Trabalho Prático 1
Servidor de Mensagens

Arthur Souto Lima

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

arthursl@ufmg.br

1. Introdução

O projeto consiste em implementar um sistema de comunicação entre um servidor e clientes via soquetes e protocolo TCP, usando a biblioteca POSIX. O sistema permite que os clientes enviem mensagens ao servidor e demonstrem interesse em algumas tags. Quem tiver interesse numa tag deve receber as mensagens enviadas ao servidor com tal tag. Além da biblioteca POSIX, as bibliotecas padrão de entrada e saída, bem como de TADs da STL foram utilizads. Para tratar o multithreading, a biblioteca *pthread* foi usada no projeto.

O programa foi implementado em C++, compilado pelo compilador G++ da GNU Compiler Collection. Juntamente com os códigos-fonte, está incluso também um arquivo *makefile* que é utilizado para compilar o programa através do comando *make*.

2. Implementação

O projeto deveria utilizar o protocolo TCP fazendo a comunicação por soquetes da biblioteca POSIX. Assim, o código que trata desse dessas funcionalidades de comunicação e conexão entre servidor e clientes foi advindo da implementação da aula de Programação em Redes por [Cunha 2020]. Além disso, outras funcionalidades de um sistema de comunicação via soquetes foi inspirado nas implementações de [Sinha 2019] e de [Tomar 2019].

A seguir são discutidas algumas funcionalidades requeridas, bem como as respectivas soluções implementadas.

2.1. Tags

As tags eram um conceito central do TP, já que, como explicitado na especificação, um cliente pode se inscrever numa tag e receber todo tipo de mensagem que chegar ao servidor com tal tag. Essas duas funcionalidades exigiram algumas funções auxiliares e uma estrutura de dados para serem implementadas.

Primeiramente, era necessário descobrir as tags de uma mensagem. A função auxiliar que faz isso procura o caractere ”#” e avalia a porção de texto subsequente. Se encontrar uma tag válida, adiciona essa porção da mensagem, ou seja, a tag, a um conjunto (da STL, o *set*). O uso dessa estrutura de dados permite que mesmo que uma tag seja utilizada várias vezes na mesma mensagem, ela será contada apenas como uma vez para motivos de envio aos seguidores.

O segundo passo é termos alguma forma de armazenar as inscrições dos clientes. A solução adotada foi criar um mapa (da STL, o *map*) que associava um cliente a um

vector (da STL) de tags às quais ele tinha se inscrito. Há funções auxiliares que inserem e removem tags de um determinado cliente, chamadas *subscribeToTag* e *unsubscribeFromTag*, implementadas no módulo *common*.

Uma observação é que o cliente é identificado no mapa como uma string que concatena seu IP e sua porta. Essa string é a saída da função *addrstr*, implementada por [Cunha 2020].

2.2. Subscribe

Como já comentado, um cliente pode demonstrar interesse numa tag. Essa informação é guardada num mapa e dados são colocados ou removidos através das funções auxiliares citadas acima.

Após a inscrição, o servidor deve enviar uma confirmação para o cliente. Além disso, caso ele requeira se inscrever numa tag à qual ele já está inscrito, a mensagem deve ser levemente diferente, apenas avisando essa situação. Esse caso é detectado pelo retorno da função *subscribeToTag*. Caso ela retorne *true*, significa que o mapa foi alterado, ou seja, o cliente não seguia a tag ainda, agora ele a segue. Caso retorne *false*, o cliente já seguia a tag e nada foi alterado no mapa. A situação é análoga para a desinscrição, com o retorno significando novamente a alteração do mapa ou não.

Através do retorno, podemos identificar qual mensagem enviar para o cliente após sua requisição.

2.3. Notify

Como colocado na especificação, seguindo a arquitetura Publish/Subscribe, tem-se que enviar a mensagem a todos os inscritos de alguma das tags usadas (notify). Cada usuário só pode receber uma cópia da mensagem mesmo se estiver inscrito em mais de uma tag usada. Esses requisitos novamente propiciam o uso de um *set*, agora de clientes.

No servidor, quando a mensagem é processada, após ser recebida, descobrimos o conjunto de tags utilizadas, como foi descrito anteriormente, e, em seguida, iteramos pelo mapa de tags e clientes, verificando se cada cliente deve receber a mensagem. Se sim, colocamos ele (no caso seu identificador, ou seja, a string com o IP e a porta) no conjunto. O uso do *set* permite que a mesma mensagem não seja enviada mais de uma vez para cada cliente, mesmo se ele seja inscrito em mais de uma tag usada.

Finalmente, temos que enviar a mensagem a cada cliente no conjunto obtido. Para fazer o envio, precisamos do soquete o qual cada cliente está conectado ao servidor, o que nos leva à necessidade de outra estrutura de dados. Um outro mapa (*map*) que associa um cliente (string IP e porta) ao soquete que ele utiliza (um número inteiro). O mesmo é preenchido sempre que um cliente se conecta ao servidor. Assim, com esse mapa de soquetes, podemos enviar apenas para os clientes do conjunto.

Um outro detalhe quanto ao processo de Notify é que o próprio cliente que enviou não deve receber a própria mensagem, mesmo que esteja inscrito em alguma tag que utilizou na mensagem. Assim, antes de fazer a notificação a todos os subscritores, temos que remover do conjunto o remetente, se ele estiver lá.

2.4. Mensagens Especiais

Há algumas mensagens especiais, como requerido na especificação. Isso é tratado logo no processamento da mensagem. Recebendo uma mensagem que inicia com ”+” ou com ”-”, o servidor repassa essa mensagem para as funções de subscribe e unsubscribe. Se for uma mensagem ”##kill”, ele encerra sua execução (detalhes sobre a desconexão, na seção sobre isso).

É importante ressaltar que antes de qualquer processamento da mensagem, logo após seu recebimento, checa-se a validade dos caracteres. Caso haja algum inválido, segundo a especificação, o cliente que o utilizou é desconectado.

2.5. Desconexão

Para tratar desconexões súbitas, devido ao uso do CTRL+C, por exemplo, utilizávamos o retorno da função *recv* para detectar tal situação. Se um cliente encerra sua execução, o soquete é encerrado pelo processo e o retorno da função é 0, indicando essa desconexão. Quando o servidor detecta esse fim da conexão, ele deve, ainda, remover o cliente tanto do mapa de tags quanto do mapa de soquetes, já que o servidor não precisa armazenar informações de clientes que não estejam conectados.

Se o servidor encerra sua execução, quando um cliente requerer isso, por exemplo, ele pode simplesmente sair e os soquetes com todos os clientes serão fechados, fazendo com que a função *recv* dos clientes retorne 0, indicando a desconexão. Detectando-se isso, o cliente em seguida encerra sua execução.

2.6. Multithreading

Como pedido pela especificação, o servidor deveria poder se conectar com vários clientes e receber mensagens de vários concomitantemente. Isso foi feito via multithreading utilizando a biblioteca *pthread* na implementação de [Cunha 2020].

Além disso, o cliente também deveria ter multithreading para poder receber mensagens do servidor e receber dados do teclado advindos da entrada do usuário. A solução implementada foi semelhante, usando a biblioteca já citada. Um detalhe é que foi necessário utilizar uma função que requisitava que a thread de entrada do teclado esperasse a entrada do usuário para seguir sua execução. Caso contrário, ela não possibilitava que o usuário digitasse qualquer coisa.

3. Desafios

Além das questões das várias funcionalidades, bem como as soluções adotadas, já discutidas anteriormente, convém ressaltar alguns outros desafios constantes na implementação do projeto.

Um primeiro que merece ser ressaltado é uma certa dificuldade em se depurar um programa em rede, já que normalmente são mais de um executando, pelo menos um servidor e um cliente e, no nosso caso, havia, ainda, o multithreading. Desse modo, como inclusive denotado na aula [Cunha 2020], registrar as diversas ações realizadas num log, sobretudo no servidor, se mostrou muito útil.

O outro ponto é a programação em várias threads, algo que ainda não tinha sido utilizado em projetos até o momento. É uma ferramenta importante, que propicia a simultaneidade necessária em muitas aplicações em rede.

References

Cunha, I. (2020). Introdução à programação em redes. [1]. Playlist no Youtube, Acesso em: 04 jan 2021.

Sinha, A. (2019). Socket programming in c/c++. [2]. Acesso em: 04 jan 2021.

Tomar, N. (2019). Chatroom-in-c. [3]. GitHub repository, Acesso em: 04 jan 2021.

[1] https://www.youtube.com/playlist?list=PLyrH0CFXIM5Wzmbv-lC-qvoBejsa803Qk
[2] https://www.geeksforgeeks.org/socket-programming-cc/?ref=lbp
[3] https://github.com/nikhilroxtomar/Chatroom-in-C