Trabalho Prático 1 - Redes de Computadores
Servidor de Mensagens Publish/Subscribe

Feito por: Gustavo Dias de Oliveira

Introdução

A proposta do Trabalho Prático foi desenvolver o código de um cliente e servidor de troca de mensagens que utiliza o paradigma Publish/Subscribe. Os clientes podem inscrever e desinscrever de tags, além de enviar mensagens sobre uma determinada tag. O trabalho do servidor é monitorar as inscrições dos clientes e repassar as mensagens enviadas com uma tag para seus respectivos inscritos.

Implementação e Desafios

Para a implementação, primeiramente começou-se escrevendo o código de um cliente e servidor simples seguindo o modelo aprendido na aula Programação em Redes disponibilizada pelo professor. O cliente conseguia enviar mensagens ao servidor e o servidor, apesar de rústico, conseguia receber mensagens de mais de um cliente e, por utilizar threads, não dependia da ordem de conexão.

Levando em conta o paradigma Publish/Subscribe, o primeiro desafio encontrado foi que o cliente deveria sempre estar atento a mensagens transmitidas pelo servidor e poderia mandar mensagens a qualquer momento. Dessa forma, foram criadas duas threads para o cliente, uma para receber mensagens e outra para enviar mensagens. Entretanto, como as threads foram criadas logo após que a conexão era estabelecida, ocorreu um imprevisto: o programa era finalizado imediatamente depois da criação delas. Inicialmente, implementou-se um laço que era executado repetidamente até que as threads alterassem uma variável global que permitiria a finalização do programa.

```
while (true)
{
   if (close_prog_thread_send && close_prog_thread_recv)
   {
      break;
   }
}
close(sock);
exit(EXIT_SUCCESS);
```

No entanto, após analisar a biblioteca <*pthread.h*> passou-se a utilizar a função *pthread_join* que esperava pela finalização de uma thread para continuação do programa. Uma solução que torna o código melhor escrito.

```
pthread_join(send_thread, NULL);
pthread_join(recv_thread, NULL);

close(sock);
exit(EXIT_SUCCESS);
```

Com o problema resolvido, continuou-se a implementação do trabalho. A próxima tarefa implementada foi o tratamento das mensagens por parte do servidor e identificar o tipo de mensagem

que estava sendo enviada. Para isso, criou-se uma função que percorre a mensagem para identificar caracteres como ‘+’, ‘-’ e ‘#’. Com a identificação do tipo de mensagem pronta, a próxima tarefa foi implementar o que cada tipo de mensagem deveria fazer.

Para esta tarefa, o maior desafio foi como seria armazenada as tags de interesse de cada cliente. Para solucionar o problema, o servidor deve armazenar cada cliente conectado e suas inscrições em um *std::vector*. Em mais detalhes, adicionou-se um atributo para armazenar as tags dentro da *struct client_data,* que guarda os dados do cliente, e criou-se um std::vector global que recebe um ponteiro para tais structs.

```
struct client_data
{
    int c_sock;
    struct sockaddr_storage storage;
    std::vector<std::string> interest_topics = {};
};

std::vector<struct client_data *> clients_channels = {};
```

Dessa forma, sempre que o servidor aceitava a conexão de um cliente, uma *struct client_data* é criada e armazenada no vetor.

Assim, sempre que o cliente manda uma mensagem de adicionar tag (‘+’) ou remover tag (‘-’), o servidor checa o vetor de *strings* da *struct client_data* referente ao cliente que enviou a mensagem, e adiciona ou remove a tag do vetor.

Já para as mensagens de Broadcast (‘*#tag*’), o servidor extrai as tags e, para cada cliente no vetor *clients_channels*, ele compara as tags que ele tem interesse com as tags da mensagem a ser repassada. Se o cliente possuir uma das tags, o servidor repassa a mensagem para ele.

Após terminar essa funcionalidade, concluiu-se uma versão de cliente e software que consegue realizar as operações do paradigma Publish/Subscribe especificadas na descrição do trabalho.

```
>> Server 32-bit IP address (IPv4)
>> Server waiting for connections...
>> Client connected from: 127.0.0.1 54484
>> Client connected from: 127.0.0.1 54486
Received: +dota
Received: +dota
Received: jogo bom #dota
Received: +lol
Received: jogo top #lol
Received: -lol
```

```
>> Connecting to server: 127.0.0.1 / port: 3131 ...
>> 32-bit IP address (IPv4)
>> Connection established with success
+dota
Subscribed to +dota
+dota
Already subscribed to +dota
jogo bom #dota
jogo top #lol
```

```
>> Connecting to server: 127.0.0.1 / port: 3131 ...
>> 32-bit IP address (IPv4)
>> Connection established with success
jogo bom #dota
+lol
Subscribed to +lol
jogo top #lol
-lol
Unsubscribe to -lol
```

Com essa funcionalidade pronta, passou-se para a implementação de regras mais específicas. Implementou-se a funcionalidade de encerramento dos clientes e servidor quando é enviada a mensagem ‘##kil’. Após isso, criou-se a função para checagem de caracteres da mensagem recebida. Com base na tabela ASCII, analisou-se as mensagens recebidas e, para os clientes que desrespeitarem as regras especificadas na implementação, eram desconectados.

Interpretação do Protocolo

- As flags (‘+’) e (‘-’) devem aparecer uma única vez por mensagem enviada e devem estar seguidas apenas do nome da tag a ser adicionada ou excluída. Exemplo: ‘+dota’, ‘-lol’.
- A flag (‘#’) para fazer broadcast de uma mensagem deve possuir um espaço na frente e um espaço ou final de mensagem no final. Da forma: ‘jogo bom #dota’, ‘ #dota jogo bom’ e ‘jogo #dota bom’. Mas a tag ‘#dota jogo bom’ não é considerada.

Conclusão

Por fim, pode-se concluir que o trabalho foi de grande importância para o aprendizado de Redes de Computadores uma vez que exploramos o paradigma de comunicação Publish/Subscribe, além de utilizar a biblioteca POSIX para criar threads, sockets e realizar o envio e recebimento mensagens. Além disso, trabalhamos com lógica computacional para implementar o funcionamento geral da aplicação, e com estrutura de dados para armazenar os clientes conectados e suas informações.