Trabalho Prático 2: Troca de mensagens entre multiusuários.

Redes de Computadores

Thiago Cleto Miarelli Piedade

Matrícula: 2020007058

1. Introdução

A comunicação em tempo real entre usuários através de uma rede de computadores tem desempenhado um papel fundamental nas interações sociais e profissionais. Nesse contexto, este trabalho propõe a implementação de uma sala de chat utilizando conceitos de redes de computadores, empregando tecnologias como POSIX, Threads, Sockets e TCP. O objetivo é permitir que clientes se comuniquem com um servidor central, possibilitando o envio de mensagens privadas entre si e a realização de broadcasts para todos os participantes da sala.

2. Configuração da conexão

A conexão foi feita utilizando a estrutura de Sockets da biblioteca *sockets.h*. Para isso, utilizei as informações de protocolo (IPv6 ou IPv4) e porta que a comunicação se daria. A configuração do socket diferiu entre o servidor e o cliente.

No cliente, encapsulei a configuração do socket na função `setup_client()`. Ela é responsável por chamar o parser do input, converter o endereço de string para `sockaddr_storage` e chamar a conexão para esse endereço.

Já no servidor, a lógica é parecida, com o setup encapsulado na função `setup_server()`, mas com a única diferença de, ao invés de pedir para conectar, ele faz o bind da porta e fica escutando por pedidos de conexão.

O envio de informação é todo feito por strings e é manipulado pelas funções `sendMessage()` e `receiveMessage()`. Elas recebem o socket e realizam validações do sucesso do envio ou recebimento.

3. Gerenciamento de múltiplos clientes pelo servidor

O servidor de arquivos realiza a comunicação entre vários clientes por meio da criação de uma thread individual para cada cliente. O sistema gerencia um estado global com três variáveis principais: o número de clientes conectados naquele momento, a contagem de ID e uma lista de structs `clients`.

O servidor é configurado para gerenciar, no máximo, **15 conexões simultâneas na sala de chat**. Ele realiza a conexão com todos os servidores que fazem a requisição. Contudo, caso haja o número máximo de clientes, ele retorna a mensagem `close(03)` e fecha a conexão.

Cada cliente tem três informações: seu IP, sua thread e seu socket de comunicação. Eles ficam guardados em uma lista encadeada em que implementei métodos de busca, listagem, inserção e remoção. Esta estrutura de dados foi escolhida devido ao fato de facilitar a reorganização em exclusões no meio do array.

Na comunicação entre dois clientes de forma privada usei a função padrão `send_message(message, socket)` e uma busca por id na lista encadeada. Contudo, para facilitar a transmissão em massa, criei a função `broadcast(message, userList, exception)`, em que ela envia a mensagem para todos os usuários da lista, exceto o do id da exceção. Essa função é usada para a função de transmissão para todos e para registros de entrada e saída de usuários no grupo.

Quando uma mensagem direta é enviada, o servidor confere se o ID existe. Caso contrário, ele retorna `error(01).`

4. Recebendo mensagens e recebendo comandos no cliente

O cliente tem uma característica específica que ele precisa fazer duas coisas ao mesmo tempo: receber comandos de forma `stdin` e ficar escutando mensagens do servidor que ele está conectado. Para atingir esse objetivo, criamos duas threads que executam paralelamente: a thread ativa, que recebe os comandos dos usuários e a thread passiva, que recebe as mensagens do servidor e executa as ações necessárias.

Um parser usado pelo servidor e cliente quebra as mensagens, enviadas como string. Ele retorna o código do comando, e, se houver, os argumentos. O cálculo das ações de cada comando fica fora do parser, visto que a ação dentro do cliente difere do servidor.

Além disso, não há lógica nas mensagens. Por exemplo, quando enviamos uma mensagem em broadcast, os clientes diferentes de quem enviou devem receber como uma mensagem comum:

```
[01:32] 2: Bora tomar uma?
```

Contudo, a mensagem chega de uma forma diferente para o remetente:

```
[01:32] -> all Bora tomar uma?
```

A mensagem que chega para todos os clientes é a mesma:

```
MSG(2,-1,"Bora tomar uma?")
```

Quando recebe isso, o cliente remetente identifica que o ID do remetente é o mesmo seu e formata a mensagem corretamente.

Os comandos aceitos pelo cliente são:

send to <id> “<message>”: envia uma mensagem privada para o cliente especificado em <id>;

send all “<message>”: faz broadcast de uma mensagem para todos os clientes ativos

list clients: imprime todos os clientes salvos na base do cliente

close connection: envia um pedido de desconexão para o servidor que remove o cliente da sua lista e envia um broadcast para que todos os clientes façam o mesmo.

5. Padrão de mensagens

As mensagens seguem um padrão: caso não haja argumentos a mensagem é apenas o parâmetro e, caso haja, eles são colocados separados por vírgula dentro de parênteses. Veja:

Requisição de conexão: `REQ_ADD`

Requisição de desconexão: `REQ_REM(ID)`

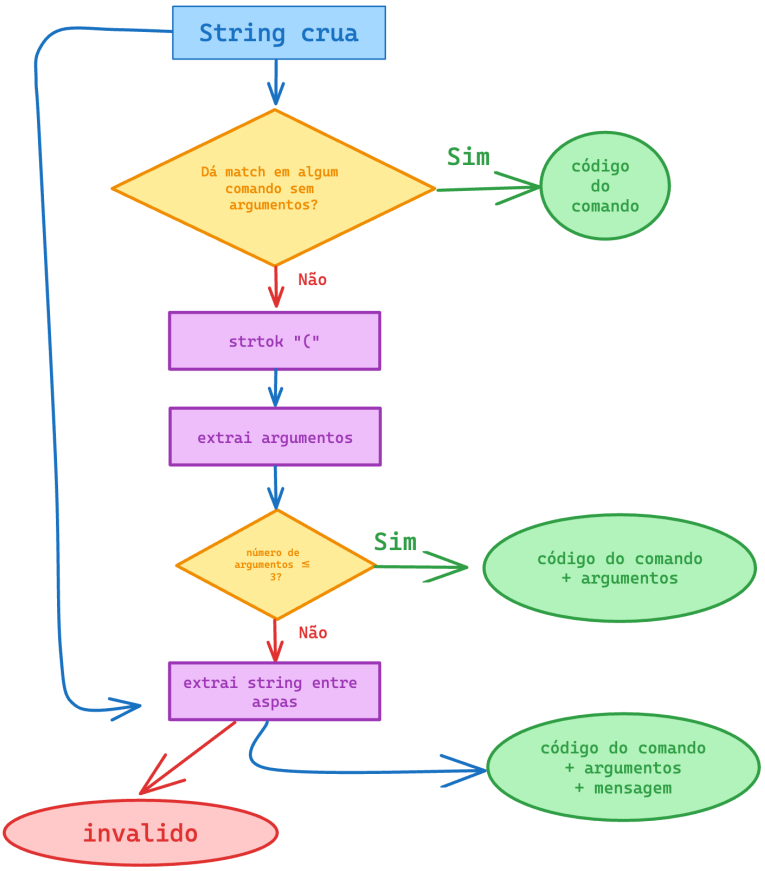
Requisição de mensagem: `MSG(ID_remetente, ID_destinatario ou -1, mensagem)`

Feedback de erro: `ERROR(codigo)`

Feedback de sucesso: `OK(id)`

6. Parser

Boa parte do tempo investido no trabalho prático foi implementando tratamento de quebra de string. O processo foi o seguinte:



7. Como executar

Para executar o sistema, use os seguintes comandos em um computador com Linux:

1. Execute make clean para remover executáveis prévios
2. Execute make para compilar os dados do sistema
3. Inicialize o servidor com `./server <v4/v6> <port>`, sendo `<v4/v6>` a escolha do protocolo e `<port>` a porta do sistema que usaremos para nos conectar.
4. Do lado do cliente, inicialize-o com `./server <address> <port>`, sendo `<address>` o endereço IP local e `<port>` a porta usada no servidor. Via de regra temos os seguintes endereços de IP:

   IPv4: `127.0.0.1` **IPv6:** `::1`

5. Use os comandos para manipular dados no cliente:
   a. `send to <id> <message>`: seleciona um cliente para envio da mensagem e envia. Caso o cliente não exista, o servidor retorna erro.
   b. `send all <message>`: Envia a mensagem para todos os clientes
   c. `list users`: Envia a mensagem para todos os clientes
   d. `close connection`: desconecta o cliente

8. Conclusão

O projeto consistiu na implementação de um sistema de transferência de mensagens entre múltiplos clientes utilizando a API POSIX e a biblioteca sockets.h no ambiente Linux. Foram desenvolvidas funcionalidades tanto para os clientes quanto para o servidor. O cliente era capaz de configurar a conexão com o servidor, enviar mensagens para outro cliente específico e fazer broadcast e encerrar a conexão, enquanto o servidor aguardava por pedidos de conexão, gerenciava a quantidade de clientes e encerrava as conexões conforme necessário.