Redes de Computadores
Trabalho Prático 2

Luís Felipe Ramos Ferreira

lframos_ferreira@outlook.com

1 Introdução

O Trabalho Prático 2 da disciplina de Redes de Computadores teve como proposta o desenvolvimento de um *Blog* que permite a interação entre vários clientes em um servidor usando *sockets* e *threads* com a linguagem C.

O repositório onde está armazenado o código utilizado durante o desenvolvimento desse projeto pode ser encontrado neste endereço.

2 Implementação

Conforme especificado no enunciado, o projeto foi todo desenvolvido na linguagem de programação C em um ambiente *Linux*, e o manuseio de *sockets* e *threads* por meio da interface POSIX disponibilizada para a linguagem. Para manter uma maior organização do código, além dos arquivos *server.c* e *client.c*, os quais possuem respectivamente as implementações do servidor e do cliente, foram criados uos arquivos auxiliares *common.c* e seu arquivo de cabeçalho *common.h*, os quais possuem as especificações e implementações de funções auxiliares que podem ser utilizadas por ambas as partes do projeto, assim como o arquivo *topic.c* e seu arquivo de cabeçalho *topic.h*, os quais possuem as especificações e implementações relativas à estrutura de dados que armazena os tópicos criados para o *Blog*.

2.1 Arquitetura do sistema

Conforme especificado, o servidor implementado deve ser capaz de lidar com múltiplas conexões de clientes ao mesmo tempo. Para realizar essa tarefa, foram utilizadas *threads* da interface POSIX. Cada cliente possui um *thread* associada a ele no servidor, a qual está desanexada da *thread* principal do programa. A escolha de implementá-las dessa maneira veio do fato de que não há a necessidade de se realizar um *join* em cada *thread* após que ela finalize sua execução, e assim que a conexão for finalizada os recursos associados à aquela *thread* em específico podem ser desalocados.

Para armazenar os tópicos criados pelos clientes, foi criada uma lista encadeada, uma vez que se trata de uma estrutura simples de se criar e manipular, assim como cresce dinamicamente de forma controlada, o que é desejável já que o número de tópicos que os clientes irão criar não é determinado a princípio.

Para manter controle do menor identificador disponível para ser atribuído à um novo cliente, um simples vetor binário que contêm um mapeamento de quais identificadores estão sendo utilizados é armazenado no servidor e este manipulado por uma função específica denominada *test_and_set_lowest_id()*.

Pelo lado do cliente, duas *threads* são utilizadas. A *thread* principal, inicialmente, estabelece a primeira conexão com o servidor e obtêm o identificador que foi atribuído pelo servidor à ele. Em seguida, ela irá continuamente ler da entrada do usuário para saber qual a próxima operação que ele deseja que seja executado.

A outra *thread* é responsável por continuamente escutar o servidor e imprimir em tela o que for necessário. Por exemplo, nos tipos de operação *list topics* e *publish in ¡topic¿*.

Para que cada *thread* do servidor saiba para quais *sockets* ele deve enviar a mensagem de adição de um novo tópico, após uma ação desse tipo, o descritor de arquivo do *socket* de cada cliente é armazenado juntamente na lista de clientes conectados, facilitadno assim a troca de mensagens na arquitetura do sistema.

2.2 Condições de corrida

Por se tratar de uma arquitetura com múltiplas *threads*, problemas como condições de corrida podem acontecer. Uma vez que estruturas como a lista encadeada de tópicos e o vetor de identificadores disponíveis podem ser acessados e manipulados por diferentes *threads* ao mesmo tempo, situações estranhas e não determinísticas podem ocorrer.

Para resolver esse problema, foi adotado o uso da estrutura de um mutex, que garante que duas threads não irão acessar/modificar a mesma estrutura de dados global ao mesmo tempo.

3 Desafios, dificuldades e imprevistos

A primeira dificuldade imposta pelo Trabalho Prático II foi a familiarização com a interface POSIX de programação em *threads*. Implementar uma arquitetura com múltiplas *threads* nunca é trivial, e garantir que condições de corridas não irão ocorrer e que a memória alocada para as *threads* será desalocada assim que elas não forem mais utilizadas foi um grande empecilho. Como comentado anteriormente, evitar problemas como condições de corrida foi feita por meio da implementação de vários *mutex*.

Um outro problema encontrado no desenvolvimento foi o da implementação da estrutura de dados para armazenamento dos tópicos do servidor e dos clientes associados a cada um deles. A princípio, a solução imediata para garantir organização foi a de criar uma *struct* específica para representar um tópico, a qual armazenaria seu nome e a lista de clientes inscritos nele e, globalmente no servidor, manter um vetor com os tópicos criados. A questão é que o número de tópicos que podem ser criados não é determinado a princípio e, dessa forma, a estrutura que os armazena deveria ser dinâmica. Por isso, optou-se por criar uma lista encadeada de tópicos, como detalhado melhor na seção anterior. A implementação da lista foi trabalhosa, uma vez que exige uma manipulação complexa de ponteiros e referências para posições de memória, e diversos erros foram encontrados durante o desenvolvimento, como *segmentation fault* por exemplo. A resolução desses problemas e a implementação de uma lista encadeada eficiente e correta foi possível com o uso de ferramentas de *debugging* assim como a revisitação de conteúdos vistos durante a disciplina de Estruturas de Dados.

O principal desafio do trabalho, no entanto, se referiu à como fazer a operação de *publish in ¡topic¿* funcionar. Quando um cliente executa essa ação, uma mensagem de aviso deve ser redirecionada para todos os outros clientes inscritos naquele tópico em particular, e compreender como fazer esse tipo de comunicação tomou muito tempo. A solução adotada segue a seguinte estrutura:

1. O servidor possui N *threads*, onde $N-1$ são utilizadas para manter a conexão com cada um dos $N-1$ clientes atualmente conectados e a outra responsável por escutar e aceitar novas conexões.

2. O cliente possui duas *threads*. Uma delas é responsável por ler da entrada do usuário e processar as operações desejadas, enquanto a outra constantemente escuta possíveis mensagens enviadas pelo servidor.

3. O servidor mantêm uma lista com os descritores de arquivo dos *sockets* atrelados a cada cliente conectado. Quando a operação de *publish in ¡topic¿* é processada, a lista de clientes inscritos naquele tópico é analisada e, para cada um deles, a *thread* responsável pelo cliente que enviou a operação inicial envia para cada um desses outros clientes a mensagem de aviso de nova publicação.

Essa solução é funcional e cumpriu os critérios estabelecidos, embora acredite que existam soluções mais complexas onde as próprias *threads* responsáveis por cada cliente consigam enviar as mensagens de aviso para seus respectivos clientes por meio de algum tipo de comunicação entre *threads*.

4 Conclusão

Em suma o projeto permitiu grandes aprendizados tanto na parte teórica como na parte prática no que se refere à programação em redes. Em particular, a arquitetura necessária para esse projeto em particular possibilitou que importantes conceitos ligados a programação com *threads* fossem revisitados.

Assim como o Trabalho Prático I, esse trabalho tornou compreender melhor como funciona o protocolo de comunicação TCP, como deve ser feita e mantida a comunicação entre um servidor e um cliente, etc.

5 Referências

- Livros:
  - Tanenbaum, A. S. & Wetherall, D. (2011), Computer Networks, Prentice Hall, Boston.
  - TCP/IP Sockets in C: Practical Guide for Programmers, Second Edition
- Web:
  - https://www.ibm.com/docs/en/zos/2.3.0?topic=sockets-using-sendto-recvfrom-calls
  - https://www.educative.io/answers/how-to-implement-tcp-sockets-in-c
- Youtube:
  - Jacob Sorber
  - Think and Learn sockets playlist
  - Playlist do professor Ítalo Cunha