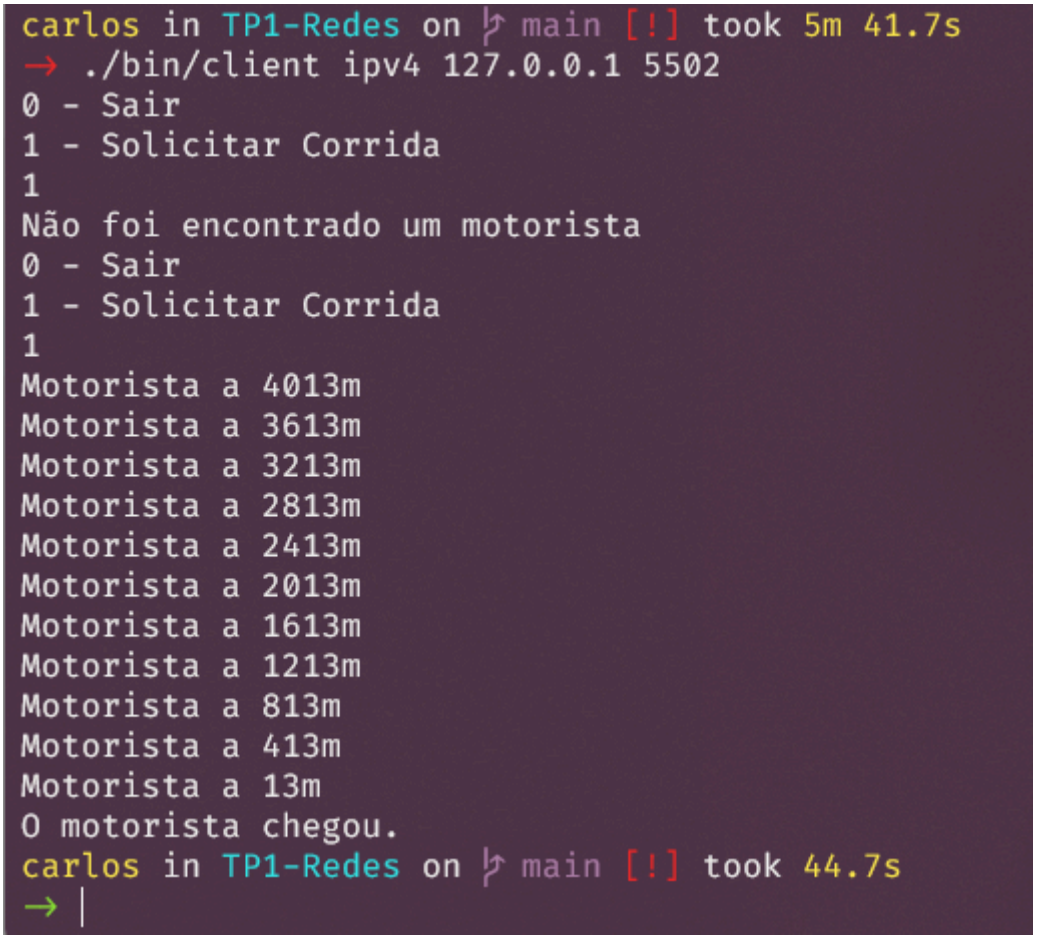
Exemplo 1 - Cliente solicita corrida duas vezes (IPV4)

Nesse caso de uso, o servidor será executado para receber conexões IPV4 na porta 5502. Nesse âmbito, o cliente receberá o endereço do servidor (endereço local: 127.0.0.1), a porta e o tipo IPV4.

Nesse exemplo, o usuário escolhe a opção de solicitar corrida, o motorista recebe as coordenadas do cliente, mas recusa a corrida. O usuário recebe a informação que nenhum motorista foi encontrado e solicita outra corrida. Dessa vez, o motorista aceita, o que gera diversas mensagens para o usuário da distância do motorista. Por fim, quando a distância chega a zero, os consoles do servidor e do cliente imprimem a mensagem de que o motorista chegou. O programa do servidor continua executando para receber outras possíveis solicitações de corrida, enquanto o do cliente é encerrado.



Exemplo 1 - Output Servidor

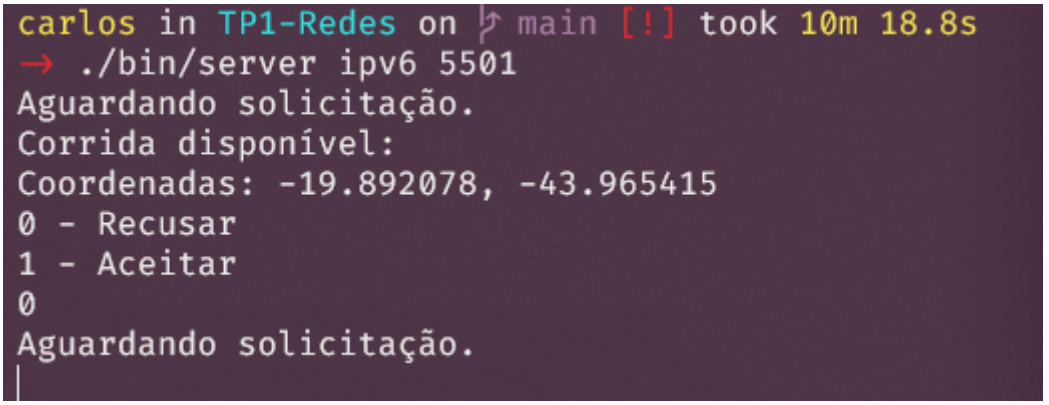


Exemplo 1 - Output Cliente

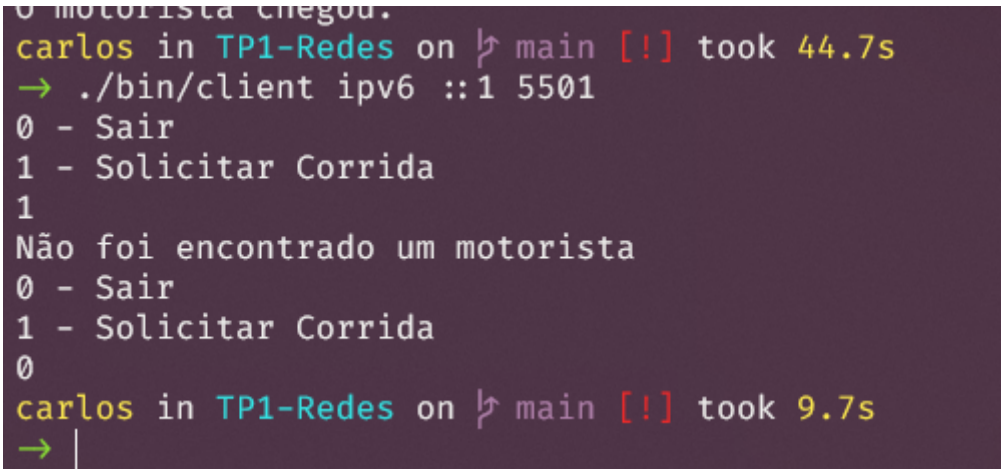
Exemplo 2 - Cliente solicita corrida e encerra aplicação (IPV6)

Nesse caso de uso, o servidor será executado para receber conexões IPV6 na porta 5501. Nesse âmbito, o cliente receberá o endereço do servidor (endereço local: ::1), a porta e o tipo IPV6.

Nesse exemplo, o usuário escolhe a opção de solicitar corrida, enviando uma mensagem com suas coordenadas para o servidor. Com isso, o motorista recebe essas coordenadas e rejeita a corrida, assim, o servidor envia a mensagem de que não foi encontrado nenhum motorista para a viagem. Por fim, o usuário escolhe a opção de sair da aplicação, que encerra o programa, enquanto o programa do servidor continua ativo para receber outras possíveis solicitações de corrida.



Exemplo 1 - Output Servidor



Exemplo 1 - Output Cliente

CONCLUSÃO

Neste trabalho prático, exploramos os fundamentos da comunicação entre clientes e servidores em redes de computadores, utilizando a linguagem de programação C e conexões TCP. Ao simular um cenário semelhante ao do aplicativo Uber, foi possível compreender os principais aspectos envolvidos na troca de mensagens entre processos distribuídos, incluindo a configuração de sockets, a comunicação cliente-servidor e o tratamento de erros e exceções.

Ao longo do desenvolvimento do projeto, destacou-se a importância de uma comunicação confiável e ordenada, garantida pelo TCP, para garantir a integridade das mensagens trocadas entre clientes e servidores. Além disso, a utilização de estruturas de endereços, como IPV4 e IPV6, ampliou a compreensão sobre a diversidade de protocolos de rede e a adaptação necessária para suportar diferentes ambientes de comunicação.

Ao finalizar este trabalho, estou mais familiarizado com os conceitos práticos de programação de redes e tenho uma visão mais clara dos desafios e oportunidades associados à construção de sistemas distribuídos. Este projeto proporcionou uma base sólida para explorar tópicos mais avançados em redes de computadores e preparou para enfrentar futuros desafios na área de desenvolvimento de software distribuído.