Entendendo o socket no Python criando um bot de IRC



Yan Orestes

02/04/2019

[Formação Iniciante em Programação](https://www.alura.com.br/formacao-programacao)

Aqui na empresa em que eu trabalho, usamos um sistema de comunicação interna que, por alguns, já é considerado um tanto antiquado. É o [**IRC**](https://pt.wikipedia.org/wiki/Internet_Relay_Chat), um protocolo de comunicação que funciona, principalmente, como bate-papo por texto, e que teve seu *boom* lá pros anos 90. Com ele, podemos mandar mensagens privadas e conversar em grupos nos denominados **canais**.

Se é tão antigo, por que escolhemos continuar usando ele? A questão é que, mesmo antigo, ele ainda funciona muito bem! O protocolo é muito simples e leve, além de dar espaço para modificações e extensões diversas, o que torna seu uso bastante desejável.

Além de uma própria rede de IRC interna nossa, usamos um canal na conhecida rede [**Rizon**](https://rizon.net/), o #minha-empresa, para facilitar a comunicação com clientes.

Este canal é nosso principal meio de comunicação com os clientes. O problema é que estávamos com poucos funcionários para acompanhar todo o conteúdo das mensagens de lá a todo momento.

Frente a esse problema, nosso time de tecnologia me encarregou de criar algo que poderia nos ajudar com a comunicação, não decepcionando os clientes - um [**bot de IRC**](https://pt.wikipedia.org/wiki/Rob%C3%B4_de_IRC) que, automaticamente, mandaria uma mensagem de boas vindas a todo mundo que entrasse no canal.

A ideia é bacana! Mas como podemos implementar isso? Como fazer um programa que se conecte ao IRC e ainda envie uma mensagem específica para todos que se conectem a um determinado canal?

**Conhecendo os *sockets***

A primeira coisa que precisamos conseguir fazer com nosso programa robô é que ele entre em nosso canal #minha-empresa lá dentro da Rizon.

A questão é que, antes de entrarmos no canal em específico, precisamos começar alguma comunicação com o próprio servidor da Rizon. Mas como?

Nosso bot estará rodando em um processo em um servidor próprio da empresa, enquanto o servidor de bate-papo IRC da Rizon funciona em um processo completamente diferente, fisicamente em outra(s) máquina(s). De alguma maneira, precisamos fazer uma comunicação entre esses processos.

Tratando-se de comunicação entre dois processos diferentes, a saída quase sempre é a mesma - [**soquetes**](https://pt.wikipedia.org/wiki/Soquete_de_rede), ou, originalmente, *sockets*. Apesar de abstrato, esse conceito aparece para todos nós, usuários da Internet, cotidianamente!

O nosso navegador, por exemplo, utiliza *sockets* para fazer **todo tipo de requisição web**. Quem trabalha com desenvolvimento de software ainda se depara com outros múltiplos exemplos, como na [**conexão a um servidor via SSH**](https://www.alura.com.br/artigos/como-acessar-servidores-remotamente-com-ssh/) ou integração de um sistema a um banco de dados.

Tudo isso necessita do uso de *sockets* para efetuar a comunicação entre dois processos diferentes (que, na verdade, podem até estar em máquinas diferentes!).

Então como é que o IRC entra nisso? A questão é que o protocolo IRC funciona com base no modelo [**cliente/servidor**](https://pt.wikipedia.org/wiki/Cliente-servidor), no qual os usuários usam uma aplicação cliente para, através de um **socket**, se conectar ao servidor. Em nosso caso, o bot é justamente a aplicação cliente! Mas como podemos fazer isso com o Python?

**Criando sockets com o módulo socket no Python**

Assim como diversas outras linguagens de programação, o Python tem algumas especificações que nos permite trabalhar com *sockets*. No caso, temos um [**módulo inteiro**](https://docs.python.org/3/library/socket.html), o próprio socket.

O módulo socket abriga uma classe essencial, que representa, justamente, um *socket*. A classe tem o mesmo nome do módulo - socket. Assim, instanciá-la não deve ser muito difícil, certo? Basta importamos o módulo para nosso programa e chamarmos a função construtora dessa classe:

import socket

s = socket.socket()

Testando o código, vemos que ele realmente funciona! É tão simples assim, então? Bem… mais ou menos!

Na realidade, não basta falarmos que queremos um *socket* para ele magicamente aparecer. Precisamos especificar algumas definições. Afinal, como o computador vai saber como o *socket* criado deve funcionar?

O que acontece é que, por padrão, o Python passa isso tudo automaticamente através de parâmetros padrões no construtor. Vamos entender isso.

**Especificações do socket**

Em primeiro lugar, de alguma maneira o computador precisa saber, ao criar um *socket*, com o que ele vai conseguir se comunicar. Como já vimos, *sockets* podem ter funcionalidades bem diversas - podendo se comunicar com processos dentro ou fora de uma mesma máquina.

Com o nosso bot, estamos tratando de comunicação remota. Nesse caso, temos uma constante própria que especifica isso, chamada de **AF\_INET**.

Essa constante faz parte de um grupo denominado **famílias de endereços**, ou ***address families***, que constitui exatamente o primeiro parâmetro opcional do construtor socket. A **AF\_INET** abrange os endereços do tipo [**IPv4**](https://pt.wikipedia.org/wiki/IPv4), antigo padrão da Internet e, por isso, pode ser ideal para nós!

Além da família de endereços, precisamos especificar mais uma coisa - **o tipo do socket**. Como assim? O que acontece é que, na Internet, existem diversas formas e modelos para se efetuar uma comunicação. Assim, precisamos definir um, ao criarmos nosso objeto socket.

Em relação ao tipo do socket, existem duas constantes mais importantes - a **SOCK\_STREAM**, que define *sockets* de fluxo, e a **SOCK\_DGRAM**, que define *sockets* de datagrama. Mas, na prática, qual a diferença delas duas?

Basicamente, um *socket* de fluxo se refere ao protocolo **TCP**, enquanto um *socket* de datagrama, ao **UDP**. As diferenças entre TCP e UDP não são poucas, mas, no nosso caso, não precisamos ir tão fundo.

Queremos estabelecer uma **conexão** com o servidor do IRC. Entretanto, no modelo UDP não há conexão! É um protocolo chamado de *connectionless*, por isso. Mas como não há conexão?

O que acontece é uma troca de pacotes desordenada e pouco confiável. A vantagem fica na alta velocidade, como já discutimos antes.

No caso de um sistema de chat por texto, as desvantagens citadas acabam sendo extremamente negativa, tirando o sentido do uso desse protocolo. Assim, por padrão, o protocolo IRC usa o **TCP** e, assim, a constante **SOCK\_STREAM**.

Agora já podemos criar nosso objeto socket no Python, com muito mais especificação:

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

Agora que temos nosso objeto socket, basta nos conectarmos ao servidor da Rizon. Mas como?

**Se conectando ao servidor de IRC**

Com nosso objeto socket criado, partir para a parte de conexão não é difícil, basta um método - o connect(). E como o Python vai saber com quem se conectar? Bem, a princípio ele não sabe! Precisamos especificar isso através de um parâmetro na chamada do método.

Sockets da família AF\_INET necessitam apenas de um parâmetro, que é justamente o endereço com o que queremos nos conectar. Esse endereço é uma [**tupla**](https://www.alura.com.br/artigos/conhecendo-as-tuplas-no-python/) contendo o host que queremos nos conectar (pode ser desde um IP, a um *hostname*) e a porta pela qual queremos realizar a conexão.

Com uma rápida visita no [site oficial da Rizon](https://rizon.net/info/access), descobrimos que o *hostname* padrão dessa rede é o **irc.rizon.net** e a porta oficial, assim como é por padrão em outras redes de IRC, a **6667**. Assim, podemos efetuar a conexão:

s.connect(('irc.rizon.net', 6667))

Com a conexão estabelecida, podemos partir para a comunicação!

**Comunicação com o socket**

Agora que temos nosso objeto socket configurado e conectado a um servidor da Rizon, podemos começar nossa comunicação.

No geral, como clientes, começamos a comunicação checando se o servidor quer enviar algo pra gente.

Com um objeto socket, podemos fazer isso simplesmente pedindo para receber (*receive*), através do método recv(), que toma como parâmetro o número de bytes que queremos receber do servidor.

[**A própria documentação do Python**](https://docs.python.org/3/library/socket.html#socket.socket.recv) nos recomenda usar, nesse parâmetro, uma pequena potência de 2, "para melhor compatibilidade com as realidades de hardware e rede". Assim, podemos usar um valor como **2048** (2¹¹):

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.connect(('irc.rizon.net', 6667))

dados = s.recv(2048)

print(dados)

*É importante notar que recv() é um****método blocante****, o que significa que todo o resto do programa vai esperar a execução dele para continuar rodando. Até recebermos algo, o programa vai continuar esperando.*

Rodando esse código, recebemos a seguinte resposta:

b":irc.falconspy.org NOTICE \* :\*\*\* Looking up your hostname...\r\n:irc.falconspy.org NOTICE \* :\*\*\* Couldn't look up your hostname\r\nERROR :Closing Link: 177.139.156.225 (Registration timed out)\r\n"

De fato é uma mensagem, mas desse jeito fica difícil de ler… por quê? Acontece que, como o b" no começo indica, recebemos os dados em [**bytes**](http://blog.caelum.com.br/quais-as-diferencas-entre-python-2-e-python-3/#tipos-texto), não em string!

Para melhorar a impressão, podemos decodificar os bytes em uma string com o método decode(), que recebe como parâmetro o *encoding* queremos usar. No caso, usaremos o [**UTF-8**](https://pt.wikipedia.org/wiki/UTF-8), por suportar [**Unicode**](https://pt.wikipedia.org/wiki/Unicode):

msg = s.recv(2048).decode('UTF-8')

print(msg)

Agora sim, olha o que apareceu:

:irc.falconspy.org NOTICE \* :\*\*\* Looking up your hostname...

:irc.falconspy.org NOTICE \* :\*\*\* Couldn't look up your hostname

ERROR :Closing Link: 177.xxx.156.xxx (Registration timed out)

Hum… por que todos esses "NOTICE" e ainda essa mensagem de erro, no final? A mensagem avisa que a conexão está sendo fechada porque o tempo de registro expirou. Ué, fizemos algo errado? Que registro é esse?

**Burocracias do IRC**

O IRC é um protocolo há vários anos já bem definido, que especifica muito bem como uma conexão deve ser trabalhada com esse protocolo.

No [**seu primeiro RFC**](https://tools.ietf.org/html/rfc1459), foi definido padrões de registro para todo usuário do IRC, uma série de passos que todo cliente deveria seguir ao se conectar a uma rede de IRC.

Esses passos se resumem a três comandos, nessa ordem:

* **PASS**, a senha de registro do nickname em uso;
* **NICK**, o nickname que será usado e
* **USER**, que especifica outros detalhes do usuário (como nome verdadeiro).

Sabendo disso, agora podemos fazer o registro!

**Registrando um usuário**

Em primeiro lugar, precisaríamos enviar a senha. Como nem temos nick registrado, podemos simplesmente ignorar esse passo e partir para o comando **NICK**, com o nickname que queremos usar, que pode ser **BotDaEmpresa**.

A princípio, é só enviar o comando NICK BotDaEmpresa para o servidor da Rizon, através de nosso socket, mas na realidade não é bem assim… O que acontece é que o protocolo IRC carrega mais algumas burocracias para funcionar.

Uma delas, por exemplo, diz respeito a tudo que é enviado do cliente para o servidor - todos os comandos devem terminar com a sequência de caracteres especiais '\r\n', representando um **ENTER**, o envio. Agora, podemos usar o método send() de nosso objeto socket para enviar o comando:

nick = 'BotDaEmpresa'

comando\_nick = f'NICK {nick}\r\n'

s.send(comando\_nick)

Quando rodamos um código… erro!

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: a bytes-like object is required, not 'str'

Assim como quando recebemos dados com nosso socket, precisamos enviá-los como bytes. O processo é similar ao que usamos quando decodificamos os bytes recebidos para uma string, mas dessa vez faremos o contrário, com o método encode().

nick = 'BotDaEmpresa'

comando\_nick = f'NICK {nick}\r\n'.encode('UTF-8')

s.send(comando\_nick)

Certo! Podemos, agora, enviar o último comando de registro - o **USER**. O comando **USER** toma quatro parâmetros (*username*, *hostname*, *servername* e *realname*), todos os quais podem ser resumidos pelo próprio nick que escolhemos.

Vale notar que esse último parâmetro, apesar de não ser importante para nós, deve ser precedido com um :, indicando que ele pode conter espaços (e que tudo depois do : faz parte de um único parâmetro):

nick = 'BotDaEmpresa'

comando\_nick = f'NICK {nick}\r\n'.encode('UTF-8')

s.send(comando\_nick)

comando\_user = f'USER {nick} {nick} {nick} :{nick}\r\n'.encode('UTF-8')

s.send(comando\_user)

Legal! Funciona, mas nosso código está começando a ficar feio, com tanta repetição da parte burocrática de enviar um comando…

Para simplificar o processo, podemos fazer algumas funções que fazem todo esse processo de envio de comandos e registro automaticamente:

def envia\_comando(sock, comando):

comando += '\r\n'

sock.send(comando.encode('UTF-8'))

def registra(sock, nick):

envia\_comando(sock, 'NICK {}'.format(nick))

envia\_comando(sock, 'USER {0} {0} {0} :{0}'.format(nick))

registra(s, 'BotDaEmpresa')

Além disso, após a efetivação da conexão, vou fazer um loop que me permita sempre ver o que estou recebendo do servidor, dessa forma:

while True:

msg = s.recv(2048).decode('UTF-8')

print(msg)

Será que agora vai? Rodamos o código e olha só o que apareceu na tela:

:irc.falconspy.org NOTICE \* :\*\*\* Looking up your hostname...

:irc.falconspy.org NOTICE \* :\*\*\* Couldn't look up your hostname

PING :3685587899

ERROR :Closing Link: 177.xxx.156.xxx (Registration timed out)

Olha só! Dessa vez apareceu uma mensagem diferente, esse tal de **PING** com alguns números estranhos. Ao final de tudo, ainda recebemos aquele mesmo erro. Mas e agora? Não conseguimos fazer o registro?

**O comando PING**

Na verdade, fizemos o registro adequadamente, e por isso mesmo recebemos a mensagem diferente de **PING**. O problema está justamente nela, ou, mais especificamente, no fato de a termos ignorado.

E o que significa esse **PING**? Esse tipo de comando é muito comum em sistemas que exigem a manutenção de uma conexão. Ele serve para que se teste a latência dela, isto é, quanto tempo demora para um conjunto de dados ser enviado de uma máquina a outra e receber uma resposta.

O IRC trabalha suas conexões através desse comando. De quando em quando, o sistema do servidor envia um comando **PING** seguido de uma mensagem (nesse caso, 3685587899) e aguarda a resposta do cliente com essa mesma mensagem. A diferença é que a resposta do cliente, em vez de ir com o comando **PING**, deve usar o **PONG**.

Se o servidor não receber a resposta do cliente em um determinado tempo, a conexão é efetivamente encerrada, por isso recebemos aquela última mensagem de erro.

Precisamos já implementar um tipo de automatização - sempre que recebermos um **PING**, precisamos responder com um **PONG**. E como fazer isso?

Em primeiro lugar, precisamos conseguir identificar quais mensagens recebidas contêm um **PING**. Como vimos, mensagens desse tipo contêm o texto PING : seguido pelos dados que precisamos enviar de resposta. Para evitar complicações, podemos simplesmente verificar se esse trecho de texto existe na mensagem com o conhecido operador in:

while True:

msg = s.recv(2048).decode('UTF-8')

print(msg)

if 'PING :' in msg:

pass

Verificando que a mensagem recebida é do tipo **PING**, podemos enviar de volta o comando **PONG**. Mas como vamos capturar o que recebemos do **PING**? Existem, de fato, diversas formas.

Uma opção mais simples é cortar nossa string de mensagem na parte do PING : e, então, pegar o que vem depois dessa substring. Podemos fazer isso com o método split(), que toma, como parâmetro, a parte da string em que queremos fazer a separação e retorna uma lista com todas as partes separadas.

O código que temos que devolver no **PONG** vai ser o que vem depois da separação, ou seja, o último elemento da lista, podendo ser acessado pelo índice **-1**:

if 'PING :' in msg:

codigo\_ping = msg.split('PING :')[-1]

resposta\_pong = 'PONG :{}'.format(codigo\_ping)

envia\_comando(s, resposta\_pong)

Podemos até separar em uma função, para melhorar a organização:

def checa\_ping(sock, msg):

if 'PING :' in msg:

codigo\_ping = msg.split('PING :')[-1]

resposta\_pong = 'PONG :{}'.format(codigo\_ping)

envia\_comando(sock, resposta\_pong)

while True:

msg = s.recv(2048).decode('UTF-8')

print(msg)

checa\_ping(s, msg)

Rodando esse código, olha o que recebemos, logo após o comando **PING**:

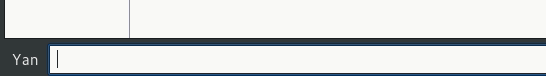
:irc.falconspy.org NOTICE BotDaEmpresa :\*\*\* Your host is masked (1ED94617.82A3866B.693A3A77.IP)

:irc.falconspy.org 001 BotDaEmpresa :Welcome to the Rizon Internet Relay Chat Network BotDaEmpresa

Deu certo! Podemos até checar se nosso bot está online usando, manualmente, o comando /whois BotDaEmpresa no IRC:

whois bot

Certo! O Bot está online :D! Podemos até enviar uma mensagem pra ele:



E olha o que aparece no terminal do bot:

:Yan!~Yan@1XXX17.82A3XXXB.69XXX77.IP PRIVMSG BotDaEmpresa :Oi, bot!

O bot recebeu a mensagem! Junto com ela, ele ainda recebeu várias informações acerca do meu nick e do meu usuário, o que vai nos ajudar bastante para todo o controle do programa. O único problema é que o bot ainda não está no canal #minha-empresa! E agora?

**Entrando no canal e respostas automáticas**

Entrar em um canal no IRC é, no geral, bem simples! Basta usarmos o comando **JOIN** . Como queremos que isso aconteça automaticamente, sempre que o bot é iniciado, podemos até colocar direto no código:

envia\_comando(s, 'JOIN #minha-empresa')

Quando executamos o bot novamente, temos a confirmação que deu certo!

\* BotDaEmpresa (~BotDaEmpr@1ED94617.82A3866B.693A3A77.IP) has joined

Ótimo! Agora só falta o bot mandar as mensagens de boas vindas aos novos no canal, o que acaba sendo fichinha pra gente, depois de tudo. Olha só o que aparece para o bot quando alguém entra no canal:

:Yan!~Yan@1ED94617.82A3866B.693A3A77.IP JOIN :*#minha-empresa*

Para checarmos se há um *join* no canal **#minha-empresa** nas mensagens recebidas pelo bot, podemos simplesmente usar um *in*, como fizemos com o **PING**. Para capturar o nick de quem entrou, basta o uso do método split(). Enfim, só precisamos enviar ao IRC o comando **PRIVMSG** especificando o canal, com a mensagem de boas vindas:

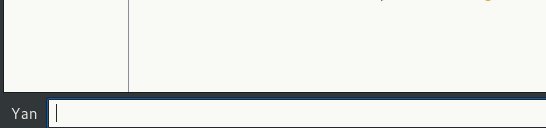
if 'JOIN :#minha-empresa' in msg:

nick\_que\_entrou = msg[1:].split('!')[0]

boas\_vindas = f'Olá, {nick\_que\_entrou}! Seja bem vindo :D'

envia\_comando(s, f'PRIVMSG #minha-empresa :{boas\_vindas}')

E olha só como deu certo agora:



**Sockets em todo lugar**

O conceito de socket, por ser um tanto abstrato, chega a assustar alguns desenvolvedores. Mesmo assim, todos nós provavelmente nos deparamos com sockets todos os dias, ao utilizarmos a Internet!

Nesse artigo, enfrentamos um problema empresarial de automatização no contato com clientes através do protocolo IRC.

Assim, passamos pela superfície da ideia e do uso dessa tecnologia no Python, entendendo o porquê não devemos temer esse conceito tão presente em nosso dia-a-dia.

Se gostou do post e quer aprender mais sobre a linguagem Python, dê uma olhada em nossos [**cursos da Alura**](https://www.alura.com.br/cursos-online-programacao/python)!