SERVIDOR IRC

1 Introdução

O IRC (*Internet Relay Chat*) foi um dos primeiros protocolos para troca de mensagens instantâneas pela internet. Lançado em 1988, ele ainda hoje é amplamente usado para reunir a comunidade de diversos projetos de software livre. Nesta prática, vamos implementar um servidor compatível com um subconjunto do protocolo IRC.

O protocolo IRC é definido no documento RFC 2812. Você pode usar esse documento como referência, mas vamos fazer diversas simplificações para tornar a prática mais fácil.

Após dar uma visão geral do protocolo, este roteiro te guiará em passos para que você construa aos poucos a implementação que esperamos.

2 Protocolo

2.1 Visão do usuário

O usuário escolhe um apelido (*nickname*) logo ao conectar. Caso o apelido já esteja em uso por outra pessoa, o servidor retorna uma mensagem de erro, situação na qual o cliente deve escolher outro apelido. O usuário é considerado como "registrado" no servidor apenas após escolher um apelido válido e único.

Uma vez "registrado", o usuário pode enviar mensagens privadas para outros usuários, ou ingressar (*join*) em canais e enviar mensagens para estes. Os canais são grupos de pessoas — uma mensagem enviada para um canal é recebida por todos os outros participantes do canal. O usuário também pode sair (*part*) de um canal quando quiser parar de receber mensagens enviadas a este.

Um usuário só é notificado a respeito de mudanças de apelido ou da saída de outro usuário se estiver pelo menos em um canal em comum com ele.

2.2 Mensagens do protocolo

As mensagens trocadas pelo protocolo são sempre de uma única linha terminada por `'\r\n'`. O seu servidor deve ser capaz de interpretar os seguintes tipos de mensagem:

1. `NICK apelido`: Define o apelido inicial ou troca o apelido atual. Pode gerar as seguintes mensagens como resposta:
   (a) `:server 432 apelido_atual apelido :Erroneous nickname` se o apelido escolhido não for válido. Consideraremos como válido um apelido que começa com uma letra e que contém apenas letras, números, `_` e `-`. Você pode usar a seguinte função em Python para fazer essa validação:

   ```
   def validar_nome(nome):
       return re.match(br'^[a-zA-Z][a-zA-Z0-9_-]*$', nome) \
              is not None
   ```

   (b) `:server 433 apelido_atual apelido :Nickname is already in use` se o apelido escolhido já estiver em uso por outro cliente.
   (c) `:server 001 apelido :Welcome` seguido de `:server 422 apelido :MOTD File is missing` se o apelido inicial for definido com sucesso, para indicar que o usuário agora está "registrado" no servidor.

(d) `:apelido_antigo NICK apelido` se o usuário trocar de apelido com sucesso (não deve ser enviado se o usuário estiver definindo o apelido inicial!). Envie essa mensagem, também, para usuários que estejam em pelo menos um canal em comum com o usuário que trocou de apelido.

Caso o usuário esteja tentando definir o apelido inicial, o valor adotado para `apelido_atual` nas mensagens de erro é `*`

2. `PING payload`: Verifica se o servidor está respondendo. O servidor deve responder com `:server PONG server :payload`
3. `PRIVMSG destinatário :conteúdo`: Envia uma mensagem de texto para um usuário ou para um canal. O nome de canais começa sempre com `#`, então o servidor consegue distinguir pelo primeiro caractere de `destinatário` se este é um canal ou se é um usuário. O servidor não deve responder nada para o remetente, mas deve enviar a mensagem da seguinte forma para o destinatário:
   `:remetente PRIVMSG destinatário :conteúdo`
4. `JOIN canal`: Ingressa no canal. O nome do canal deve começar com `#` e os demais caracteres devem seguir as mesmas regras dos nomes de apelidos. Caso o nome do canal seja inválido, responda com `:server 403 canal :No such channel`. Senão:
   (a) Envie `:apelido JOIN :canal` para o usuário que entrou e para todos os usuários que já estão dentro do canal.
   (b) Envie `:server 353 apelido = canal :membro1 membro2 membro3` para o usuário que entrou no canal. A lista de membros deve incluir o apelido do usuário que acabou de entrar e deve estar ordenada em ordem alfabética. Se a lista de membros for muito grande, quebre em várias mensagens desse tipo, cada uma com no máximo 512 caracteres (incluindo a terminação `'\r\n'`).
   (c) Envie `:server 366 apelido canal :End of /NAMES list.` para o usuário que entrou no canal, a fim de sinalizar que todos os pedaços da lista de membros já foram enviados.
5. `PART canal`: Sai do canal. Envie `:apelido PART canal` para todos os membros do canal, incluindo o que pediu para sair.

Algumas observações importantes:

1. Apelidos e nomes de canais devem sempre ignorar maiúsculas / minúsculas.
2. Se um cliente fechar a conexão com o servidor, deve-se avisar todos os outros usuários que estejam em pelo menos um canal em comum com ele, enviando:
   `:apelido QUIT :Connection closed`

3 Cliente

Existem duas formas recomendadas de testar o seu servidor:

1. Usando o HexChat. Após instalá-lo, adicione uma nova rede de IRC e insira nela o servidor `localhost/6667`.
2. Usando o netcat ou telnet e construindo na mão os comandos do protocolo IRC, como definidos na seção anterior. Execute algum dos seguintes comandos:
   (a) `nc localhost 6667`
   (b) `telnet localhost 6667`

4 Servidor

Implemente o seu servidor no arquivo `servidor`. Os exemplos que daremos são em Python, mas você pode usar a linguagem que quiser. Se você usar uma linguagem compilada, como C ou Rust, crie um shell script chamado `compilar` contendo os comandos necessários para compilar seu código. Se precisar de algum compilador que não estiver instalado no servidor, entre em contato com o professor para verificar a possibilidade de instalá-lo.

Para testar seu código, execute `./autograde.py`.

Passo 1

Você recebeu um arquivo de exemplo chamado `servidor` que escuta na porta 6667 e mostra na saída todas as conexões estabelecidas, dados recebidos e conexões fechadas.

Estude o código contido no arquivo `tcp.py` para entender como o servidor utiliza os recursos do sistema operacional e da linguagem Python.

Complete o código do servidor para tratar mensagens do tipo `PING` recebidas do cliente e respondê-las corretamente.

Passo 2

Um erro muito comum de pessoas que estão começando a trabalhar com *sockets* é acreditar que uma mensagem sempre vai ser transportada de uma só vez de uma ponta até a outra. Essa situação ideal pode até persistir enquanto você estiver testando seu programa localmente, mas quando ele estiver funcionando em uma rede real, duas situações eventualmente acontecerão:

1. Uma mensagem do tipo `"linha\r\n"` pode ser quebrada em várias partes. Por exemplo, podemos receber primeiro `"lin"`, depois `"h"` e depois `"a\r\n"`.
2. Duas ou mais mensagens podem ser recebidas de uma só vez. Por exemplo, podemos receber `"linha 1\r\nlinha 2\r\nlinha 3\r\n"`.

As duas coisas também podem acontecer ao mesmo tempo. Podemos receber, por exemplo, algo do tipo `"a 1\r\nlinha 2\r\nli"`.

Adapte seu servidor para tratar situações similares às descritas acima. Você deve cortar os dados recebidos nas quebras de linha e tratar múltiplas mensagens sempre que várias mensagens forem recebidas de uma vez só. Sempre que os dados recebidos não terminarem em um fim de linha, você deve armazenar os "dados residuais" para juntar com os dados que serão recebidos na próxima vez que a função for chamada. Recomendamos que você armazene os "dados residuais" com um atributo do próprio objeto `conexao`.

Por enquanto, continuaremos enviando apenas mensagens do tipo `PING`, mas tente deixar seu código organizado para implementar o tratamento de novos tipos de mensagem, que serão necessárias nos passos seguintes.

Passo 3

Trate mensagens do tipo `NICK`. Verifique se o apelido solicitado é válido usando a função `validar_nome`. Se for inválido, responda com a mensagem de erro 432 (como descrita na Seção 2). Senão, responda com as mensagens 001 e 422, para indicar sucesso.

Passo 4

Adicione alguma estrutura de dados (por exemplo, um dicionário) para mapear cada apelido para a conexão correspondente. Quando um usuário pedir para definir um apelido, verifique se esse apelido já está em uso. Se estiver em uso, responda com a mensagem de erro 433 (vide Seção 2). Implemente também o suporte a trocar de apelido após definido um apelido inicial.

Não esqueça que devemos ignorar maiúsculas / minúsculas quando tratamos apelidos ou nomes de canais. Assim, `ApElido` deve ser considerado duplicado se alguém já estiver usando `apelido`.

Passo 5

- Implemente suporte a troca de mensagens entre usuários usando `PRIVMSG`.

  O seu código só vai passar no teste se você estiver acompanhando corretamente a troca de apelidos e direcionando as mensagens à conexão correta.

  Ignore mensagens enviadas para apelidos que não existem ou que não estão mais em uso.

Passo 6

- Implemente o suporte a entrar em canais com `JOIN` e a enviar mensagens para esses canais usando `PRIVMSG`.

  Por enquanto, você não precisa enviar a lista de membros do canal (mensagens 353 e 366).

  Não esqueça que devemos ignorar maiúsculas / minúsculas quando tratamos apelidos ou nomes de canais. Assim, `#cAnaL` deve ser considerado a mesma coisa que `#canal`.

Passo 7

- Implemente o suporte a sair de canais com `PART`.

Passo 8

- Quando uma conexão fechar, envie mensagens do tipo `QUIT` para todos os usuários que estiverem em pelo menos um canal em comum com o usuário que fechou a conexão.

  Você pode implementar essa funcionalidade na função `sair` do código de exemplo.

Passo 9

- Passe a enviar a lista dos membros de um canal (mensagens 353 e 366) quando alguém entrar no canal (`JOIN`).

Passo 10

- Certifique-se que, quando um usuário fecha a conexão, você está retirando o nome dele da lista de membros dos canais dos quais ele fazia parte.