Eduardo Maximiano de Oliveira

Trabalho de Redes

Dourados, MS
Julho 2019

Sumário

1 DESCRIÇÃO DO PROBLEMA 2
1.1 Servidor 2
1.2 Cliente 2

2 DESCRIÇÃO DOS ALGORITMOS 3
2.1 Dados 3
2.2 Algoritmos 3
2.2.1 Algoritmo de Haversine 3
2.2.2 Checksum 4
2.3 Principais Funções no Servidor 4
2.3.1 Função: abrindoServidor() 4
2.3.2 Função: novaPorta() 4
2.3.3 Função: portaAleatoria() 5
2.3.4 Função: clienteNovo() 5
2.3.5 Função: pacoteDuplicado() 5
2.3.6 Função: grava_Arq() 5
2.3.7 Função: verificaDados() 5
2.3.8 Função: arquivoVazio() 6
2.3.9 Função: pesquisa_Arq() 7
2.4 Principais Funções no Cliente 8
2.4.1 Função: novaPorta() 8
2.4.2 Função: transmissao() 8
2.4.3 Função: esperaACK() 8
2.5 Decisões de implementação 8

3 RETRANSMISSÃO DE DADOS 9

4 TESTES 10
4.1 Dados Validos 10

5 REFERÊNCIAS 12

1 Descrição do Problema

O problema consiste em desenvolver um sistema capaz de enviar preços de vários postos de combustíveis e solicitar o preço mais barato de uma certa área. O sistema será dividido em dois programas: um servidor e um cliente.

1.1 Servidor

O servidor será responsável por receber os dados do cliente e fazer uma das duas operações: gravar no arquivo ou retornar o posto com o combustível mais barato na área solicitada.

1.2 Cliente

O cliente poderá enviar mensagens de dados que serão salvas em um arquivo pelo servidor ou fazer uma pesquisa para procurar o posto com o menor preço para aquele combustível na região.

2 Descrição dos Algoritmos

2.1 Dados

As mensagens trocadas entre o servidor e cliente ficam salvas em uma estrutura chamada **Pacote**, que tem os seguintes atributos:

```
typedef struct{
    //salva o cheksum
    unsigned char cheksum;
    //Salva o indicador atual da mensagem
    int indicador;
    //Salva os dados a serem enviados
    char mensagem[100];
} Pacote;//Struct que envia e recebe dados do cliente
```

Figura 1 – Estrutura do pacote

- checksum: Salva o resultado do checksum do pacote
- indicador: Representa o ACK do pacote que muda entre 0 e 1
- mensagem: Os dados que serão trocados entre o servidor e cliente

2.2 Algoritmos

2.2.1 Algoritmo de Haversine

Esse algoritmo foi pego em site com o autor Jaime. Ele calcula a distância entre duas coordenadas. Sua formula é a seguinte:

- R = raio da terra (6,371km)
- $lat = lat2 - lat1$
- $long = long2 - long1$
- $a = sin^2(lat/2) + cos(lat1) * cos(lat2) * sin^2(long/2)$
- $c = 2 * atan2(sqrt(a), sqrt(1 - a))$
- $d = R * c$

```
//calcula a distancia entre as coordenadas
double Distancia(double latitude1, double longitude1, double latitude2, double longitude2){
    int RaioDaTerra = 6371; //Raio da terra em kilometros
    //Pega a diferença entre dois pontos
    //e converte para radianos
    double nDLat = (latitude2 - latitude1) * (PI/180);
    double nDLon = (longitude2 - longitude1) * (PI/180);
    double nA = pow ( sin(nDLat/2), 2 ) + cos(latitude1) * cos(latitude2) * pow ( sin(nDLon/2), 2 );
    //A função atan2(double y, double x), retorna o arco tangente entre y/x
    double nC = 2 * atan2( sqrt(nA), sqrt( 1 - nA ));
    double nD = RaioDaTerra * nC;

    return nD; //Retorna a distância entre os pontos
}
```

Figura 2 – Algoritmo de distância

A diferença entre as latitudes e longitudes foram convertidas para radianos, multiplicando o resultado por (PI/180).

2.2.2 Checksum

O algoritmo abaixo foi tirado de um *post* do site *stackoverflow* e nele é feita a subtração de cada caracter em uma variável do tipo *unsigned char*. Ao chegar no servidor é feita a mesma soma com o pacote recebido e se ele tiver corrompido o resultado será diferente da soma feita no cliente.

2.3 Principais Funções no Servidor

2.3.1 Função: abrindoServidor()

Ao chamar essa função o servidor é aberto e fica esperando dados dos clientes dentro de um *While*. Os dados que forem recebidos pelo *recvfrom* desse *While* indica a primeira conexão do cliente, e então será aberto um processo novo para esse cliente, o *socket* atual será fechado e um novo será aberto para o processo atual, e também irá fazer a troca de portas entre esse processo e o cliente. Essa troca é necessária para criar um processo novo por cliente, caso não fizer a troca será feito um processo novo para cada mensagem.

2.3.2 Função: novaPorta()

Essa função é responsável pela troca de portas entre o processo e o cliente. Ela recebe uma porta aleatória da função *portaAleatoria()* e manda para o cliente e em seguida atualiza sua própria porta.

2.3.3 Função: portaAleatoria()

Essa função é responsável por retornar uma porta disponível para ser enviada para o cliente. Para pegar esse porta é preciso definir o *sin_port* da estrutura *sockaddr_in* como zero, e em seguida realizar um *bind*, com isso o SO irá pegar uma porta aleatória e realizar o *bind*. Depois de pegar a porta, fechamos o *socket* para libera-la.

2.3.4 Função: clienteNovo()

Aqui está o corpo principal da comunicação entre cliente e servidor. Uma vez trocadas as portas o cliente irá se comunicar somente no *recvfrom* desta função até encerrar sua conexão.

2.3.5 Função: pacoteDuplicado()

Compara se o pacote atual é igual ao anterior, se for descarta o pacote atual e caso a mensagem for de pesquisa manda a resposta para o servidor.

2.3.6 Função: grava_Arq()

São utilizados três arquivo para os dados, gasolina, álcool e diesel. Para saber qual usar é feita a verificação do tipo de combustível da mensagem para abrir o arquivo correspondente e em seguida gravar.

2.3.7 Função: verificaDados()

Verifica se os dados recebido estão no formato correto, se tiver algo errado ele retorna um aviso para o cliente. Abaixo tem os dois formatos ideais:

- P 2 20 -22.2218 -54.8064
  - P: indica mensagem de pesquisa
  - 2: indica o tipo de combustível
    - 0-diesel
    - 1-álcool
    - 2-gasolina
  - 20: indica o raio de busca
  - -22.2218: latitude
  - -54.8064: longitude
- D 0 2540 -22.2218 -54.8064

  - D: indica mensagem de dados, para gravar
  - 0: indica o tipo de combustível
    * 0-diesel
    * 1-álcool
    * 2-gasolina
  - 2540: preço do combustível multiplicado por 1000
  - -22.2218: latitude
  - -54.8064: longitude

Qualquer dado fora desse formato será invalido e o cliente será informado. Os formatos abaixo estão errados e foram testados:

- D -0 4439 -21.22182 -54.8064
- D 0 -4439 -21.22182 -54.8064
- D 0 44.39 -21.22182 -54.8064
- d 0 4439 -21.22182 -54.8064
- D 0 4439 -21.2.2182 -54.80.64
- D 0 4439 -21.22182 -54.8064 *tem mais de um espaço entre as partes
- P -0 20 -21.22182 -54.8064
- P 0 -20 -21.22182 -54.8064
- P 0 20.0 -21.22182 -54.8064
- p 0 20 -21.22182 -54.8064
- P 0 20 -21.2.2182 -54.80.64
- P 0 20 -21.22182 -54.8064 *tem mais de um espaço entre as partes

2.3.8 Função: arquivoVazio()

Verifica se o arquivo em que será feito a pesquisa está vazio. Para verificar o arquivo usa a função *fseek()* para ir até o final do arquivo, em seguida a função *ftell()* que retorna o valor da posição atual, se o valor for zero o arquivo está vazio.

2.3.9 Função: pesquisa_Arq()

A função começa abrindo o arquivo correspondente ao tipo de combustível da mensagem, em seguida pega a mensagem e quebra em partes para salvar em variáveis separadas e fazer comparações. O mesmo acontece com os dado do arquivo, lê uma linha inteira e quebra em partes. Antes de ler o arquivo é definido variáveis que irão pegar os dados do posto com menor preço, uma variável *precom* é declarada com valor alto só para a primeira comparação. A primeira comparação é para ver se o posto está no raio de busca do cliente, se a distância for menor que o raio ele está na área e é feito a comparação de preços em busca do menor.
Se não tem posto no raio de busca ele retorna para o cliente que não há postos na região, caso contrario ele imprime as informações no servidor e em seguida manda para o cliente.

2.4 Principais Funções no Cliente

2.4.1 Função: novaPorta()

Manda um pacote para o servidor pedindo uma nova porta e espera a resposta do servidor. Quando trocar as portas o cliente irá se comunicar com o processo responsável por ele no servidor.

2.4.2 Função: transmissao()

Função responsável por enviar os dados para o servidor. Pode digitar a palavra *"exit"* para fechar o cliente e finalizar o processo no servidor. O cliente pode mandar qualquer coisa, o servidor que decide se está correto ou não. Após o envio da mensagem ele vai para a função *esperaACK()*.

2.4.3 Função: esperaACK()

Espera o *ACK* do servidor e verifica o pacote que foi recebido. Começa esperando o *ACK* do servidor, tempo de espera é de 1 segundo e 250 milissegundos ou 1,25 segundos, se o tempo se esgotar ele envia o pacote de novo, fazendo no até no máximo 4 retransmissões. Após receber o *ACK* ele verifica a mensagem do pacote. A seguir está as operações realizadas caso tenha algum erro, se não houver erros irá chegar na última opção, a ordem é a mesma que no código:

- Dados no formato incorreto: para a retransmissão dos dados
- *ACK* diferentes: reenvia o pacote
- *Checksum* incorreto: reenvia o pacote
- Pacotes duplicados: para a retransmissão e espera o resultado se for uma pesquisa
- Nenhum erro no envio, se for pesquisa espera o resultado do servidor

2.5 Decisões de implementação

O programa servidor utiliza processos para tratar cada cliente. Usa a função *fork()* para criar um processo novo e deixa-lo disponível para o cliente.

3 Retransmissão de dados

Para tratar a retransmissão de dados, foi usado o *rdt3.0*, protocolo com bit alternante. O cliente envia o pacote para o servidor com um ACK, 0 ou 1, e espera resposta do servidor. Quando o pacote chega no servidor ele verifica se o pacote está correto e se estiver manda o ACK de volta para o cliente. O cliente possui um tempo de espera de 1.25 segundos, se o tempo se esgotar ele manda novamente o pacote, a retransmissão é feita até quatro vezes, se mandar pela quarta vez e não receber o ACK é cancelado o envio. Se o ACK recebido for diferente do esperado, será feita a retransmissão. Se o servidor recebeu o pacote, mandou o ACK mas não chegou no cliente, o cliente irá mandar outro pacote, e assim terá pacotes duplicados no servidor. O servidor detecta a duplicação e descarta esse pacote. Independente dos erros na transmissão, o cliente só irá fazer quatro retransmissões. A imagem abaixo mostra parte da função *esperaACK()* responsável pela retransmissão caso passe do tempo limite.

Figura 3 – Função: esperaACK()

```
//Se nao receber o ACK dentro do tempo limite manda denovo os dados
void esperaACK(int clienteSocket, struct sockaddr_in servidor,socklen_t len,Pacote pacote, Pacote resposta,int numRet){
    struct timeval timeout;
    timeout.tv_sec = 1;//1 Segundo
    timeout.tv_usec = 250000;//250000 microsegundos = 250 milisegundos
    numRet++;
    if(numRet < 5){
        puts("Esperando ACK!!");
        if(setsockopt(clienteSocket,SOL_SOCKET,SO_RCVTIMEO,&timeout,sizeof(timeout)) < 0){
            perror("Erro");
        }
        //Espera o ACK do servidor
        if(recvfrom(clienteSocket,&resposta,sizeof(Pacote),0,(struct sockaddr *)&servidor,&len) < 0){
            //Se entra aqui é porque não recebeu resposta do servidor
            if((numRet+1) == 5){
                puts("Retransmissão limite alcançada. Envio cancelado!!");
            }else{
                puts("Tempo esgotado. Enviando outro pacote!!");
                // sendto(clienteSocket,&pacote,sizeof(Pacote),MSG_CONFIRM,(const struct sockaddr *)&servidor,sizeof(servi
                enviar(pacote,clienteSocket,servidor,len);
                esperaACK(clienteSocket,servidor,len,pacote,resposta,numRet);
            }
        }else{
            //Se entrar aqui é porque recebeu o ACK do servidor
```

A função trabalha de forma recursiva, pois caso tenha retransmissão será necessário esperar o ACK novamente, então a função é chamada novamente.

4 Testes

Abaixo tem os testes realizados e os resultados do servidor e do cliente:

4.1 Dados Validos

Os testes abaixo foram executados com os seguintes comandos: *./servidor 5000* e *./cliente 172.26.4.142 5000*

- Teste 1: D 0 3396 -22.546 -54.336

Figura 4 – Servidor

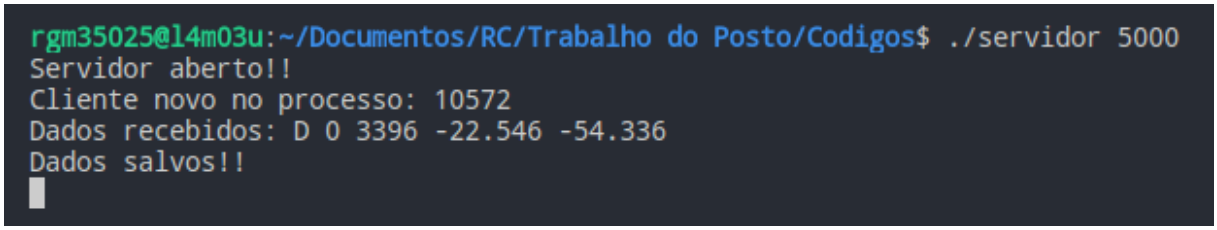
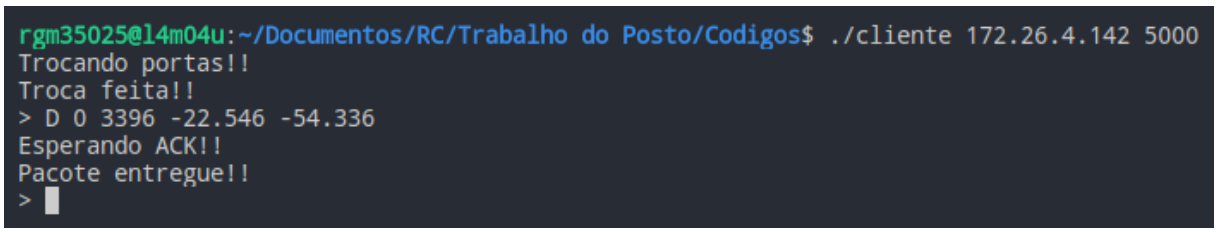


Figura 5 – Cliente



- Teste 2: P 0 25 -22.22 -54.79

Figura 6 – Servidor

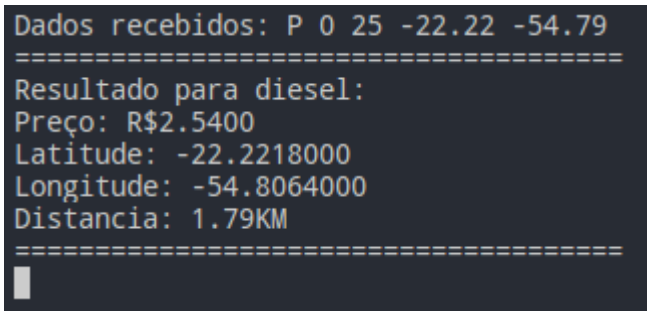
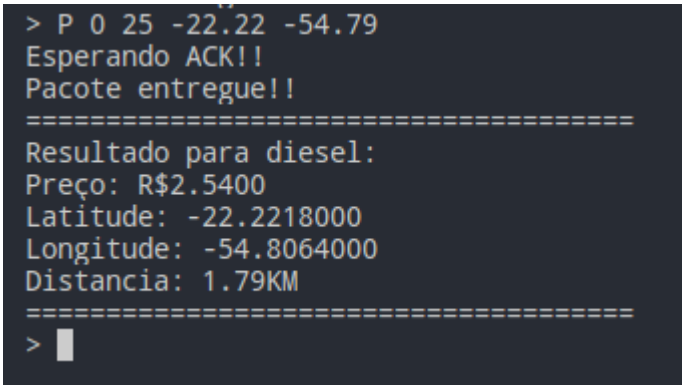


Figura 7 – Cliente



5 Referências

Uso do frok() para a criação de processos:
https://www.geeksforgeeks.org/creating-multiple-process-using-fork/
Construção do servidor e cliente:
https://www.geeksforgeeks.org/udp-server-client-implementation-c/
https://github.com/hzxie/Multiplex-Socket
Algoritmo da Fórmula de Haversine:
http://www.jaimerios.com/?p=39
Algoritmo do Checksum:
https://stackoverflow.com/questions/3463976/c-file-checksum
Uso e manipulação de arquivos:
https://www.tutorialspoint.com/cprogramming/c_file_io