Trabalho Prático 2 - Redes de Computadores

Maria Luiza Leão Silva - 2020100953
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil
marialuizaleao@dcc.ufmg.br

---

1. Introdução

Este documento aborda o desenvolvimento de uma plataforma de blog semelhante ao Medium, onde a interação entre múltiplos clientes remotos é viabilizada por meio de sockets e threads em C. O objetivo principal deste projeto é criar tanto um servidor robusto quanto um cliente que permita a publicação e leitura de conteúdos à distância. No servidor, reside a responsabilidade de gerenciar as postagens, inscrições em tópicos e atualizações, enquanto o cliente tem a função de enviar solicitações de leitura, escrita e interação com o blog.

Este projeto proporcionou uma oportunidade significativa para aplicar conhecimentos de programação em um contexto de rede, contribuindo para uma compreensão mais profunda e prática dos conceitos envolvidos na comunicação entre clientes e servidores por meio de sockets em C.

Na próxima seção, vamos mergulhar na parte de modelagem computacional do problema, onde examinaremos detalhadamente como os dados das postagens são estruturados no servidor e como as interações dos clientes são tratadas. Em seguida, na seção 3, exploraremos os desafios enfrentados durante o processo de desenvolvimento, seguidos por um guia passo a passo que explica como compilar e executar o programa do servidor e do cliente para a plataforma do blog.

2. Modelagem computacional do problema

a. Servidor

Tendo em vista que o servidor deve ser o responsável por armazenar o blog. Temos que as seguintes estruturas foram criadas:

```c
struct Client
{
    int id;
    int sock;
    pthread_t thread;
};
```

```c
struct Post
{
    int id;
    char content[BUFFER_SIZE];
    int madeBy;
};
```

```c
struct Topic
{
    int id;
    char name[50];
    bool subscribed[MAX_USERS];
    int subscribedCount;
    struct Post posts[MAX_POSTS];
    int postsCount;
};
```

```c
struct Blog
{
    struct Topic topics[MAX_TOPICS];
    struct Client clients[MAX_USERS];
    bool users[MAX_USERS];
    int topicsCount;
    int usersCount;
};
```

A struct Client representa cada cliente conectado, contendo um identificador único, o socket associado para comunicação e a thread que lida com as interações desse cliente. Já a struct Post armazena informações sobre cada post, incluindo um identificador único, o conteúdo do post com um limite definido (BUFFER_SIZE) e o identificador do cliente que fez o post. Ademais, a struct Topic é essencial para organizar os posts por tópicos, mantendo um nome para o tópico, um vetor booleano que rastreia quais usuários estão inscritos nele, contagem de inscritos, uma lista de posts associados a esse tópico e a contagem de posts. Por fim, a struct Blog serve como a estrutura principal que guarda todas as informações do blog. Ela mantém arrays de tópicos disponíveis, clientes conectados e uma matriz booleana para rastrear a presença de usuários. Além disso, registra o número total de tópicos e usuários no blog.

Foi instanciada uma variável global de tipo struct Blog para representar o blog. As funções de manipulação, adicionar novos usuários, novos posts ou inscritos, alteraram diretamente essa variável global. Assim, temos uma única cópia do blog durante toda a execução.

Para lidar com os pedidos de cada cliente, foram utilizadas threads. O seu uso será explicado mais profundamente posteriormente.

b. Cliente

Não foi criada nenhuma estrutura específica no código do cliente. Temos que sua principal função é ler os inputs e a partir deles gerar pedidos que serão enviados para o servidor. Com esse objetivo, foram feitas as funções parseCommand e parseContent.

A função parseCommand recebe uma string de entrada do usuário e identifica o comando digitado. Ela verifica se o comando é "exit", "list topics", "subscribe", "unsubscribe" ou "publish". Dependendo do comando reconhecido, ela retorna um código correspondente para indicar a ação a ser executada no programa.

Já a função parseContent trabalha em conjunto com a parse Command. Ela recebe o código do comando identificado anteriormente e a entrada do usuário. Com base no comando recebido, ela interpreta o conteúdo adicional fornecido pelo usuário. Por exemplo, para um novo post (NEW_POST), ela extrai e retorna o conteúdo do post. Para comandos de inscrição (SUBSCRIBE) ou desinscrição (UNSUBSCRIBE), ela captura o nome do tópico ao qual o usuário deseja se inscrever ou desinscrever.

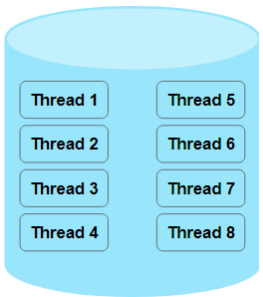
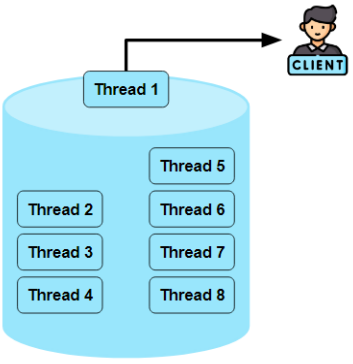
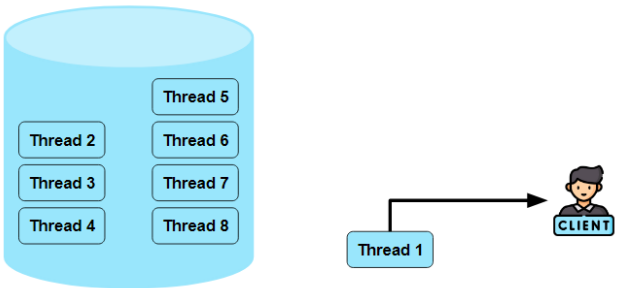
O principal fluxo de execução acontece na main, que, após iniciar os parâmetros e o socket, estabelece a conexão com o servidor e entra em um loop while(true) onde parseCommand é chamada. Cada caso possível de input é tratado por um switch. Neste, a função parse content é chamada no caso de ser um pedido de NEW_POST, SUBSCRIBE ou UNSUBSCRIBE. No final do while, o pedido é enviado para o servidor.

Para receber respostas e notificações do servidor, foi criada uma thread que roda a função waitForResponse. O uso das threads será explicado melhor no próximo tópico.

c. Interação Cliente/Servidor

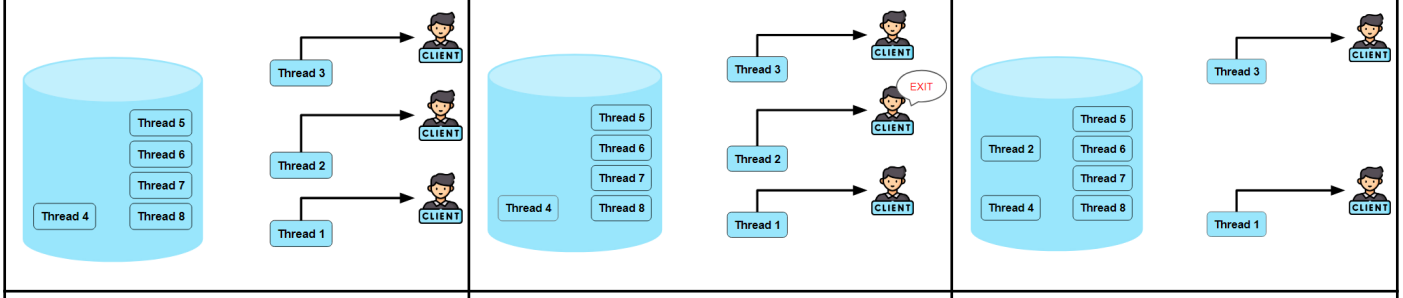
Na arquitetura cliente-servidor com uso de threads, a interação segue um fluxo específico. No servidor, a cada conexão de um cliente, uma nova thread é alocada para atendê-lo individualmente. Isso cria um "pool" de threads disponíveis para múltiplos clientes. Esse “pool” de thread é representado dentro da estrutura do blog pela variável struct Client clients[MAX_USERS], isso pois cada objeto Client tem uma thread, declarada como pthread_t thread.

Cada thread no servidor é responsável por receber e processar as requisições vindas dos clientes conectados. Essas requisições podem variar desde publicações de novos posts até solicitações de inscrição em tópicos ou atualizações sobre determinados assuntos. O servidor é encarregado de armazenar todas as informações do blog, como posts, clientes conectados e tópicos disponíveis, garantindo a integridade e o acesso adequado a esses dados.

Pool de threads		
		
Nenhum cliente conectado.	Quando o primeiro cliente conecta, uma thread será alocada para ele.	Essa thread roda em paralelo com as outras threads e a main.

Quando um cliente se desconecta do servidor, seja pelo EXIT ou pelo fim de sua execução, a thread ao qual estava alocado, retorna para o pool. Assim, está disponível para um futuro cliente que se conectar ao servidor.

Desconexão de um cliente



Clientes rodando em paralelo	Cliente manda o pedido EXIT	Cliente é desconectado e sua thread volta para o pool

No lado do cliente, uma thread é dedicada à recepção de notificações do servidor. Esta thread permanece em constante espera, aguardando por mensagens ou atualizações do servidor. Essa abordagem permite que o cliente seja imediatamente notificado sempre que houver um novo post em um tópico em que esteja inscrito. Através da sua thread principal, o cliente pode enviar requisições para o servidor, como a escrita de novos posts, inscrição em tópicos específicos ou a busca por informações sobre determinados temas.

Essa implementação com threads facilita o gerenciamento de múltiplas operações simultâneas tanto no servidor quanto nos clientes, garantindo que a interação entre eles seja eficiente e sem bloqueios desnecessários. Além disso, a utilização de uma thread dedicada para notificações no cliente permite uma comunicação assíncrona e responsiva, melhorando a experiência do usuário ao receber atualizações em tempo real.

3. Desafios

Durante minha jornada de implementação do blog, me deparei com vários desafios e obstáculos que precisei superar. Essa seção irá explorar os desafios principais encontrados e como eles foram resolvidos.

a. Socket

Um desafio foi lidar com a configuração e o gerenciamento dos sockets para estabelecer a comunicação entre o cliente e o servidor. Foi necessário aprender a configurar detalhes como endereços IP, portas, tipo de conexão e versão do endereçamento. Para superar essa complexidade, dediquei tempo ao estudo da biblioteca de sockets em C, consultando os recursos disponíveis, a fim de garantir a configuração correta. Uma questão importante foi a complexidade de tempo e espaço, especialmente ao considerar a possibilidade de expansão para tabuleiros maiores. Garantir um desempenho eficiente e um consumo de recursos controlado exigiu a implementação de algoritmos otimizados e estruturas de dados eficazes, independentemente do tamanho do tabuleiro utilizado.

b. Thread

A utilização de threads tanto no servidor quanto no cliente apresenta desafios únicos. Primeiramente, tive dificuldade para entender como seria o fluxo de execução de ambos os códigos, principalmente com a execução de threads em paralelo. Abstrair quais partes do código deveriam estar rodando em paralelo foi fundamental para que conseguisse evoluir na resolução do trabalho. Utilizando o tp1 como base, tive que pensar como extrair funções somente para as partes paralelas.

Além disso, gerenciar o ciclo de vida das threads do servidor é desafiador. Criar e alocar threads para lidar com clientes conectados, garantindo que sejam liberadas de forma adequada quando os clientes se desconectarem, é essencial para evitar vazamento de recursos ou

sobrecarga do sistema. No cliente, a principal preocupação está na receptividade às notificações. A thread dedicada para receber atualizações do servidor deve aguardar essas notificações de maneira eficiente, sem consumir recursos em excesso.

c. Client ID

Outro desafio foi garantir que o novo cliente recebesse o menor ID disponível. Isso porque quando um cliente desconecta seu ID passa a ficar disponível para futuros clientes. A solução encontrada foi o uso de um vetor booleano de tamanho 10. Se uma posição n está verdadeira, isso significa que o ID n já está sendo usado por algum cliente. Quando um cliente desconecta a posição do seu ID volta a ser falsa e quando um cliente conecta pela primeira vez, seu ID será a primeira posição falsa encontrada nesse vetor.

Esse problema do ID também influenciou como guardar a lista de inscritos de um tópico, isso porque ao se desconectar, o cliente deve ser retirado da lista de inscritos de todos os tópicos caso ele esteja lá. Para evitar que tenhamos que verificar todo o vetor de inscritos de todos os tópicos do blog, o vetor de inscritos é um tipo de hash booleano que utiliza o ID do cliente para saber exatamente qual posição checar.

d. Envio de notificações para inscritos

Outro desafio foi o envio de notificações para os usuários inscritos. Para isso foi preciso guardar o número do socket de cada um dos clientes, algo que não tinha colocado inicialmente na estrutura. Mas uma vez que o vetor de inscritos funciona como um hash, como explicado anteriormente, foi facilitada a busca pelo cliente e pelo seu socket.

Além disso, durante a fase final da implementação, enfrentei o desafio de garantir que meu código se mantivesse claro e organizado. Isso envolveu a separação do código em vários arquivos para modularizar as funcionalidades, a redução do tamanho da função main() e a revisão e refatoração constante do programa. Essas medidas foram essenciais para melhorar a legibilidade, facilitar a manutenção e promover boas práticas de programação, tornando o código mais compreensível. Finalmente, após a conclusão do código, revisei a documentação da biblioteca de sockets para garantir que diferentes implementações do blog pudessem interoperar adequadamente e que as conexões funcionassem conforme o esperado.

4. Instruções para compilação e execução

1 – Extraia o arquivo .zip na pasta desejada.

2 – Execute o comando ‘make’ no terminal.

3 – Execute os programas client e server da pasta bin.

- Funcionamento do cliente: ./bin/client <endereço ip> <porta>
- Funcionamento do servidor: ./bin/server <v4|v6> <porta>