Trabalho prático 1 - Rede de Computadores

Janderson Glauber Mendes dos Santos - Matrícula 2020054544

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG) - Belo Horizonte - MG-BRASIL.

jandersonglauber@gmail.com

1.INTRODUÇÃO

Neste trabalho prático, será desenvolvida uma aplicação de console que simula o funcionamento básico do aplicativo Uber. O cliente desempenha o papel do usuário, enquanto o servidor atua como o motorista. O objetivo é obter uma compreensão prática da comunicação entre clientes e servidores por meio de conexão TCP.

Para a realização deste trabalho, além do conhecimento teórico adquirido durante a disciplina de Rede de Computadores, foram utilizadas as videoaulas do Professor Italo Cunha como referência para a estruturação do código.

1.1 AMBIENTE DE DESENVOLVIMENTO E BIBLIOTECAS:

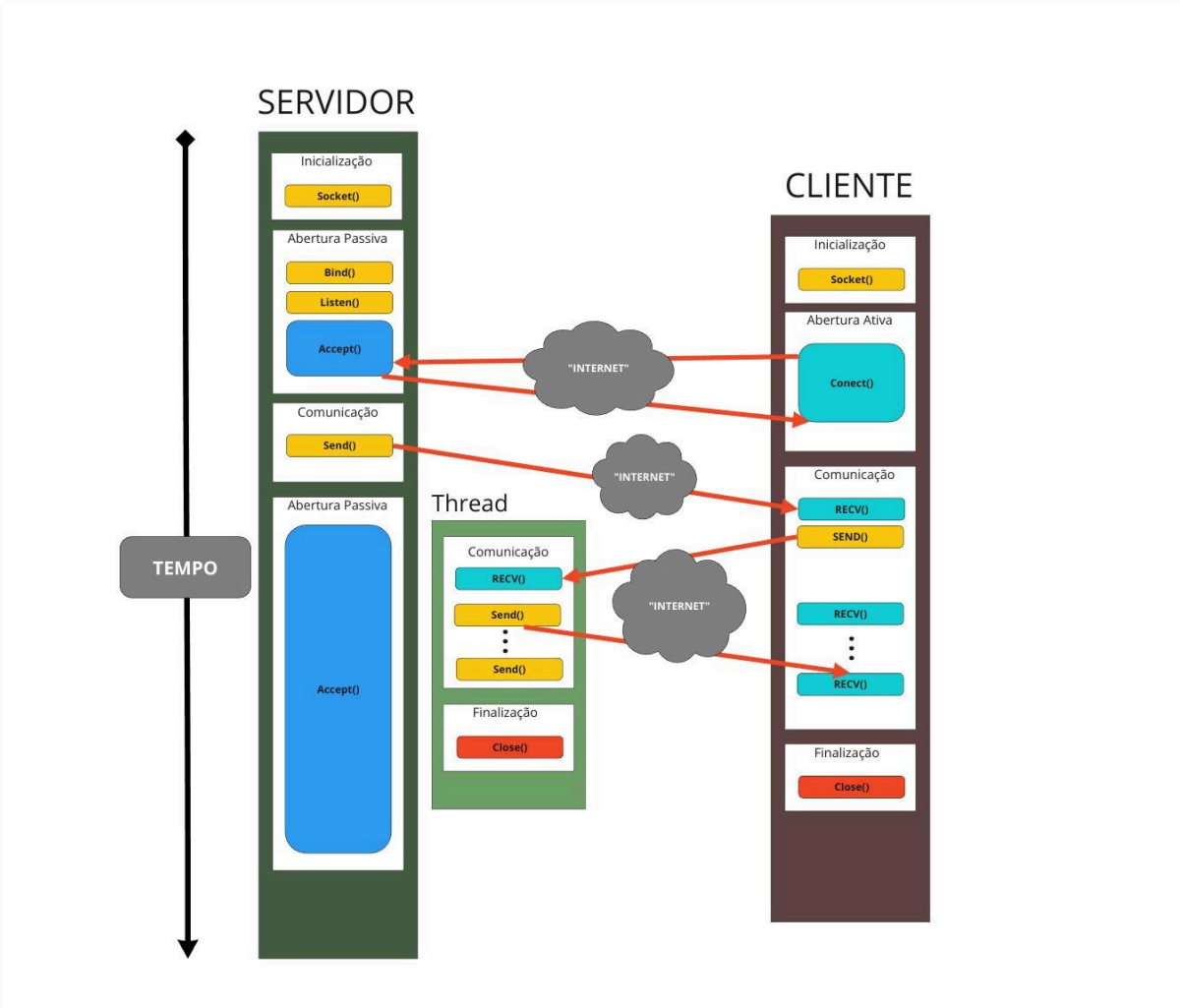
Linguagem : C
Compilador: GNU Compiler Collection (GCC)
Sistema Operacional: Ubuntu 22.04.1 LTS

Principais Bibliotecas:

- **pthread.h:** Esta biblioteca fornece suporte para programação concorrente e criação de threads. É usada para lidar com múltiplos clientes ou conexões simultâneas.
- **stdio.h:** Esta biblioteca fornece funções para entrada e saída padrão, como printf e scanf.
- **stdlib.h:** Fornece funções para alocação de memória dinâmica, conversão de tipos e outras utilidades básicas.
- **string.h:** Oferece várias funções para manipulação de strings, como strcpy e strlen.
- **unistd.h:** Essa biblioteca fornece acesso a várias constantes e funcionalidades do sistema operacional Unix, como close() e sleep().
- **sys/socket.h e sys/types.h:** Essas bibliotecas são usadas para trabalhar com soquetes e tipos de dados relacionados a soquetes, necessários para comunicação por rede.
- **math.h:** Oferece funções matemáticas, como sqrt() e pow().
- **arpa/inet.h:** Essa biblioteca contém funções e definições para manipulação de endereços IP e conversão de endereços de rede entre formatos.
- **inttypes.h:** Fornece tipos inteiros com largura específica, como int8_t e uint16_t, úteis para lidar com diferentes arquiteturas.

1.2 : DIAGRAMA DE FUNCIONAMENTO DO SERVIDOR:

OBJ



A partir do diagrama, podemos compreender o funcionamento geral do programa. Ele representa um fluxo completo, onde um cliente solicita uma corrida e um motorista aceita essa solicitação. Para que o programa siga essa lógica de negócios, inicializa-se um socket para o servidor, que então se comunica com o sistema operacional para estabelecer conexões apropriadas.

Após a inicialização do socket, o servidor utiliza a função bind() para especificar o endereço de rede e a porta onde ficará disponível para os clientes se conectarem. Essas operações ocorrem localmente e retornam imediatamente.

Em seguida, a função listen() é acionada para iniciar uma espera passiva, permitindo que o servidor aguarde por conexões. Finalmente, para concluir essa espera passiva, o servidor utiliza a função accept() para esperar a chegada da conexão do cliente. É importante observar que a função accept() pode envolver comunicação com um dispositivo remoto e, portanto, pode levar um tempo variável para retornar.

Eventualmente, um cliente surgirá para estabelecer uma conexão. Este cliente também passará por um processo de inicialização, criando um socket e, em seguida, avançará para a etapa de inicialização ativa. Nessa fase, o cliente invocará a função connect(), fornecendo o identificador do servidor para especificar onde se conectar e enviar ou receber dados.

Após o servidor receber a chamada connect(), ele responderá positivamente, confirmando a conexão.

A seguir, o servidor entra em uma fase de comunicação, enviando uma resposta sobre a corrida para o cliente usando a função send(), que é recebida pelo cliente através da função recv(), iniciando assim sua própria fase de comunicação. Em seguida, o servidor cria uma nova thread para atender múltiplos clientes simultaneamente, enquanto ambos, servidor e cliente, mantêm as comunicações necessárias para a execução das regras de negócio.

É crucial observar que, após criar uma nova thread, o servidor retorna à etapa de abertura passiva para aguardar novos clientes. Outro ponto fundamental é a sincronização durante a fase de comunicação; tanto o servidor quanto o cliente devem estar sincronizados adequadamente para o envio e recebimento de mensagens, caso contrário, a comunicação não ocorrerá conforme o esperado.

Nos próximos tópicos, serão abordadas a implementação de cada uma dessas etapas, assim como as regras de negócio envolvidas em cada uma. Por fim, ao término do trabalho, será apresentado o funcionamento do código.

2.Detalhes de implementação:

2.1 Arquivo Common.c:

Este arquivo contém implementações de funções auxiliares destinadas a serem utilizadas tanto pelo server.c quanto pelo client.c. Essas funções foram desenvolvidas com base nas videoaulas disponibilizadas para consulta na documentação do trabalho, sendo creditadas ao Professor Ítalo Cunha. Algumas adaptações foram realizadas para atender às exigências específicas do projeto.

```
7   void logexit(const char *msg)
8   {
9       perror(msg);
10      exit(EXIT_FAILURE);
11  }
```

A função logexit é responsável por imprimir uma mensagem de erro formatada utilizando perror e encerrar a execução do programa com um código de saída indicando falha (EXIT_FAILURE):

Parâmetro:

- const char *msg: Uma string contendo a mensagem de erro a ser exibida.

Funcionamento:

- perror(msg): Esta função imprime a string msg, seguida por uma descrição do erro atual, que é determinado pela variável global errno. A descrição do erro é impressa no formato "msg: descrição do erro\n".

- exit(EXIT_FAILURE): Encerra a execução do programa com um código de saída indicando falha (EXIT_FAILURE). Isso implica que algum erro crítico ocorreu e o programa não pode continuar sua execução normal.

```c
int addrparse(const char *addrstr, const char *portstr, struct sockaddr_storage *storage)
{
    if (addrstr == NULL || portstr == NULL)
    {
        return -1;
    }
    uint16_t port = (uint16_t)atoi(portstr);      You, 22 hours ago • Uncommitted changes
    if (port == 0)
    {
        return -1;
    }
    port = htons(port);

    struct in_addr inaddr4;
    if (inet_pton(AF_INET, addrstr, &inaddr4))
    {
        struct sockaddr_in *addr4 = (struct sockaddr_in *)storage;
        addr4->sin_family = AF_INET;
        addr4->sin_port = port;
        addr4->sin_addr = inaddr4;
        return 0;
    }
    struct in6_addr inaddr6;
    if (inet_pton(AF_INET6, addrstr, &inaddr6))
    {
        struct sockaddr_in6 *addr6 = (struct sockaddr_in6 *)storage;
        addr6->sin6_family = AF_INET6;
        addr6->sin6_port = port;
        memcpy(&(addr6->sin6_addr), &inaddr6, sizeof(inaddr6));
        return 0;
    }

    return -1;
}
```

Essa função addrparse é responsável por analisar uma representação de endereço IP (IPv4 ou IPv6) e uma porta, e armazená-los em uma estrutura sockaddr_storage.

Parâmetros:

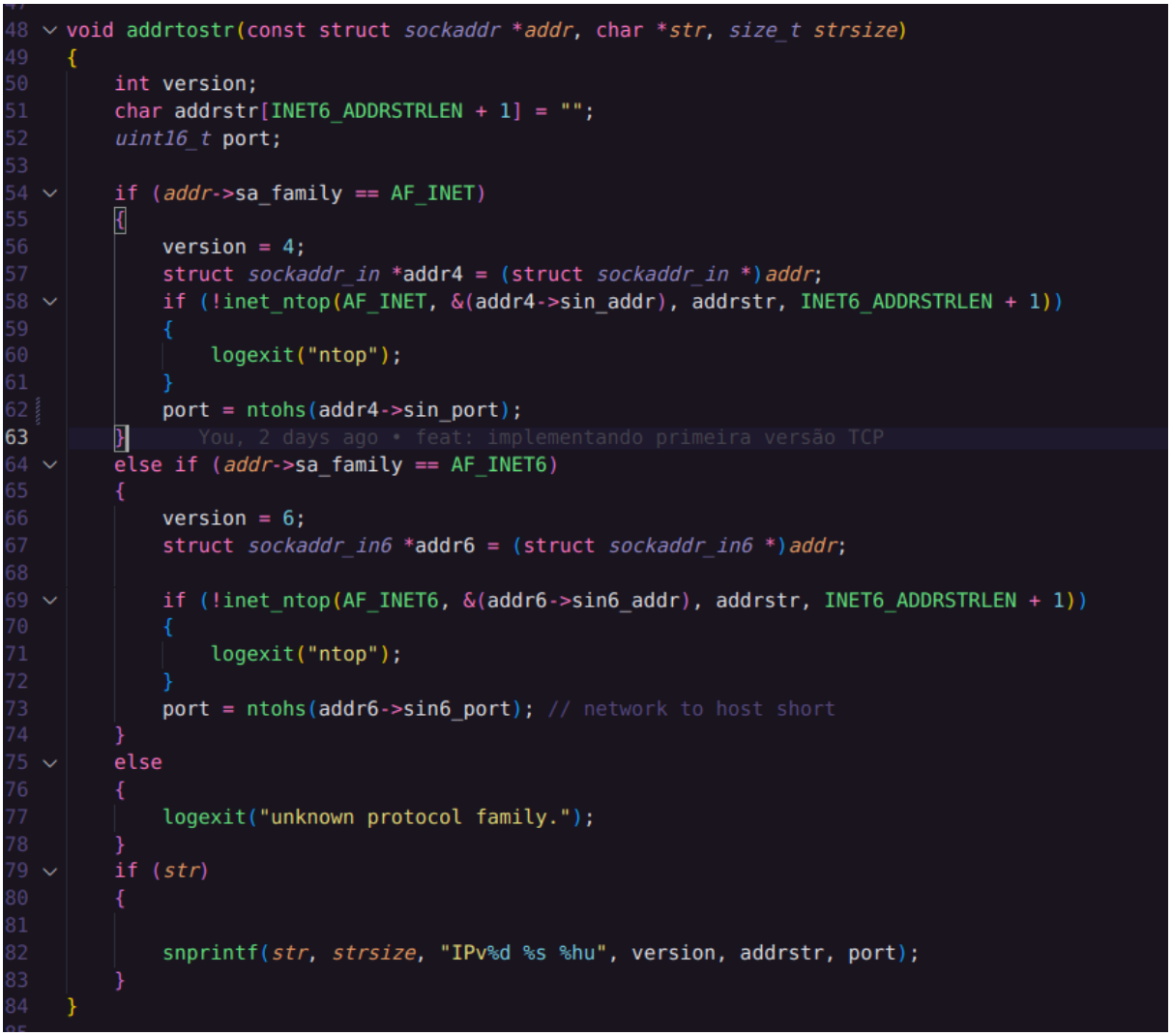
- const char *addrstr: A representação do endereço IP (IPv4 ou IPv6) como uma string.
- const char *portstr: A representação da porta como uma string.
- struct sockaddr_storage *storage: Um ponteiro para a estrutura de armazenamento do endereço.

Funcionamento:

- Verifica se os parâmetros addrstr e portstr são diferentes de NULL. Se algum deles for NULL, retorna -1, indicando um erro.

- Converte a string portstr para um número inteiro usando atoi, que converte uma string para um inteiro.
- Verifica se a porta é válida (diferente de 0). Se a porta for 0, retorna -1, indicando um erro.
- Converte a porta para o formato de byte de ordem de rede (big-endian) usando htons, que converte um número inteiro de 16 bits do formato de host para o formato de rede.
- Tenta converter addrstr para um endereço IPv4 usando inet_pton. Se a conversão for bem-sucedida, preenche a estrutura sockaddr_in com os dados do endereço e da porta e retorna 0, indicando sucesso.
- Se a conversão para IPv4 falhar, tenta converter addrstr para um endereço IPv6 usando inet_pton. Se a conversão for bem-sucedida, preenche a estrutura sockaddr_in6 com os dados do endereço e da porta e retorna 0, indicando sucesso.
- Se nenhuma das conversões for bem-sucedida, retorna -1, indicando um erro.

Essa função é útil para validar e converter representações de endereços IP e portas e armazená-las em uma estrutura de endereço, facilitando a configuração e uso de sockets de rede.



```c
void addrtostr(const struct sockaddr *addr, char *str, size_t strsize)
{
    int version;
    char addrstr[INET6_ADDRSTRLEN + 1] = "";
    uint16_t port;

    if (addr->sa_family == AF_INET)
    {
        version = 4;
        struct sockaddr_in *addr4 = (struct sockaddr_in *)addr;
        if (!inet_ntop(AF_INET, &(addr4->sin_addr), addrstr, INET6_ADDRSTRLEN + 1))
        {
            logexit("ntop");
        }
        port = ntohs(addr4->sin_port);
    }      You, 2 days ago • feat: implementando primeira versão TCP
    else if (addr->sa_family == AF_INET6)
    {
        version = 6;
        struct sockaddr_in6 *addr6 = (struct sockaddr_in6 *)addr;

        if (!inet_ntop(AF_INET6, &(addr6->sin6_addr), addrstr, INET6_ADDRSTRLEN + 1))
        {
            logexit("ntop");
        }
        port = ntohs(addr6->sin6_port); // network to host short
    }
    else
    {
        logexit("unknown protocol family.");
    }
    if (str)
    {

        snprintf(str, strsize, "IPv%d %s %hu", version, addrstr, port);
    }
}
```

Essa função addrtostr converte uma estrutura de endereço (struct sockaddr) em uma representação de string legível.

Parâmetros:

- const struct sockaddr *addr: Um ponteiro para a estrutura de endereço a ser convertida.
- char *str: Um ponteiro para o buffer onde a string resultante será armazenada.
- size_t strsize: O tamanho do buffer str.

Funcionamento:

- Declara variáveis locais:
    - int version: Indica a versão do protocolo IP (IPv4 ou IPv6).
    - char addrstr[INET6_ADDRSTRLEN + 1]: Uma string que armazenará o endereço IP convertido.
    - uint16_t port: A porta do endereço.
- Verifica o tipo de família de endereço (sa_family) da estrutura de endereço (addr).
- Se a família de endereço for AF_INET (IPv4):
    - Define version como 4.
    - Converte o endereço IPv4 para uma representação de string legível usando inet_ntop.
    - Obtém a porta do endereço em um formato de host (big-endian) usando ntohs.
- Se a família de endereço for AF_INET6 (IPv6):
    - Define version como 6.
    - Converte o endereço IPv6 para uma representação de string legível usando inet_ntop.
    - Obtém a porta do endereço em um formato de host (big-endian) usando ntohs.
- Se a família de endereço não for nem AF_INET nem AF_INET6, chama a função logexit para imprimir uma mensagem de erro indicando uma família de protocolo desconhecida e encerra o programa.
    - Usa snprintf para formatar a string resultante com a versão do protocolo, o endereço IP e a porta.
    - Armazena a string resultante no buffer str.

Essa função é útil para obter uma representação legível de um endereço IP e sua porta a partir de uma estrutura de endereço, facilitando a exibição ou registro de informações de conexão em aplicativos de rede.

```
You, 2 days ago • feat: implementando primeira versão TCP
int server_sockaddr_init(const char *proto, const char *portstr, struct sockaddr_storage *storage)
{
    uint16_t port = (uint16_t)atoi(portstr); // unsigned short
    if (port == 0)
    {
        return -1;
    }
    port = htons(port); // host to network short

    memset(storage, 0, sizeof(*storage));
    if (0 == strcmp(proto, "ipv4"))
    {
        struct sockaddr_in *addr4 = (struct sockaddr_in *)storage;
        addr4->sin_family = AF_INET;
        addr4->sin_addr.s_addr = INADDR_ANY;
        addr4->sin_port = port;
        return 0;
    }
    else if (0 == strcmp(proto, "ipv6"))
    {
        struct sockaddr_in6 *addr6 = (struct sockaddr_in6 *)storage;
        addr6->sin6_family = AF_INET6;
        addr6->sin6_addr = in6addr_any;
        addr6->sin6_port = port;
        return 0;
    }
    else
    {
        return -1;
    }
}
```

A função server_sockaddr_init é responsável por inicializar uma estrutura de endereço para uso em um servidor.

Parâmetros:

- const char *proto: Uma string indicando o protocolo a ser usado, que pode ser "ipv4" ou "ipv6".
- const char *portstr: Uma string representando o número da porta do servidor.
- struct sockaddr_storage *storage: Um ponteiro para a estrutura de armazenamento do endereço do servidor.

Funcionamento:

- Converte a string portstr para um número inteiro usando atoi, que converte uma string para um inteiro.
- Verifica se a porta é válida (diferente de 0). Se a porta for 0, retorna -1, indicando um erro.
- Converte a porta para o formato de byte de ordem de rede (big-endian) usando htons, que converte um número inteiro de 16 bits do formato de host para o formato de rede.
- Inicializa a estrutura sockaddr_storage apontada por storage com zeros usando memset.
- Se o protocolo especificado for "ipv4":
    - Converte o ponteiro storage para struct sockaddr_in.
    - Define a família de endereço como AF_INET (IPv4).
    - Define o endereço IP como INADDR_ANY, indicando que o servidor deve aceitar conexões em qualquer interface de rede disponível.

    - Define a porta do servidor.
    - Retorna 0 para indicar sucesso.

- Se o protocolo especificado for "ipv6":
    - Converte o ponteiro storage para struct sockaddr_in6.
    - Define a família de endereço como AF_INET6 (IPv6).
    - Define o endereço IP como in6addr_any, indicando que o servidor deve aceitar conexões em qualquer interface de rede IPv6 disponível.
    - Define a porta do servidor.
    - Retorna 0 para indicar sucesso.

- Se o protocolo especificado não for "ipv4" nem "ipv6", retorna -1, indicando um erro.

Essa função é útil para configurar e inicializar uma estrutura de endereço para um servidor TCP/IP, preparando-a para ser usada na vinculação (bind) a um socket do servidor.

2.2 Arquivo server.c:

Esse arquivo implementa o servidor/motorista que receberá a solicitação de corrida dos clientes. Abaixo será analisada em detalhe a implementação desse servidor.

Funções e estruturas auxiliares:

```c
void usage(int argc, char **argv)
{
    printf("usage: %s <ipv4|ipv6> <server port>\n", argv[0]);
    printf("example: %s ipv4 51511", argv[0]);
    exit(EXIT_FAILURE);
}
```

Esta função é responsável por imprimir uma mensagem de uso do programa, mostrando como ele deve ser executado e um exemplo de uso. Ela é frequentemente chamada quando o usuário fornece argumentos inválidos ao executar o programa.

```c
    You, 2 days ago | 1 author (You)
22  struct client_data
23  {
24      int csock;
25      struct sockaddr_storage storage;
26  };
27
```

Descrição: Esta estrutura de dados é utilizada para armazenar informações sobre um cliente que se conecta ao servidor, como o descritor de arquivo do socket do cliente (csock) e a estrutura de endereço associada (storage).

```
double haversine(double lat1, double lon1,
                 double lat2, double lon2)
{
    double dLat = (lat2 - lat1) *
                  M_PI / 180.0;
    double dLon = (lon2 - lon1) *
                  M_PI / 180.0;

    lat1 = (lat1)*M_PI / 180.0;
    lat2 = (lat2)*M_PI / 180.0;

    double a = pow(sin(dLat / 2), 2) +
               pow(sin(dLon / 2), 2) *
                  cos(lat1) * cos(lat2);
    double rad = 6371;
    double c = 2 * asin(sqrt(a));
    return rad * c;
}
```

Esta função calcula a distância entre dois pontos na superfície da Terra utilizando a fórmula de Haversine. É comumente usada para calcular a distância entre duas coordenadas geográficas (latitude e longitude).

A autoria da função de haversine é creditada ao blog geeks for geeks e indicada para uso na proposta do trabalho.

Estruturação do servidor: main:

```c
int main(int argc, char **argv)
{

    struct sockaddr_storage storage;
    if (0 != server_sockaddr_init(argv[1], argv[2], &storage))
    {      You, 2 days ago • feat: implementando primeira versão TCP
        usage(argc, argv);
    }

    int s;
    s = socket(storage.ss_family, SOCK_STREAM, 0);
    if (s == -1)
    {
        logexit("socket");
    }

    int enable = 1;
    if (0 != setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)))
    {
        logexit("setsockopt");
    }

    struct sockaddr *addr = (struct sockaddr *)(&storage);
    if (0 != bind(s, addr, sizeof(storage)))
    {
        logexit("bind");
    }

    if (0 != listen(s, 10))
    {
        logexit("listen");
    }

    printf("Aguardando Solicitação.\n");
```

Inicialmente, o servidor inicia criando um socket com a função server_sockaddr_init(), a qual determina se será utilizado o IPv4 ou IPv6 e realiza as transformações necessárias para configurar adequadamente o endereço e a porta. Além disso, os parâmetros passados pelo terminal são verificados utilizando a função usage() para garantir que estejam corretos.

Em seguida, é criado um socket s com os parâmetros storage.ss_family, SOCK_STREAM, 0, representando, respectivamente, o tipo de endereço extraído da estrutura storage, o tipo de socket e a ausência de opções específicas. Após a criação do socket, é verificado se foi criado com sucesso, concluindo assim a fase de inicialização. Junto a isso usamos o setsockopt() para definir o reuso do servidor , ou seja , utilizaremos o endereço mesmo que ele ja esteja sendo utilizado, isso é importante para que o servidor seja capaz de atender mais de um cliente ao mesmo tempo

Posteriormente, o servidor entra na etapa de espera passiva, na qual define o endereço com a estrutura struct sockaddr e realiza o bind() associando o socket s ao endereço addr, juntamente com seu tamanho. Em seguida, é executado um listen() no socket s com um backlog de 10, indicando o número máximo de conexões pendentes. Após essa etapa, o servidor emite a mensagem "Aguardando conexão", indicando que está pronto para aceitar conexões.

```c
while (1)
{
    struct sockaddr_storage cstorage;
    struct sockaddr *caddr = (struct sockaddr *)(&cstorage);
    socklen_t caddrlen = sizeof(cstorage);

    int csock = accept(s, caddr, &caddrlen);
    if (csock == -1)
    {
        logexit("accept");
    }
    int acceptClient = -1;
    printf("Corrida Disponivel: \n");
    printf("0 - Recusar\n");
    printf("1 - Aceitar\n");
    scanf("%d", &acceptClient);
    if (acceptClient == 0)
    {
        char *msg = "Não foi encontrado um motorista";
        int count = send(csock, msg, strlen(msg) + 1, 0);
        if (count != strlen(msg) + 1)
        {
            logexit("send");
        }
        close(csock);
    }
```

Para manter o servidor em execução de forma contínua, implementamos um loop while(1) para garantir sua atividade constante. Utilizamos a função accept() para receber o socket do cliente, criando variáveis cstorage, caddr e caddrlen para manipular o endereço do cliente. O accept() cria um novo socket para o cliente, identificado como csock neste código. Em seguida, verificamos se a operação ocorreu sem problemas, marcando o fim da fase de conexão passiva e permitindo que o servidor receba conexões de clientes. Caso um cliente se conecte e solicite uma corrida, o servidor/motorista tem duas opções: aceitar ou recusar. Se o servidor/motorista recusar a solicitação, envia uma mensagem (send()) ao cliente informando que nenhum motorista foi encontrado. O servidor verifica se o envio da mensagem ocorreu sem erros e, por fim, fecha a conexão usando close() no socket do cliente.

```
    }
    else if (acceptClient == 1)
    {
        char *msg = "";
        int count = send(csock, msg, strlen(msg) + 1, 0);
        if (count != strlen(msg) + 1)
        {
            logexit("send");
        }

        while (getchar() != '\n')
            ;
        struct client_data *cdata = malloc(sizeof(*cdata));
        if (!cdata)
        {
            logexit("malloc");
        }
        cdata->csock = csock;
        memcpy(&(cdata->storage), &cstorage, sizeof(cstorage));

        pthread_t tid;
        pthread_create(&tid, NULL, client_thread, cdata);      You, yeste
    }
}
```

Ainda no while, quando o servidor decide aceitar a solicitação do cliente para uma corrida, ele enviamos uma mensagem vazia ao cliente para indicar que a solicitação foi recebida. Depois disso, o servidor aloca dinamicamente memória para uma estrutura client_data, que é usada para passar informações sobre o cliente para uma nova thread. Se a alocação de memória falhar, o programa é encerrado. Caso contrário, os dados do socket do cliente (csock) e o endereço do cliente (cstorage) são copiados para a estrutura client_data. Então, uma nova thread é criada para lidar com as comunicações com esse cliente.

```
void *client_thread(void *data)
{
    Coordinate coordServ = {-19.9227,
                            -43.9451};

    struct client_data *cdata = (struct client_data *)data;

    Coordinate coordCli;
    memset(&coordCli, 0, sizeof(coordCli));
    ssize_t count = recv(cdata->csock, &coordCli, sizeof(coordCli), 0);
    if (count < 0)
    {
        perror("recv");
```

```
        exit(EXIT_FAILURE);
    }
    double distance = haversine(coordServ.latitude, coordServ.longitude,
coordCli.latitude, coordCli.longitude) * 1000;
    while (distance > 0)
    {
        distance = distance - 400;
        int count = send(cdata->csock, (double *)&distance,
sizeof(distance), 0);
        if (count != sizeof(distance))
        {
            logexit("send");
        }

        usleep(2000000);
    }
    printf("O motorista chegou!\n");

    close(cdata->csock);
    pthread_exit(EXIT_SUCCESS);
}
```

Esta função é responsável por criar as threads que gerenciarão a comunicação entre o servidor e o cliente, permitindo que o servidor continue atendendo outras conexões simultaneamente. Inicialmente, as coordenadas do servidor são definidas para permitir cálculos de distância em relação ao cliente posteriormente. Em seguida, a estrutura client_data, contendo informações sobre o cliente, é acessada a partir do parâmetro data. As coordenadas do cliente são inicializadas e, então, as coordenadas do cliente são recebidos pelo socket do cliente (cdata->csock). Se ocorrer algum erro durante a recepção dos dados, o programa é encerrado. Com base nas coordenadas do cliente e do serviço, é calculada a distância entre eles utilizando a fórmula de Haversine. Enquanto a distância entre o cliente e o serviço for maior que zero, a distância é atualizada e enviada de volta ao cliente em intervalos regulares de 2000ms. Quando a distância se torna zero, indicando que o motorista chegou ao destino do cliente, uma mensagem é impressa na tela do servidor informando que o motorista chegou. Por fim, o socket do cliente é fechado e a thread é encerrada. É importante destacar que o envio das mensagens (send()) está sincronizado com o recv() do cliente, garantindo que as mensagens sejam enviadas e recebidas corretamente.

Dessa forma finalizamos a compreensão do arquivo do servidor.

2.3 Arquivo client.c:

Nesse arquivo é implementado o cliente que solicitará corridas ao motorista/servido.

```c
int main(int argc, char **argv)
    while (driverNotFound)
    {

        int userMenuOption = -1;
        printf("0 - Sair\n");
        printf("1 - Solicitar Corrida\n");
        scanf("%d", &userMenuOption);
        if (userMenuOption == 0)
        {       You, yesterday • Feat: Rascunho de todas as funcionalidades
            return 0;
        }
        else if (userMenuOption == 1)
        {
            while (getchar() != '\n')
                ;
        }

        struct sockaddr_storage storage;
        if (0 != addrparse(argv[1], argv[2], &storage))
        {
            usage(argc, argv);
        }

        int s;
        s = socket(storage.ss_family, SOCK_STREAM, 0);

        if (s == -1)
        {
            logexit("socket");
        }
        struct sockaddr *addr = (struct sockaddr *)(&storage);
        if (0 != connect(s, addr, sizeof(storage)))
        {
            logexit("conect");
        }

        char buf[BUFSZ];
        memset(buf, 0, BUFSZ);
        int count = recv(s, buf, BUFSZ - 1, 0);
        if (count == 0)
        {
            printf("Conexão encerrada pelo servidor.\n");
            exit(EXIT_SUCCESS);
        }
```

O código do cliente começa com um loop while que continua até que um motorista seja encontrado para uma carona. Se o servidor/motorista recusar a carona, o cliente retorna ao menu inicial, onde tem a opção de solicitar outra corrida ou sair do programa. Quando o cliente solicita uma corrida, o programa analisa o endereço IP e a porta fornecidos pelo cliente, realiza o parsing desses dados e inicializa um socket (etapa de inicialização). Em seguida, é estabelecida uma conexão com o servidor por meio da função connect(), iniciando assim uma abertura ativa. Após a conexão, o servidor tem duas opções: aceitar ou rejeitar a solicitação, como já discutido anteriormente. O cliente aguarda a decisão do servidor por meio de um recv(). Se a resposta não for uma mensagem vazia, significa que o servidor respondeu com "Motorista não encontrado", então a conexão é fechada e o cliente retorna ao início do loop while para tentar uma nova conexão com um motorista.

```
        }
        else
        {
            if (strcmp(buf, "") != 0)
            {
                printf("%s\n", buf);
            }
            else
            {
                driverNotFound = 0;
                Coordinate coordCli = {-19.8679429, -43.9697909};
                memset(buf, 0, BUFSZ);
                count = send(s, (void *)&coordCli, sizeof(coordCli) + 1, 0);
                if (count != sizeof(coordCli) + 1)
                {
                    logexit("send");
                }

                memset(buf, 0, BUFSZ);
                double distance = 1;
                while (distance > 0)
                {
                    ssize_t count = recv(s, &distance, sizeof(distance), 0);
                    distance > 0 ? printf("Motorista a %dm \n", (int)distance) : printf("O motorista chegou.");
                    if (count == 0)
                    {
                        break;
                    }
                }

                close(s);
                puts(buf);
                exit(EXIT_SUCCESS);
            }
        }
    }
}
```

Quando a solicitação da corrida é aceita, o cliente envia as coordenadas para o servidor (por meio do socket s). Em seguida, estabelecemos uma lógica de sincronização com o servidor/motorista: o cliente recebe a distância do motorista enviada pelo servidor até que ela se torne zero. As funções recv() são configuradas para receber os dados seguindo a mesma estratégia definida pelo servidor, ou seja, até que a distância seja zero. Isso garante a sincronização entre o cliente e o servidor durante toda a interação. Após a chegada do motorista, o socket é fechado, a conexão é encerrada e o programa é finalizado.

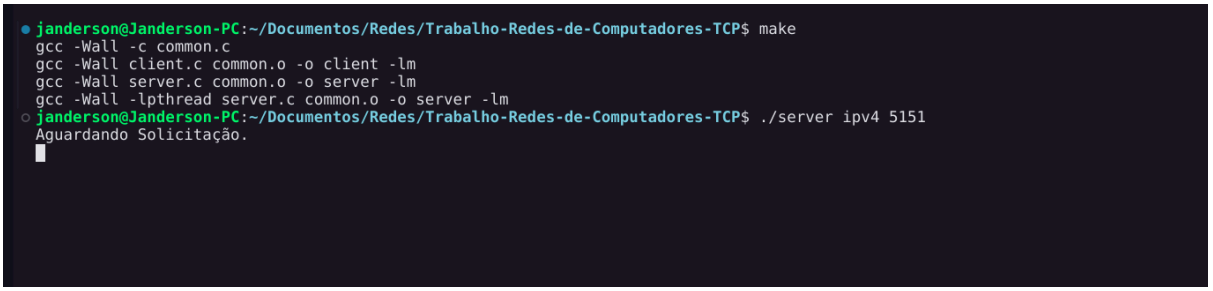
2.3 Detalhamento das principais funções da biblioteca de redes:

- **socket():** A função socket() é usada para criar um novo ponto de extremidade de comunicação, ou socket. Ela retorna um descritor de arquivo que pode ser usado em operações de entrada e saída relacionadas à rede. Ao chamar socket(), você especifica o tipo de comunicação desejado (por exemplo, TCP ou UDP) e o protocolo de rede a ser usado.
- **setsockopt():** A função setsockopt() é usada para definir opções para um socket. Por exemplo, você pode usar setsockopt() para configurar opções de soquete, como reutilização de endereço ou timeout de conexão.
- **bind():** A função bind() associa um nome a um socket, ou seja, atribui um endereço IP e um número de porta a um socket. Isso é necessário para que outros programas possam se conectar a esse socket.
- **listen():** A função listen() é usada em sockets do tipo SOCK_STREAM (como TCP) e instrui o sistema a aceitar conexões de entrada para esse socket. Ela define o socket para o modo de escuta, permitindo que ele aceite conexões de entrada de clientes.
- **accept():** A função accept() é usada em sockets que estão no modo de escuta (após a chamada listen()) para aceitar uma nova conexão de entrada. Quando uma conexão é aceita, accept() cria um novo socket conectado e retorna um descritor de arquivo para esse novo socket.
- **send():** A função send() é usada para enviar dados através de um socket conectado. Ela aceita o descritor de arquivo do socket, um ponteiro para os dados a serem enviados e o tamanho desses dados.
- **close():** A função close() é usada para fechar um descritor de arquivo, neste caso, um socket. Ela libera os recursos associados ao socket e encerra a comunicação. Depois de chamar close(), o socket não pode mais ser usado para enviar ou receber dados.
- **connect():** A função connect() é usada em sockets do tipo SOCK_STREAM (como TCP) para iniciar uma conexão com outro ponto de extremidade de comunicação, geralmente um servidor remoto. Ela recebe o descritor de arquivo do socket, uma estrutura sockaddr contendo o endereço do servidor (geralmente inicializada com a função addrparse() e convertida em um ponteiro struct sockaddr *) e o tamanho dessa estrutura. connect() estabelece uma conexão de rede com o servidor especificado.
- **recv():** A função recv() é usada para receber dados de um socket conectado. Ela aceita o descritor de arquivo do socket, um ponteiro para o buffer onde os dados recebidos serão armazenados e o tamanho máximo desses dados a serem recebidos. recv() bloqueia o programa até que os dados sejam recebidos ou até que ocorra um erro. Após a recepção dos dados, recv() retorna o número de bytes recebidos ou -1 em caso de erro.

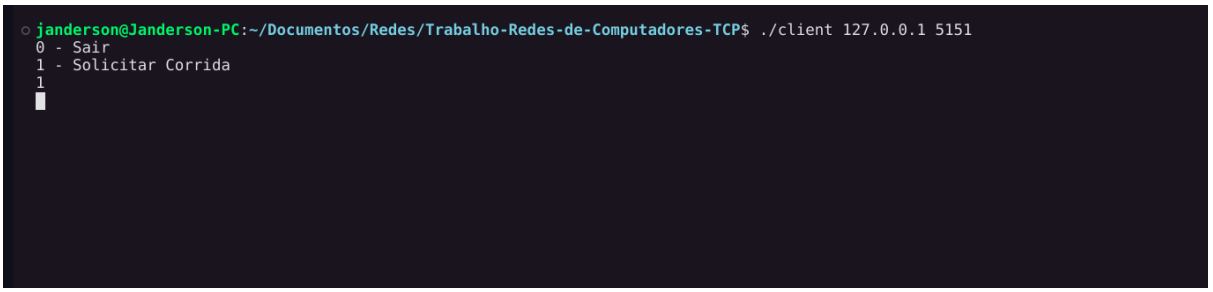
3.Funcionamento do Código em Execução:

Comando para compilação: make

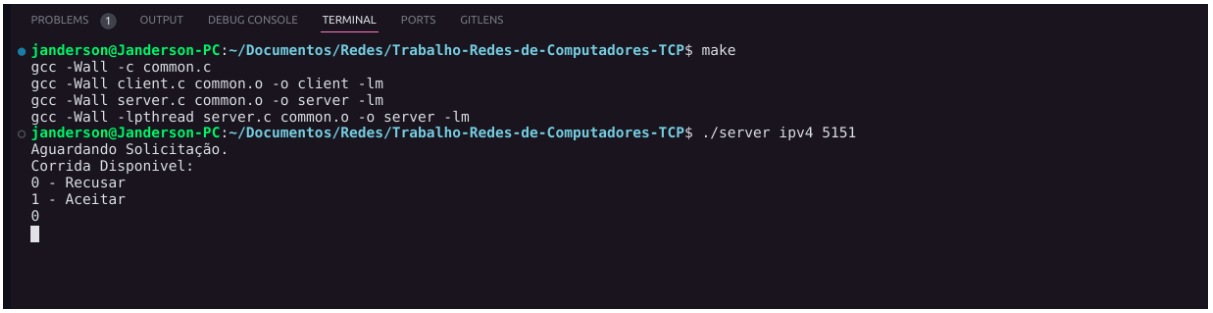
3.1 Motorista Recusa Solicitação IPv4:



É executado o comando make para compilação, em sequencia executamos o servidor com a versão IPv4 e Porta 5151. O servidor então espera por uma conexão.



Executamos o cliente passando como parâmetro o endereço ipv4 do servidor e a porta a qual ele será conectado. Após isso escolhemos a opção 1 para solicitar uma corrida.



Motorista recebe a solicitação de corrida e opta por recusá-la. Após isso o servidor retorna para um estado de espera aguardando uma conexão.

```
janderson@Janderson-PC:~/Documentos/Redes/Trabalho-Redes-de-Computadores-TCP$ ./client 127.0.0.1 5151
0 - Sair
1 - Solicitar Corrida
1
Não foi encontrado um motorista
0 - Sair
1 - Solicitar Corrida
```

Enquanto isso o cliente recebe uma mensagem na qual a corrida foi recusada e é retornado para o menu inicial novamente.

```
janderson@Janderson-PC:~/Documentos/Redes/Trabalho-Redes-de-Computadores-TCP$ ./client 127.0.0.1 5151
0 - Sair
1 - Solicitar Corrida
1
Não foi encontrado um motorista
0 - Sair
1 - Solicitar Corrida
0
janderson@Janderson-PC:~/Documentos/Redes/Trabalho-Redes-de-Computadores-TCP$
```

Ao clicar em sair o cliente desiste de encontrar um motorista e a conexão é finalizada.

3.2 Motorista Aceita Solicitação IPv4:

```
janderson@Janderson-PC:~/Documentos/Redes/Trabalho-Redes-de-Computadores-TCP$ ./server ipv4 5151
Aguardando Solicitação.
```

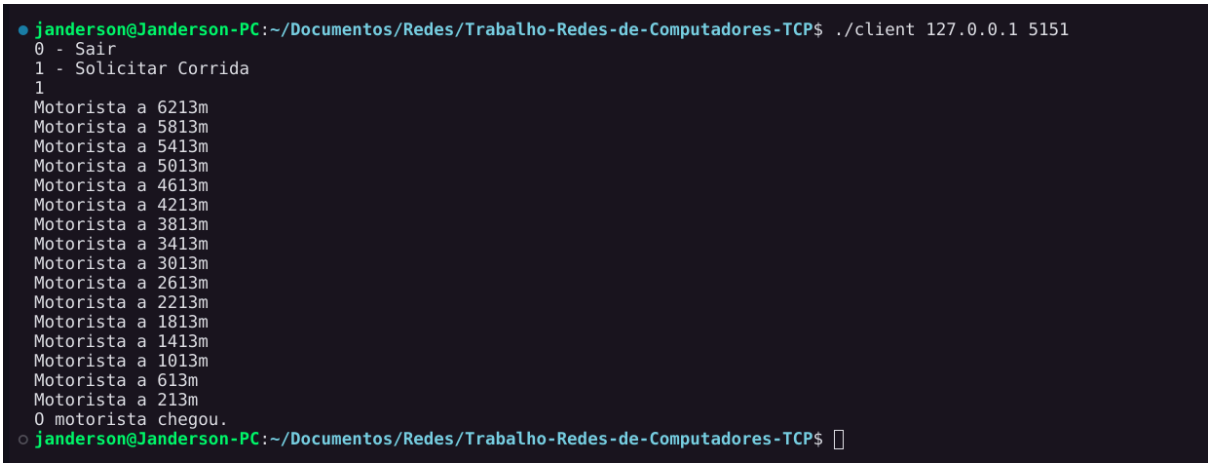
Motorista aguarda por um cliente

```
janderson@Janderson-PC:~/Documentos/Redes/Trabalho-Redes-de-Computadores-TCP$ ./client 127.0.0.1 5151
0 - Sair
1 - Solicitar Corrida
1
```

Cliente solicita corrida

```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS
janderson@Janderson-PC:~/Documentos/Redes/Trabalho-Redes-de-Computadores-TCP$ ./server ipv4 5151
Aguardando Solicitação.
Corrida Disponivel:
0 - Recusar
1 - Aceitar
1
```

Motorista/Servidor aceita corrida e a partir desse momento é criado uma thread para que aconteça a comunicação com o cliente, nesse momento o servidor volta ao estado de espera para que seja possível atender outros clientes. Após isso o servidor recebe as coordenadas enviadas pelo cliente .

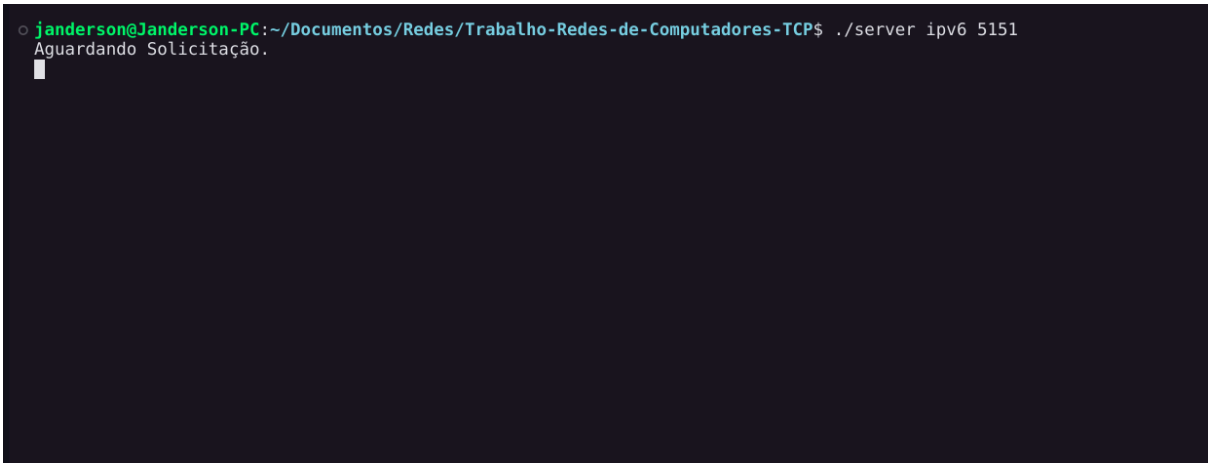


O cliente recebe as atualizações da distancia do motorista ate que o motorista chegue ao ponto de encontro combinado. Ao chegar no ponto de encontro é enviada uma mensagem “O motorista chegou”.

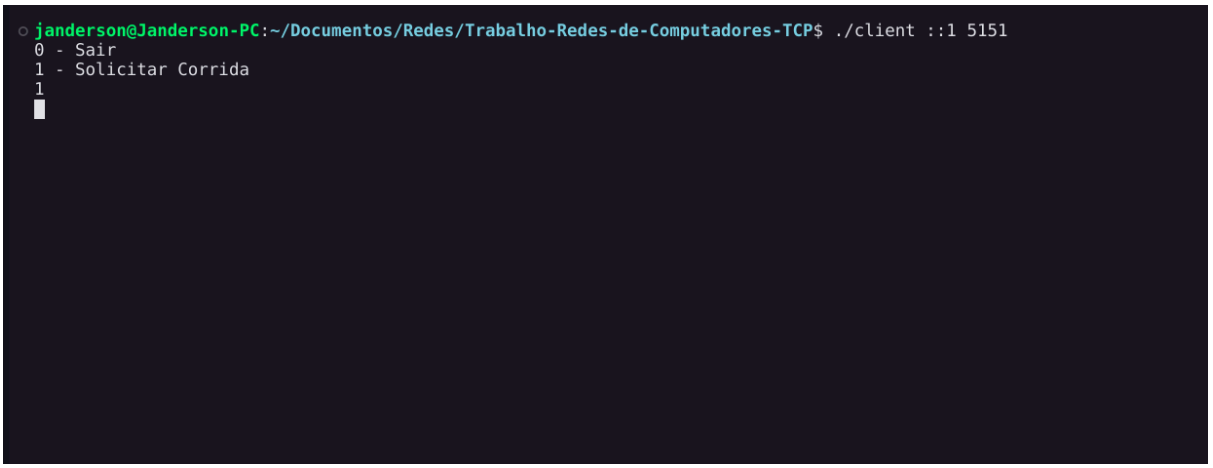
Da mesma forma, a mensagem é informada no servidor. Após o motorista chegar o servidor se mantém em estado de espera para que novas solicitações de outros clientes sejam possíveis.

3.1 Motorista Recusa Solicitação IPv6:

Abaixo será apresentado o mesmo fluxo dos exemplos com IPv6, os comentários serão omitidos já que em todo o fluxo o comportamento é idêntico para quando se é utilizado o IPv4.



Servidor é executado em ipv6



Cliente faz uma solicitação para um motorista no servidor em IPv6

Servidor recusa a solicitação

```
janderson@Janderson-PC:~/Documentos/Redes/Trabalho-Redes-de-Computadores-TCP$ ./client ::1 5151
 0 - Sair
 1 - Solicitar Corrida
 1
 Não foi encontrado um motorista
 0 - Sair
 1 - Solicitar Corrida
 0
janderson@Janderson-PC:~/Documentos/Redes/Trabalho-Redes-de-Computadores-TCP$
```

Cliente recebe a informação que a corrida não foi aceita e finaliza o programa.

3.1 Motorista Aceita Solicitação IPv6:

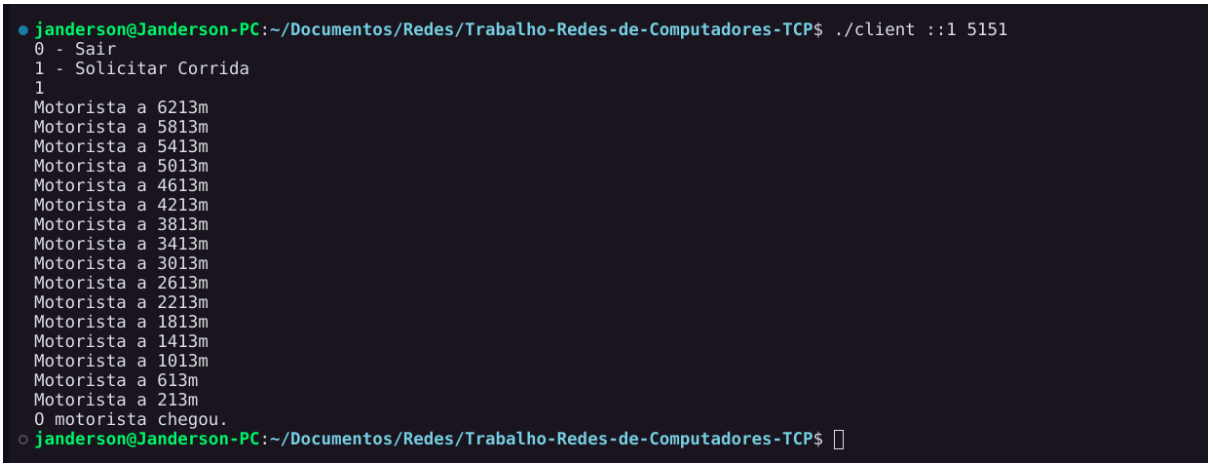
```
janderson@Janderson-PC:~/Documentos/Redes/Trabalho-Redes-de-Computadores-TCP$ ./server ipv6 5151
Aguardando Solicitação.
```

Servidor é iniciado e está apto a receber conexões

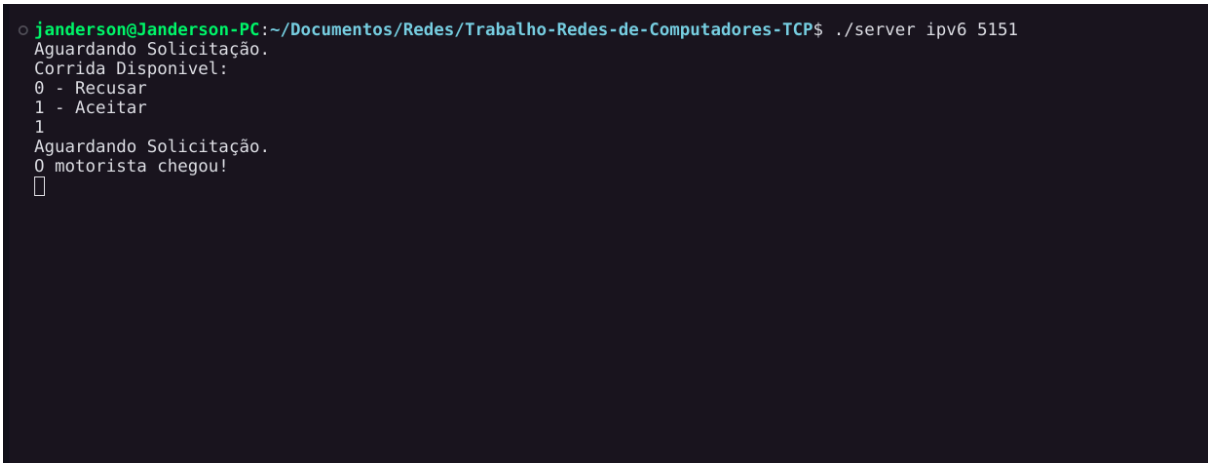
```
janderson@Janderson-PC:~/Documentos/Redes/Trabalho-Redes-de-Computadores-TCP$ ./client ::1 5151
 0 - Sair
 1 - Solicitar Corrida
 1
```

Cliente é iniciado e solicita corrida

Servidor aceita solicitação



Cliente recebe a atualização da localização do motorista e programa é finalizado



Servidor exibe mensagem quando alcança o cliente

2.Referências:

Função Haversine:
https://www.geeksforgeeks.org/haversine-formula-to-find-distance-between-two-points-on-a-sphere/

Play list Introdução à Programação em Redes por Italo Cunha:
https://www.youtube.com/watch?v=tJ3qNtv0HVs&list=PLyrH0CFXIM5Wzmbv-IC-qvoBejsa803Qk&index=1