

TRABALHO PRÁTICO 1

Programação socket

Tarcizio Augusto Santos Lafaiete

Universidade Federal de Minas Gerais(UFMG)
Belo Horizonte - MG - Brasil

tarcizio-augusto@hotmail.com

1.Introdução

O problema proposta para implementação neste trabalho prático é uma aplicação console que simula o funcionamento básico do Uber, para este objetivo deve-se utilizar os conceitos de programação em socket aprendidos na disciplina para implementar um cliente e um servidor utilizando o protocolo TCP/IP. A aplicação se inicia com a inicialização do servidor e a conexão do cliente, em seguida caso o cliente solicite uma corrida e o servidor aceita esta, o cliente então envia suas coordenadas para ele e assim ele calcula a distância entre eles atualizando ela em 400m a cada 2000ms.

2.Implementação

Nesta seção será explicado a implementação do servidor e do cliente em C de forma detalhada.

2.1 Configurações de socket

Neste primeiro subtópico será explicado funções que são utilizadas tanto pelo cliente como pelo servidor, assim facilitando o entendimento das seguintes subseções. Estas funções estão implementadas em socketUtils.h.

```
#define IPV4 0
#define IPV6 1
#define UNK 2

#define CLOSE_CLIENT (char)0xC1
#define REQUEST_DRIVE (char)0xC2
#define REQUEST_DRIVE_ACCEPTED (char)0xC3
#define REQUEST_DRIVE_REFUSED (char)0xC4
#define ACKNOWLEDGE_REQ (char)0xC5
#define DRIVE_FINISH_REQ (char) 0xEF

typedef struct{
    int type;
} typeIP;

typedef struct{
    struct sockaddr_in addr;
    struct sockaddr_in6 addr6;
    socklen_t len;
} socket_address;

typeIP translateIP(char* ip);
socket_address configure_addr(typeIP inet, int port);
void create_socket(int* socket_fd, typeIP inet);

const char* receiveMessage(int socket);
void sendMessage(const char* msg, int socket);
```

Figura 1 - Arquivo socketUtils.h

traslatelP: Nesta função implementa-se uma simples conversão entre o argumento de entrada para o tipo do ip para uma struct chamada *typeIP* que carrega um dos valores definidos pelos *defines* IPV4, IPV6 ou UNK.

configure_addr: O *configure_addr* recebe como parâmetro um *typeIP* e uma porta, a partir disso, ele configura a struct *socket_address*. Sendo que, caso o *typeIP* seja IPV4 ele configura a família, a porta e o endereço da struct *sockaddr_in* presente em *socket_address*. Já no caso de ser IPV6 ele configura os mesmos parâmetros porém da struct *sockaddr_in6*, e assim retorna a struct *socket_address*. Importante ressaltar que em ambos os casos o endereço passado é o localhost.

create_socket: Aqui passa-se como parâmetro, o file descriptor do socket por referência e o *typeIP*, com isto cria-se um descritor para o socket utilizando a função *socket()*, esta função recebe como para um *_type*, um *_domain* e um *_protocol*, neste caso o *type* corresponde ao IP e por isso utiliza-se o *typeIP* para definir o socket como AF_INET no caso de ipv4 e AF_INET6 no caso de ipv6. O *_domain* é definido por SOCK_STREAM, uma vez que estamos utilizando tcp na comunicação. Por fim passamos 0 para *_protocol* em ambos os casos por padrão. Abaixo, na figura 2 pode-se observar a sua implementação.

```
void create_socket(int* socket_fd, typeIP ip){  
  
    if(ip.type == IPV4){  
        *socket_fd = socket(AF_INET, SOCK_STREAM, 0);  
    }  
    else if(ip.type == IPV6){  
        *socket_fd = socket(AF_INET6, SOCK_STREAM, 0);  
    }  
    else{  
        *socket_fd = -1;  
    }  
  
    if(*socket_fd < 0){  
        printf("Nao foi possivel criar o socket \n");  
        exit(EXIT_FAILURE);  
    }  
}
```

Figura 2 - Função *create_socket*

receiveMessage: Esta função recebe como parâmetro o file descriptor do socket, que cria um buffer de tamanho igual a uma constante *max_message_size*, em seguida utiliza a função *read*, para ler preencher o buffer com dados recebidos no socket.

sendMessage: Recebe uma string e o descritor do socket como parâmetros e internamente chama a função *send()* passando como parâmetros o descritor, a string e o tamanho da mensagem, para assim está função enviar através da conexão estabelecida os dados desejados.

2.2 Servidor

Em seguida, será apresentado a implementação do servidor e o seu fluxo de mensagens com o cliente, seguindo como base a implementação feita no arquivo

server.c. Para entendermos melhor o funcionamento do servidor deve-se observar como é o seu funcionamento básico como demonstrado na imagem abaixo:



State diagram for server and

Figura 3 - Operação do Servidor

Em *server.c* implementamos estes procedimentos com ajuda das funções do arquivo *serverCore.h*, estas funções utilizam as chamadas nativas de socket do linux e abstraem alguns detalhes da mesma. O fluxo principal do servidor pode ser visto em alto nível na main de *server.c* como mostrado abaixo:

```

int main(int argc, char* argv){

    int* sock = initServer(argv);

    while(1){
        const char* firstRequest = receiveMessage(sock[1]);
        if(firstRequest[0] == CLOSE_CLIENT){
            close_server(sock[0], sock[1]);
            sock = initServer(argv);
        }
        else{
            int drive_situation = acceptDrive(sock);
            if(!drive_situation){
                printWaitingRequest();
            }
            else{
                Coordinate coordClient = rcvClientCoordinate(sock);
                double distance = haversine(coordClient, coordServ);
                updateDistance(sock, distance);
            }
        }
    }

    return 0;
}

```

Figura 4 - Main de server.c

O programa se inicia com a chamada da função `initServer`, que configura o servidor e retorna os file descriptors do socket e da conexão estabelecida, para compreender melhor esta inicialização, pode-se observar a figura abaixo:

```

int* initServer(char* argv){
    serverCore core;

    core.inet = translateIP(argv[1]);

    create_socket(&core.server_fd, core.inet);
    core.socket = configure_addr(core.inet, atoi(argv[2]));
    configure_options(core.server_fd);
    binding(&core);
    listening(core.server_fd);

    printWaitingRequest();

    int conn_sock = acceptConnection(&core);
    if(conn_sock < 0){
        printf("Sem conexao no aceita \n");
        exit(EXIT_FAILURE);
    }

    int* fd = (int*)malloc(sizeof(int) * 2);
    fd[0] = core.server_fd;
    fd[1] = conn_sock;

    return fd;
}

```

Figura 5 - Função `initServer` em server.c

Em `initServer` cria-se uma struct chamada `serverCore`, esta struct possui em si um `server_fd` que guarda o file descriptor do socket, uma struct `socket_address` e um `typeIP`. Cria-se então um socket, utilizando a função `create_socket` descrita na subseção anterior, em seguida realiza-se a configuração do endereço do socket e

realiza-se algumas configurações da conexão utilizando a função `configure_options`, que abstrai a função `setsockopt` que na aplicação em questão configura no servidor o reuso do endereço e da porta para o socket pertencente ao servidor.

Prosseguindo no procedimento de abertura do socket, utiliza-se a função *binding* que internamente chama a função *bind* que é utilizada para associar as configurações presentes em *socket_address* ao socket que foi criado pelo servidor. Com a associação feita, chama-se a função *linstening* que abstrai a chamada nativa *listen* utilizada para permitir a escuta de conexões por parte do socket. Por fim, em *acceptConnection* abstrai-se a chamada de *accept*, que é utilizado para permitir conexões entre o servidor e um cliente, está chamada *accept* é bloqueante, ou seja o código fica travado nesta até que uma conexão seja estabelecida e retorna um descritor, que é diferente do descritor do socket, para aquela conexão estabelecida. Por fim, retorna-se um vetor de 2 posições contendo os descritores do socket e da conexão.

Com toda a camada de configuração e abertura de conexão do servidor estabelecida, ele está apto para trocar mensagens com o cliente, utilizando as funções *receiveMessage* e *sendMessage* explicadas anteriormente. Prosseguindo na função *main*, nota-se um loop *while* infinito na qual é realizado a troca de mensagens entre as partes, inicialmente em qualquer momento da comunicação caso o servidor venha a receber uma requisição `CLOSE_CLIENT` ele fecha o servidor, utilizando a função *close_server* que internamente utiliza a função *close* para fechar a conexão e o socket utilizando o file descriptor de ambos. Para em seguida reinicializar o servidor para novas conexões com clientes.

Além do fechamento do servidor as trocas de mensagem se seguem no seguinte fluxo, caso o cliente envie uma requisição de corrida, o servidor tem como opção aceitar ou não está, em ambos os casos o servidor comunica a sua decisão através dos request's `REQUEST_DRIVE_ACCEPTED` em caso positivo e `REQUEST_DRIVE_REFUSED` em caso de negativa. No caso em que o servidor aceita a proposta de corrida, ele recebe do servidor a sua latitude e longitude, e entre meio estas, apenas para manter a estrutura ping-pong da comunicação, ele envia uma `ACKNOWLEDGE_REQ` para o cliente.

Com as coordenadas recebidas, o servidor inicia o seu processo de atualização da distância entre ele e o cliente, sendo assim, a cada 2 segundos o servidor envia para o cliente uma mensagem avisando que ele se encontra 400 metros mais próximo do solicitante e a cada envio de mensagem é recebido como retorno um `ACKNOWLEDGE_REQ`, até que a distância seja menor que os 400 metros, neste momento a mensagem “*O motorista chegou!*” é enviada, um `ACK` é recebido e a requisição `DRIVE_FINISH_REQ` é enviada para o cliente, notificando o fim da viagem, neste momento o cliente envia o request `CLOSE_CLIENT` e a conexão é finalizada e o servidor fica disponível para uma nova conexão.

2.3 Cliente

Explicado o funcionamento do servidor, é chegado o momento de explicar o funcionamento do cliente da aplicação, abaixo segue uma figura demonstrando o funcionamento básico do cliente:



Figura 6 - Operação do cliente

No arquivo *client.c* implementamos este funcionamento com o auxílio de algumas funções de *clientCore.h* que abstraem as chamadas de função nativa da programação em socket do Linux. Abaixo pode-se observar o fluxo principal do cliente na aplicação:

```

int main(int argc, char* argv[]){
    int socketClient = initClient(argv);

    int mainClientLoop = 1;
    while(mainClientLoop){

        int acceptResult = acceptProcess(socketClient);
        if(acceptResult == 0){
            close_client(socketClient);
            printEndOfRide();
            return 0;
        }

        char c = REQUEST_DRIVE;
        sendMessage(&c, socketClient);
        const char* response = receiveMessage(socketClient);
        if(response[0] == REQUEST_DRIVE_ACCEPTED){
            mainClientLoop = 0;
            sendClientCoordinate(socketClient);
            break;
        }
        else{
            printNotFoundDrive();
        }
    }

    driveRoutine(socketClient);

    close_client(socketClient);

    printEndOfRide();

    return 0;
}

```

Figura 7 - Main de *client.c*

Na Main de *client.c* pode-se observar uma inicialização semelhante ao do servidor com a função *initClient* que está exposta abaixo:

```

int initClient(char* argv[]){
    clientCore core;

    core.inet = translateIP(argv[1]);

    create_socket(&core.client_fd, core.inet);
    core.serverSocket = configure_addr(core.inet, atoi(argv[3]));

    connect_client(&core);

    return core.client_fd;
}

```

Figura 8 - Função *initClient* em *client.c*

Nesta inicialização do cliente começa-se primeiro criando um *clientCore* que possui em si um *client_fd* guardando o file descriptor do cliente, um *socket_address* com as configurações do servidor e um *typeIP*. Primeiramente, chama-se a função *create_socket* que gera um descritor para o cliente, em seguida utiliza-se o *configure_addr* para gerar as configurações do servidor a qual se deseja comunicar e chama a função *connect_client* que abstrai a chamada nativa *connect*, nesta chamada tem-se como parâmetros o descritor do socket cliente e as configurações do servidor, através dessa chamada caso aquele servidor especificado estiver

esperando uma conexão, ou seja, estiver na fase de *accept*, uma conexão será estabelecida e mensagens poderão ser trocadas entre os lados da aplicação por meio das funções *receiveMessage* e *sendMessage*. Por fim, a *initClient* retorna o descritor do cliente.

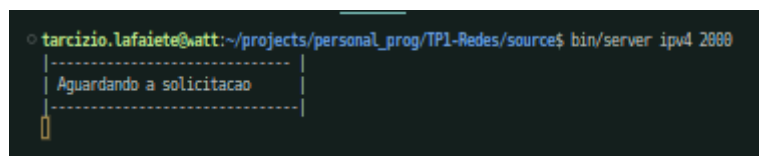
Após o retorno de *initClient*, segue-se na main para o seu loop principal, este loop consiste em duas fases. Na primeira fase, tem-se o processo de solicitação de corrida, na qual o cliente tem a opção de requisitar uma corrida, ou sair da aplicação, caso esta seja opção escolhida é chamada a função *close_client*, que internamente envia para o servidor a requisição CLOSE_CLIENT avisando do encerrado da conexão e em seguida chama a função nativa *close* para fechar o socket.

No caso de ser realizado uma solicitação é enviado ao servidor um REQUEST_DRIVE e avança-se para a segunda fase do loop, este request é respondido pelo servidor com as requisições mostradas na subseção acima, sendo que caso a resposta seja negativa volta-se para a primeira fase do loop e em caso positivo, o cliente envia as suas coordenadas para o servidor e o loop é encerrado, assim avança-se para o processo de notificação sobre o andamento da corrida, como explicado na parte do servidor. No lado do cliente este processo consiste no recebimento de atualizações sobre a viagem que são prontamente respondidos com um ACK, até que o cliente receba a requisição DRIVE_FINISH_REQ, indicando o fim da viagem. Finalizada então a corrida, o cliente chama a função *close_client*, fechando a conexão com o servidor e terminando a sua execução.

3.Funcionamento

Neste tópico tem-se a intenção de demonstrar o funcionamento do para todos os casos possíveis, a fim de validar que a implementação feita no trabalho está condizente com as especificações deste feitas pelo professor. Como a aplicação foi pensada para funcionar no terminal, algumas funções foram criadas para facilitar a execução, estas podem ser encontradas em *terminalPrinter.h*.

Primeiramente, para iniciarmos a aplicação é necessário inicializar o servidor que exibe a seguinte mensagem:

A terminal window with a dark background. The prompt is 'tarcizio.lafaiete@watt:~/projects/personal_prog/TP1-Redes/source\$'. The command 'bin/server ipv4 2888' has been executed. Below the command, a dashed rectangular box contains the text 'Aguardando a solicitacao' in a light blue color. A cursor is visible at the bottom left of the box.

```
tarcizio.lafaiete@watt:~/projects/personal_prog/TP1-Redes/source$ bin/server ipv4 2888
|-----|
| Aguardando a solicitacao |
|-----|
|
```

Figura 9 - Servidor aguardando conexão do cliente

Em seguida o cliente se conecta ao servidor e exibe esta mensagem:

```
o tarczio.lafaiete@watt:~/projects/personal_prog/TP1-Redes/source$ bin/client ipv4 127.0.0.1 2000
|-----|
| 0 - Sair |
| 1 - Solicitar Corrida |
|-----|
Solicitacao de corrida: [ ]
```

Figura 10 - Cliente tem a conexão aceita pelo servidor

A partir daqui tem-se algumas rotas que a aplicação pode tomar e serão demonstradas caso a caso nos subtópicos desta seção.

3.1. Saída do cliente

Nesta rota de execução do programa o cliente ao receber a opção entre sair ou solicitar uma corrida ele opta por sair da aplicação, com isso o cliente é fechado e o servidor fica liberado para uma nova conexão, como pode-se ver nas imagens abaixo:

```
o tarczio.lafaiete@watt:~/projects/personal_prog/TP1-Redes/source$ bin/client ipv4 127.0.0.1 2000
|-----|
| 0 - Sair |
| 1 - Solicitar Corrida |
|-----|
Solicitacao de corrida: 0
| <Programa encerrado> |
o tarczio.lafaiete@watt:~/projects/personal_prog/TP1-Redes/source$
```

Figura 11 - Cliente opta por fechar a aplicação

```
o tarczio.lafaiete@watt:~/projects/personal_prog/TP1-Redes/source$ bin/server ipv4 2000
|-----|
| Aguardando a solicitacao |
|-----|
| Aguardando a solicitacao |
|-----|
```

Figura 12 - Servidor abre nova conexão para outros clientes

3.2. Motorista recusa corrida

Outra possível rota é a recusa do motorista, o servidor, da corrida solicitada pelo cliente, neste caso o cliente decide por solicitar uma viagem ao motorista, assim que a requisição chega este tem a opção de aceitar ou não a viagem, no caso de não aceitar ele notifica o cliente, que por sua vez tem a possibilidade de requisitar uma corrida novamente. No caso do servidor ele retorna pro seu estado inicial esperando uma nova solicitação.

```
tarcizio.lafaiete@watt:~/projects/personal_prog/TP1-Redes/source$ bin/client ipv4 127.0.0.1 2000
-----
| 0 - Sair |
| 1 - Solicitar Corrida |
|-----|
Solicitacao de corrida: 1
█
```

Figura 13 - Cliente solicita a corrida

```
-----
| Aguardando a solicitacao |
|-----|
Corrida disponivel
| 0 - Recusar |
| 1 - Aceitar |
|-----|
Aceitar? █
```

Figura 14 - Servidor tem a oportunidade de aceitar ou não a solicitação

```
-----
| Corrida disponivel |
| 0 - Recusar |
| 1 - Aceitar |
|-----|
Aceitar? 0
-----
| Aguardando a solicitacao |
|-----|
█
```

Figura 15 - Servidor recusa a solicitação e volta a esperar novas solicitação

```
-----
| Nao foi encontrado um |
| motorista |
|-----|
| 0 - Sair |
| 1 - Solicitar Corrida |
|-----|
Solicitacao de corrida: █
```

Figura 16 - Cliente é avisado da recusa e retorna para o menu de solicitação

3.3. Corrida aceita

Por fim, a última possibilidade de funcionamento é no caso de que aconteça uma corrida. Neste caso, o servidor vai enviando atualizações para o cliente até finalmente o motorista chegar ao destino, quando isto ocorre o cliente é encerrado e o servidor fica apto para receber novas conexões.

```
-----
| Corrida disponivel |
| 0 - Recusar |
| 1 - Aceitar |
|-----|
Aceitar? 1
```

Figura 17 - Motorista aceita a solicitação

```
|-----|  
| Motorista a 1829.71 m |  
| Motorista a 1429.71 m |  
| Motorista a 1029.71 m |  
| Motorista a 629.71 m |  
| Motorista a 229.71 m |  
| O Motorista chegou! |  
|-----|  
| <Programa encerrado> |
```

Figura 18 - Prosseguimento da viagem vai sendo atualizada para o cliente

```
Aceitar? 1  
|-----|  
| O Motorista chegou! |  
|-----|  
| Aguardando a solicitacao |  
|-----|
```

Figura 19 - Motorista é notificado que chegou a local e servidor volta a esperar novas conexões

4. Conclusão

Por fim é importante ressaltar que o programa tem possibilidades de melhorias, como o uso de threads para permitir mais conexões e tratamentos de erros mais refinados para certos problemas de conexão, no geral este trabalho prático foi um excelente exercício para ter um contato melhor com os conteúdos vistos na disciplina e ter mais experiências com bibliotecas de socket.

Em geral, a programação do socket foi de certo modo simples, tendo uma vasta gama de materiais de consulta na internet e também com a apresentação disponibilizada pelo professor sobre o tema, facilitou o entendimento geral dos desafios do problema.