Universidade Estadual de Mato Grosso do Sul
Curso de Ciência da Computação
Disciplina de Redes de Computadores
Trabalho de Redes de Computadores

IMPLEMENTAÇÃO DE UMA APLICAÇÃO DE SERVIÇOS DE COTAÇÃO DE PREÇOS DE COMBUSTÍVEIS

Sizenando S. França
RGM: 50575

Rio Brilhante (MS), 29 de julho de 2025

1. SUMÁRIO DO PROBLEMA A SER TRATADO

O trabalho teve como objetivo o desenvolvimento de uma aplicação cliente-servidor para um serviço de cotação de preços de combustíveis, a linguagem para a implementação foi deixada em aberto e por isso, a linguagem escolhida para o determinado objetivo foi C++ usando a biblioteca de Sockets sobre o protocolo UDP devido a maior familiaridade apresentada em sala.

O sistema é composto por dois programas principais: um **servidor** que é responsável por receber, armazenar e processar dados de postos de combustíveis; e um **cliente**, que permite aos usuários enviarem novas informações de preços quanto realizar consultas para encontrar o combustível desejado mais barato em uma determinada região geográfica dentro de um raio escolhido.

O principal desafio do projeto foi a implementação de um protocolo de transferência de dados confiável, ou RDT 2.0 sobre o UDP, que naturalmente não possui garantia nenhuma da entrega de pacotes. Para isso, foi desenvolvido um mecanismo de confirmações positivas (ACKs) ou negativas (NAKs) como informado no documento do trabalho, também a retransmissão por timeout o que garante a integridade da comunicação entre cliente e servidor.

2. DESCRIÇÃO DE ALGORITMOS E TIPOS DE DADOS

Para a construção do sistema, foram utilizados tipos abstratos de dados para modelar a comunicação e algoritmos específicos para processar as requisições.

2.1. TIPOS ABSTRATOS

As seguintes `structs` foram definidas para padronizar a troca de informações:

- `struct Pacote:` Unidade principal de comunicação enviada pelo cliente:

    - `tipo_msg (char):` Possui dois tipos, ‘D’ para Dados e ‘P’ para Pesquisa.
    - `id_msg (int):` Identificador para cada pacote, essencial para o rastreamento dos ACKs/NAKs.
    - `erro (bool):` Um campo para a simulação da corrupção dos pacotes, o que força o servidor enviar um NAK.
    - `tipo_pacote (union):` O uso do union é usado para a otimização do uso da memória, o que permite que a `struct` armazene os dados de uma mensagem seja ela ‘D’ ou ‘P’, mas não as duas ao mesmo tempo:
        - `dados:` Possui os campos para o cadastro de um posto (tipo de combustível, preço, latitude e longitude).
        - `pesquisa:` Possui os campos para a consulta (tipo de combustível, raio, latitude e longitude do centro da busca).
- `struct Resposta:` Utilizado pelo servidor para enviar os ACKs e NAKs para as mensagens de tipo ‘D’:
    - `id_msg_original (int):` Copia o ID do pacote a qual a resposta se refere.
    - `is_nak (bool):` `true` se a resposta for um NAK e `false` ser for um ACK.
- `struct RespostaPesquisa:` Utilizado pelo servidor para enviar responder mensagens do tipo ‘P’:
    - `preco_encontrado (int):` Possui o menor preço encontrado dentro do raio. Pode assumir valores de controle para sinalizar status: `-1` se nenhum posto for encontrado e `-2` para indicar um NAK, ou seja, o pacote de pesquisa foi corrompido.
- `struct Posto:` Estrutura auxiliar utilizada no servidor para armazenar na memória os dados lidos do arquivo `.csv` antes de realizar uma busca.

2.2. ALGORITMOS E FUNÇÕES PRINCIPAIS

- Lógica do Cliente (Máquina de Estados Finitos – FSM): O cliente opera num loop de envio e espera (`while(!sucesso)`), que implementa a FSM do RDT 2.0.
    1. Montagem e Envio: O cliente monta a `struct Pacote` e envia ao servidor.
    2. Estado de Espera: O cliente bloqueia na chamada `recvfrom()`, aguardando uma resposta. Um timeout de 2 segundos foi configurado com `setsockopt` para evitar a espera infinita.
    3. Tratamento da Resposta: Ao receber uma resposta do servidor, o cliente verifica:
        - Se foi um timeout, aqui ele assume que o pacote se perdeu na rede e prepara a retransmissão do mesmo.
        - Se foi um NAK, ele prepara a retransmissão.
        - Se foi um ACK (ou uma resposta válida da pesquisa), o ciclo é considerado um sucesso e o cliente está pronto para a próxima entrada que o usuário decidir.
    4. Retransmissão: Caso houver timeout ou NAK, o campo `erro` do pacote será configurado para `false` antes do reenvio para garantir que a próxima tentativa seja um pacote “limpo”.
- Lógica do Servidor (Loop principal): O servidor opera num loop infinito, permitindo a conexão de vários clientes mas operando apenas um por vez de forma iterativa como proposto no documento do trabalho.
    1. Recebimento e Conversão: Aguarda um pacote com `recvfrom` e usa o `memcpy` para converter o buffer de bytes recebido pelo cliente de volta para a `struct Pacote`.

2. Verificação de Erro: A primeira verificação feita é no campo `pacote.erro`. Se for `true`, o servidor envia a resposta de falha apropriada, ou seja, um NAK ou para `RespostaPesquisa`, o valor -2, ignorando o resto do processamento.
3. Processamento por Tipo: Se o pacote estiver OK, ele passa para a verificação do `pacote.tipo_msg`:
    - Tipo ‘D’: Abre o arquivo `dados_postos.csv` em modo de adição através do `ios::app`, salvando a nova entrada. Enviando logo em seguida um ACK.
    - Tipo ‘P’: Inicia o algoritmo de busca e envia o resultado para o cliente encapsulado na `struct RespostaPesquisa`.

- Algoritmo de Busca (Servidor):
    1. Os dados existentes no arquivo `dados_postos.csv` são lidos e carregados em um `vector<Posto>`.
    2. O vetor é iterado, e para cada posto são aplicados dois filtros:
        - O tipo de combustível é o procurado?
        - A distância geográfica calculada pela função `haversine` é menor ou igual ao raio de busca desejado?
    3. Para todos os postos que passam pelos filtros, o algoritmo mantém o registro do menor preço encontrado.
    4. No final, o menor preço (ou -1, caso nenhum posto for encontrado) é retornado.
- Cálculo de Distância (`haversine`): Uma função foi implementada para calcular a distância entre as duas coordenadas geográficas (latitude e longitude) na superfície de uma esfera. Essa função é essencial para a precisão da busca por raio.

3. DECISÕES DE IMPLEMENTAÇÃO OMISSAS NA ESPECIFICAÇÃO

Durante o desenvolvimento, certas ambiguidades foram notadas na especificação do trabalho e por isso, exigiu algumas tomadas de decisões como:

- Unidade do Raio de Busca: No documento não falava sobre a unidade do raio de busca. Por isso foi decidido implementar como quilômetros (km). Essa decisão está diretamente ligada com a função `haversine`, já que ela utiliza o raio da Terra em km para seus cálculos, tornando a entrada do usuário mais direta e intuitiva.
- Protocolo de Resposta para Pesquisas Corrompidas: A especificação do RDT 2.0 (ACK e NAK) estavam sendo aplicadas diretamente a mensagens ‘D’, porém o comportamento desejado para as mensagens do tipo ‘P’ que também podem chegar corrompidas não estava explícito. Então foi decidido estender o protocolo, em vez de usar um NAK genérico, o servidor responde aquela pesquisa corrompida com uma `struct RespostaPesquisa`, utilizando o valor especial `preco_encontrado = -2`, sinalizando o erro. Isso simplifica a lógica do cliente, pois sempre sabe qual tipo de `struct` esperar como resposta para uma mensagem ‘P’.
- Comportamento da Retransmissão: Para garantir uma demonstração clara e determinística da capacidade de recuperação do protocolo, foi decidido que após um NAK ou timeout, o cliente setaria o campo `erro` do pacote para `false` antes de retransmitir. Randomizar o erro novamente seria uma simulação mais realista do mundo real, mas a abordagem escolhida evita loops de retransmissão causados por falta de sorte, focando apenas em provar que o caminho de recuperação do protocolo funciona.
- Portabilidade do Código (Extra): Para garantir que os executáveis funcionem mesmo que seja baixados diretamente (em vez de compilar o código fonte), tornando possível funcionar em diferentes ambientes sem problemas de depender de bibliotecas, foi utilizada a flag de compilação -

`static-libstdc++` no `Makefile`, que embute a biblioteca padrão do C++ no executável final.

4. COMO FOI TRATADA A RETRANSMISSÃO DE MENSAGENS

A retransmissão foi implementada através de uma combinação de mecanismos no cliente e no servidor:

1. Simulação de Erro: O cliente, a cada novo pacote, tem 50% de chance de setar o campo `bool erro` para `true`.
2. Detecção de Erro: O servidor, por sua vez, verifica esse campo do pacote ao recebê-lo. Se for `true`, ele imediatamente envia uma resposta de falha, ou NAK.
3. NAK: O cliente quando recebe o NAK, entende que o pacote chegou corrompido e reentra em seu loop de retransmissão para enviar o mesmo pacote novamente.
4. Timeout: Para lidar com a perda de pacotes (seja o pacote de dados quanto a resposta do servidor), foi configurado um timeout de 2 segundos no socket do cliente usando o `setsockopt`. Se a função `recvfrom` retornar um erro, o cliente assume que um pacote foi perdido e então reinicia o processo de retransmissão.
5. ACK: Receber um ACK pelo servidor sinaliza sucesso da transmissão, o que quebra o loop de retransmissão e permite que o programa prossiga nornalmente.

5. TESTES E ANÁLISE

#	Cenário de Teste	Entradas de Exemplo (Cliente)	Saída Esperada (Cliente)
1	Cadastro de Dados	`D 101 2 4899 -22.221 -54.812`	`ACK recebido! Sucesso!`
2	Falha e Retransmissão	(Ocorre aleatoriamente)	`NAK recebido. Retransmitindo...` Seguido de `ACK recebido! Sucesso!`
3	Pesquisa Bem-Sucedida	`P 102 2 10 -22.220 -54.810`	`Resposta do servidor: Menor preço encontrado: [preço]`
4	Pesquisa Sem Resultado (Raio)	`P 103 2 1 -22.300 -54.900`	`Resposta do Servidor: Nenhum posto econtrado`
5	Pesquisa Sem Resultado (Tipo)	`P 104 0 20 -22.220 -54.810`	`Resposta do Servidor: Nenhum posto econtrado`
6	Perda de Pacote (Timeout)	(Simulado ao desligar servidor)	`Timeout! Retransmitindo`

Análise: Perceba que os testes demonstram que o sistema se comporta como esperado em todos os cenários possíveis. A implementação do RDT 2.0 foi eficaz em lidar com os erros simulados e os timeouts, e a lógica de busca e armazenamento de dados provou ser funcional e correta.

6. PRINT SCREENS DE FUNCIONAMENTO

Cenário 1: Cadastro de postos, incluindo retransmissão por NAK

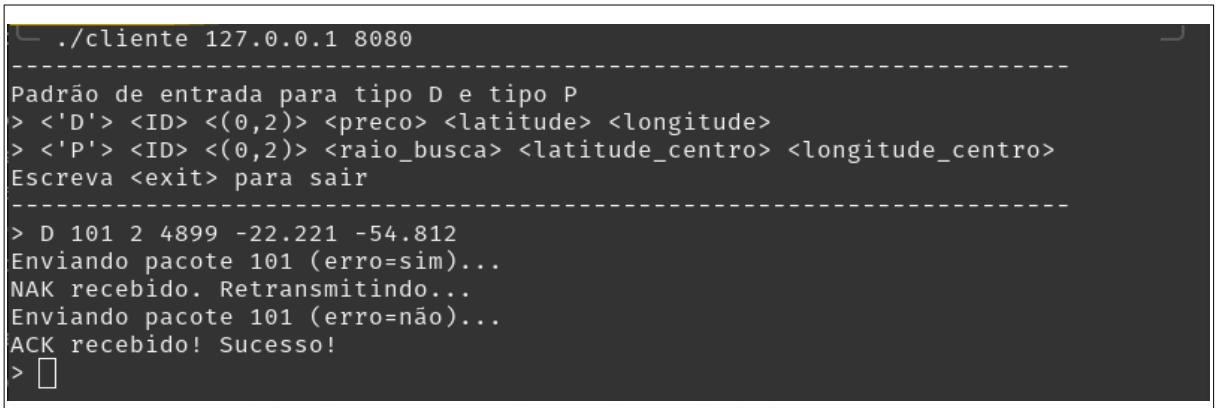

Figura 1: Terminal do Cliente mostrando o cadastro de dados e uma retransmissão.

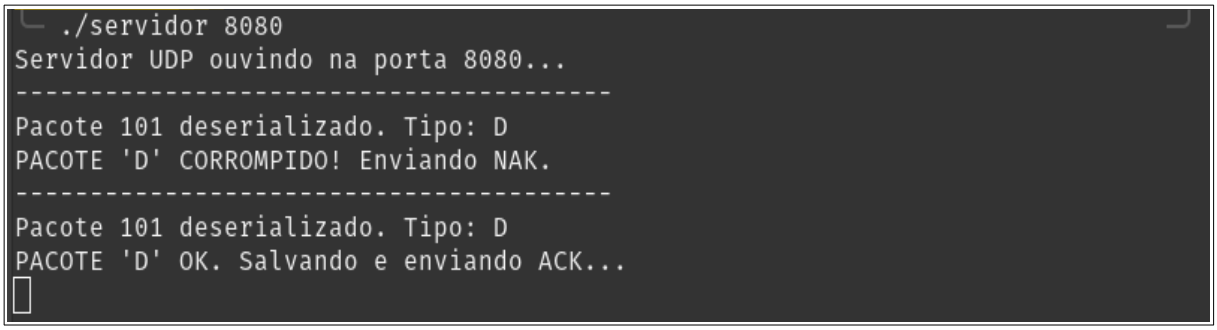

Figura 2: Terminal do Servidor mostrando o recebimento dos pacotes, o envio de NAK e o salvamento dos dados.

Cenário 2: Pesquisa bem-sucedida e pesquisa sem resultados

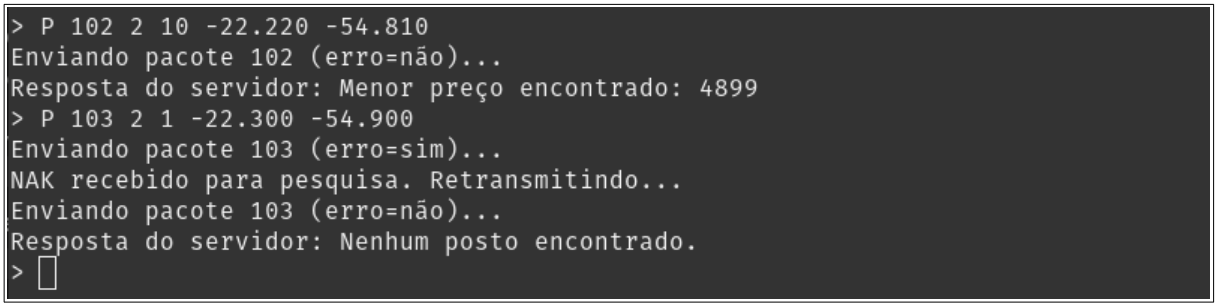

Figura 3: Terminal do Cliente mostrando uma pesquisa que retorna um preço e outra que não retornada nada.

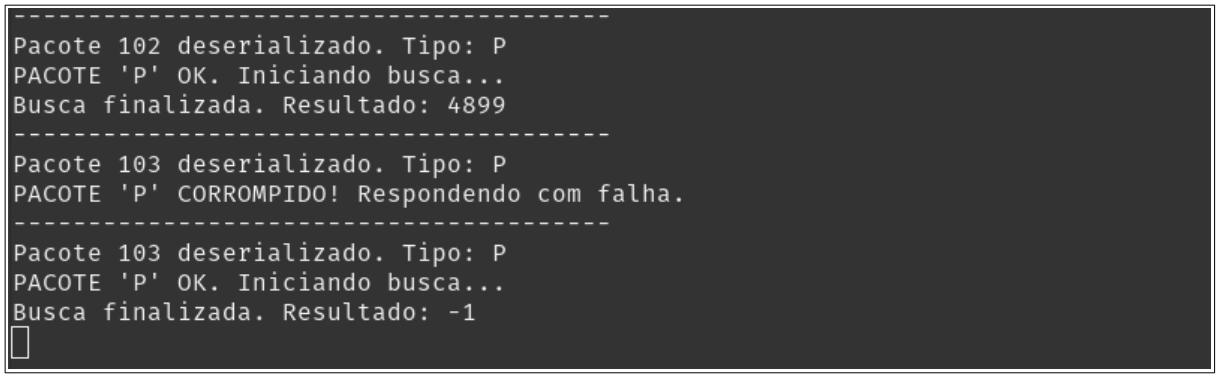

Figura 4: Terminal do Servidor mostrando o processamento das duas pesquisas.

7. CONCLUSÃO E REFERÊNCIAS BIBLIOGRÁFICAS

O desenvolvimento desse trabalho permitiu a aplicação prática de conceitos de Redes de Computadores, programação com Sockets UDP e a construção de um protocolo de confiabilidade sobre uma camada de transporte não confiável. O projeto foi concluído com sucesso atendendo todos os requisitos do documento, atendendo os requisitos funcionais resultando numa aplicação cliente-servidor funcional. Os desafios encontrados (especialmente na depuração do protocolo de comunicação) foram essenciais para o aprendizado.

- Documentação da ferramenta Make: https://www.gnu.org/software/make/manual/make.html
- Tutorial sobre Makefiles: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- Onde descobri a fórmula Haversine para cálculo de distância com latitudes e longitudes: https://codidata.com.br/glossario/o-que-e-haversine/#:~:text=Exemplo%20pr%C3%A1tico%20de%20c%C3%A1lculo%20com%20Haversine%20Para,%C3%A9%20essencial%20para%20a%20tomada%20de%20decis%C3%B5es.
- Código que usei para aplicar Haversine em C++: https://www.movable-type.co.uk/scripts/latlong.html

- Para ler e escrever arquivos em C++: https://www.guru99.com/pt/cpp-file-read-write-open.html
- Para parsear e manipulaar as strings: https://labex.io/pt/tutorials/cpp-how-to-use-stringstream-in-c-425236