Henrique Assumpção
Matrícula: 2020006620
henriquesoares@dcc.ufmg.br

TP2: Relatório
Redes de Computadores
Universidade Federal de Minas Gerais

2023-10-26

1 Introdução

O segundo trabalho prático da disciplina de redes consiste na implementação de um blog que permitirá a comunicação de um servidor com diversos clientes, utilizando a biblioteca de sockets e pthreads para sistemas Unix.
A conexão entre cliente e servidor deve ser feita por meio do protocolo TCP, utilizando endereços do protocolo IP, tanto a versão 4 quanto a versão 6. O servidor deve ser capaz de receber múltiplas conexões, armazenar os tópicos criados pelos clientes, e garantir que mensagens enviadas por um cliente cheguem em todos os demais clientes que estão inscritos em um dado tópico alvo. O cliente deve ser capaz de receber uma entrada que representa um comando, e então enviar tal comando ao servidor, e também deve ser capaz de receber mensagens do servidor informando novas mensagens que foram postadas em tópicos nos quais o cliente está inscrito.

2 Modelagem

O programa foi desenvolvido na linguagem C, no sistema operacional Windows 11, utilizando o sistema WSL de Linux. O programa foi implementado e compilado em uma máquina com 16GB de RAM em um processador Intel Core I5-9400F.
Nesta seção, iremos fornecer uma breve descrição da solução adotada para implementar o trabalho. Os detalhes de implementação podem ser encontrados nos comentários feitos no código fonte, portanto iremos prover apenas uma discussão em alto nível.

2.1 Cliente

O funcionamento do cliente se da de forma relativamente simples, sendo que o fator mais complexo relativo à sua implementação é o uso de threads para realizar procedimentos em paralelo. O funcionamento do cliente é dividido em duas componentes: a thread principal, que realiza o loop de envio (send) de mensagens ao servidor, e uma thread auxiliar que realiza o loop de recebimento (receive) de mensagens do servidor. Note que tal solução de threads é necessária: o enunciado contempla cenários onde um cliente fica apenas recebendo mensagens por tempo indeterminado – e.g. um cliente se inscreve em um tópico e fica apenas recebendo posts no tópico –, ou seja, não é possível que se faça um loop simples com um send e um receive sequenciais (pois essas operações são bloqueantes), e portanto tais operações devem ser independentes.
A implementação do cliente se deu de acordo com os seguintes passos:

1. Primeiro o cliente recebe os parâmetros da linha de comando, e então realiza o processamento necessário do endereço IP fornecido e do porto que será utilizado para a conexão. A função PARSE_CLT_ADDR é a que realiza todo o processamento dos parâmetros, isto é, ela identifica se o endereço é um IPv4 ou um IPv6, e então prossegue para inicializar as estruturas ADDR necessárias.

2. Após processar os parâmetros, o cliente inicializa seu socket e se conecta ao endereço do servidor, dando início a conexão.

3. O cliente inicializa um objeto de tipo BLOGOPERATION com os parâmetros iniciais para a primeira conexão, e então realiza um send e um receive inicial para identificar seu ID, que então é armazenado em uma variável. Após isso, o cliente dispara a thread de receive de mensagens do servidor.

4. A thread de receive funciona da seguinte maneira: um loop WHILE(1) é executado, e a cada iteração o cliente espera por uma mensagem do servidor. Ao receber a mensagem, o cliente checa pelo tipo da operação: se a operação é do tipo listagem de tópicos, novo post ou subscribe, ele imprime o seu conteúdo (note que o caso subscribe é somente para o caso de "error: already subscribed"), caso contrário checa por erros no recebimento e então passa para a próxima iteração. No caso de erros de recebimento de mensagem, o loop é encerrado e uma variável global é utilizada para sinalizar que o cliente deve encerrar a conexão com o servidor.

5. A thread principal de send funciona da seguinte forma: um loop WHILE(1) é executado, e a cada iteração o cliente recebe um input do usuário na entrada padrão, faz o parsing do input, checa por potenciais erros e então envia a mensagem ao servidor. Em caso de recebimento de uma operação de exit ou em caso de erro de recebimento, o loop se encerra e o cliente fecha a conexão com o servidor.

2.2 Servidor

O funcionamento do servidor é significativamente mais complexo que o do cliente, principalmente devido ao uso de threads para atender diversos clientes simultaneamente. O servidor é dividido em duas componentes: a thread principal que recebe conexões de clientes e dispara novas threads, e as threads de atendimento ao cliente, que são responsáveis por receber e enviar mensagens aos clientes. Note que, pelo enunciado, as threads de atendimento podem seguir um comportamento sequencial: primeiro elas recebem uma mensagem do cliente e então elas enviam uma mensagem de volta ao cliente (e a potencialmente mais clientes). O ponto vital é que: o servidor só envia uma mensagem após receber uma mensagem de algum cliente, e portanto podemos adotar o supracitado comportamento.
A implementação do servidor se deu de acordo com os seguintes passos:

1. Primeiro o servidor recebe os parâmetros da linha de comando, e então realiza o processamento do indicador da versão do protocolo IP recebido e do porto que será utilizado. A função PARSE_SERV_ADDR é utilizada para determinar a versão do protocolo IP que será utilizada, e também para inicializar as estruturas ADDR necessárias. O servidor então inicializa seu socket, realiza a operação de bind do socket com o endereço e então realiza a operação de listen.

2. Após isso, o servidor entra em um loop WHILE(1), no qual ele recebe uma nova conexão de um cliente, atribui o menor ID disponível possível ao mesmo, e então dispara uma nova thread para atender o cliente. O Loop da thread principal do servidor não termina a menos que o programa seja forçadamente encerrado, dado que nenhum procedimento de término foi especificado no enunciado.

3. A thread de atendimento ao cliente funciona da seguinte forma: primeiro espera-se uma mensagem do cliente, a mensagem é então processada, e então o servidor envia uma resposta de acordo com o tipo de operação recebido, e.g., no caso de um novo post, o servidor checa se o tópico existe (se não o tópico é criado), e então envia o post para todos os clientes inscritos no tópico. Caso o servidor receba um disconnect ou um erro no recebimento da mensagem, ele encerra o loop e fecha o socket do cliente.

2.3 Organização do Código

O código foi organizado da seguinte maneira:

- A pasta /SRC/ contém os arquivos CLIENT.C,SERVER.C,COMMON.C,CLT_UTILS.C,SERV_UTILS.C que implementam o cliente, o servidor, as funções comuns, as funções exclusivas do cliente, e as funções exclusivas do servidor, respectivamente.
- A pasta /INCLUDE/ contém os arquivos ,COMMON.H,CLT_UTILS.H,SERV_UTILS.H.
- A pasta /OBJ/ contém os arquivos .o obtidos após a execução do comando MAKE.
- A pasta /BIN/ contém os arquivos binários do cliente e do servidor obtidos após a execução do comando MAKE.
- No diretório raiz, encontra-se o arquivo MAKEFILE que compila os demais arquivos e gera os binários.

3 Desafios e Decisões de projeto

Nesta seção, iremos discutir algumas decisões de projeto e desafios que marcaram o processo de implementação do trabalho, e também explicitar as soluções adotadas para superar tais desafios.

3.1 Armazenamento de tópicos e clientes no servidor

Segundo o enunciado, o servidor é o responsável por armazenar os clientes conectados e os tópicos que são postados. Inicialmente encontramos dificuldades em decidir qual estrutura de dados seria mais adequada para armazenar informações sobre os clientes e sobre os tópicos, levando em consideração que o enunciado não configura um número máximo de tópicos, i.e., a estrutura deve ser dinâmica e ser capaz de armazenar uma quantidade arbitrária de tópicos de maneira eficiente.
Para resolver este problema, utilizamos a seguinte estratégia: criamos structs CLIENT_DATA e BLOG_TOPIC, sendo que a primeira dela é responsável por armazenar o ID e o socket do cliente, e a segunda é responsável por armazenar o nome do tópico, um vetor de ponteiros para CLIENT_DATA representando os clientes inscritos, e um ponteiro para outro BLOG_TOPIC. A ideia é criar uma lista ligada simples de tópicos criados, onde cada objeto de tipo BLOG_TOPIC aponta para o próximo elemento da lista, e uma variável global LIST_HEAD será compartilhada entre todas as threads para que qualquer uma possa manipular a lista de tópicos. Essa solução garante que a quantidade de tópicos que possam ser criados seja limitada somente pela memória, i.e., não depende de alguma variável de tamanho. Por outro lado, como o enunciado afirma que somente 10 clientes serão testados, optamos por implementar os vetores de objetos CLIENT_DATA de maneira estática, parametrizados por uma constante indicando o número máximo de clientes.
Por fim, optamos também por utilizar um vetor de inteiros parametrizado pelo número máximo de clientes para alocar IDS à novos clientes. Esse vetor é uma variável global, e a i-ésima posição do vetor assume o valor 1 caso esse ID esteja disponível, e 0 caso contrário, portanto basta que um novo cliente percorra o vetor até encontrar o primeiro valor 1, e dessa forma seu ID será o menor possível disponível.

3.2 Exclusão mútua no servidor

As soluções discutidas na última seção são baseadas em memória compartilhada para diversas threads dentro do servidor, portanto um problema natural de programação paralela emerge: **condições**

de corrida. Para garantir exclusão mútua para as variáveis globais criadas, criamos duas variáveis mutex: uma para controlar o acesso ao vetor de IDS, e outra para controlar o acesso à lista de tópicos, e portanto as threads devem travar a mutex antes de acessar a variável e destravar após o acesso. Note que essa solução é necessária: se não utilizarmos variáveis mutex, pode ocorrer de dois clientes se conectarem simultaneamente e obterem o mesmo ID, e também pode ocorrer de dois clientes realizarem um subscribe simultâneo no mesmo tópico, fazendo com que apenas um deles seja registrado como inscrito.
Além dos problemas advindos das soluções adotadas na seção anterior, a própria natureza do enunciado requer utilização de exclusão mútua para funcionamento correto. A operação de send em um socket utilizando o TCP não garante exclusão mútua, e portanto se dois sends forem feitos simultaneamente podem ocorrer erros relativos à condição de corrida. Esse cenário não só é possível como provavelmente será relativamente comum em quaisquer testes realizados com uma quantidade significativa de clientes. Considere o cenário onde os clientes C_1, C_2 estão inscritos no tópico S, e o servidor cria threads T_1, T_2 para cada cliente, respectivamente. Note que caso o cliente C_1 realize uma operação de *list topics* ao mesmo tempo que o cliente C_2 realize uma operação de *publish in* no tópico S, podemos ter uma condição de corrida: C_1 envia a mensagem a T_1, e então T_1 envia a lista de tópicos pelo socket de C_1, mas ao mesmo tempo T_2 recebe o comando publish de C_2 e envia o post a C_1 (pois ele está inscrito em S), fazendo com que ocorram dois sends simultâneos para o socket de C_1, podendo gerar erros na recepção da mensagem. A solução de tal problema é relativamente simples: utilizamos um vetor de variáveis mutex, no qual a i-ésima variável controla o acesso a operações de send ao socket do cliente de ID i, i.e., cada thread agora deve travar a mutex do respectivo cliente antes de enviar uma mensagem a ele, e destravar após o envio. Isso garante que o envio de mensagens seja feito em exclusão mútua, e portanto resolve a condição de corrida.

3.3 Tratamento de erros na troca de mensagens

Além dos erros descritos no enunciado, um erro que pode ocorrer na troca de mensagens é um possível erro na recepção dos dados. Tal erro pode ocorrer por dois motivos: a rede em si está comprometida, ou quem enviou a mensagem está comprometido. De maneira semelhante ao que foi feito no último trabalho, optamos por encerrar a conexão em ambos os casos, isto é, se o servidor identifica que ocorreu um erro de transmissão, ele encerra a conexão com o cliente que enviou os dados. Similarmente, se o cliente recebe uma transmissão defeituosa do servidor, ele também desconecta e encerra seu programa. O enunciado não especifica o que deve ser feito em tais casos, portanto optamos por seguir uma estratégia conservadora e simples.

4 Conclusão

O segundo trabalho prático da disciplina foi uma experiência desafiadora, mas no entanto extremamente gratificante. A experiência prática adquirida ao longo do trabalho agrega ao conhecimento fornecido pelo professor em sala de aula, fazendo com que o aprendizado seja mais completo. De modo geral, acreditamos que fornecemos uma implementação completa dentro do contexto do enunciado, que não só executa as ações esperadas mas também lida com alguns cenários de erro de forma criativa e consistente com o trabalho.