

# Sistemas Distribuídos - TP3 -

## Rock Album vol. 1 - Implementação utilizando socket

Leandro Lázaro Araújo Vieira - 3513<sup>1</sup>, Mateus Pinto da Silva - 3489<sup>2</sup>

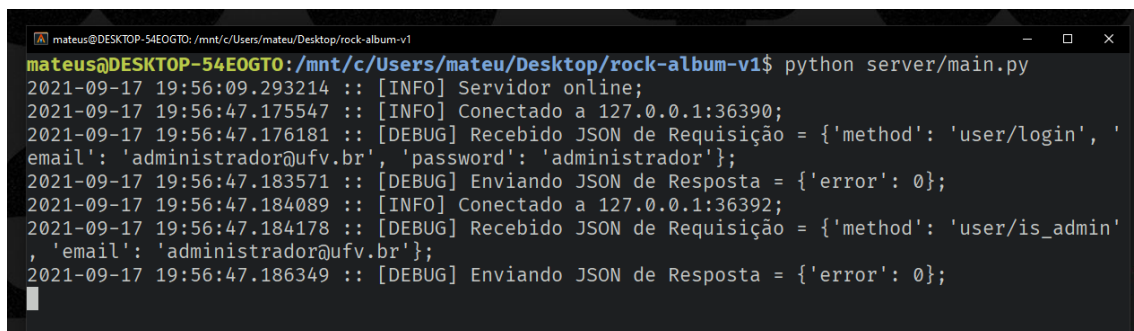
<sup>1</sup>Ciência da Computação – Universidade Federal de Viçosa (UFV-caf)  
– Florestal – MG – Brasil

(leandro.lazaro<sup>1</sup>, mateus.p.silva<sup>2</sup>)@ufv.br

**Resumo.** Implementamos a nossa especificação de sistema distribuído de álbum de figurinhas, o Rock Album vol. 1, utilizando a API de Socket. Optamos por fazer isso da forma mais simplificada o possível, utilizando a linguagem de programação Python e o banco de dados SQLite. A troca de mensagens ocorre via JSONs serializados e codificados em bytes através do código UTF-8, e a arquitetura de software escolhida foi a de rotas, tanto no cliente quanto no servidor. Não fizemos interface gráfica, tratamentos para segurança nem multithreading. Como resultado, conseguimos implementar toda a nossa própria especificação de forma bastante concisa, e aprendemos bastante sobre o empoderamento que a API de Sockets oferece. Além disso, nosso trabalho é facilmente reproduzível, bastando que se instale o interpretador Python 3 para executá-lo, visto que as implementações da API de Sockets e do SQLite são nativas e automaticamente instaladas junto com a máquina virtual da linguagem.

### 1. Informações importantes

Nossos objetivos com este trabalho são aprender o máximo possível do conteúdo da disciplina de forma prática - ou seja, entender o funcionamento da API de Sockets, como utilizá-la e principalmente quando utilizá-la - e também conseguir uma boa nota. Como interface gráfica, tratamentos para segurança e multithreading não serão avaliados, optamos por não fazê-los. Embora pareça uma decisão evasiva e até preguiçosa de nossa parte, isso nos permitiu um foco completo em realizar a troca de mensagens via API de Socket, além de nos poupar tempo que foi gasto também no conteúdo de outras disciplinas. Assim, toda a documentação também seguirá este foco, deixando de lado alguns detalhes sobre como lidar com a linguagem Python, SQL, etc para detalhar e explicitar as decisões sobre a API de Socket, porquê as tomamos e porquê acreditamos ter sido boas escolhas.



```
mteus@DESKTOP-54E0GTO: /mnt/c/Users/mateu/Desktop/rock-album-v1$ python server/main.py
2021-09-17 19:56:09.293214 :: [INFO] Servidor online;
2021-09-17 19:56:47.175547 :: [INFO] Conectado a 127.0.0.1:36390;
2021-09-17 19:56:47.176181 :: [DEBUG] Recebido JSON de Requisição = {'method': 'user/login', 'email': 'administrador@ufv.br', 'password': 'administrador'};
2021-09-17 19:56:47.183571 :: [DEBUG] Enviando JSON de Resposta = {'error': 0};
2021-09-17 19:56:47.184089 :: [INFO] Conectado a 127.0.0.1:36392;
2021-09-17 19:56:47.184178 :: [DEBUG] Recebido JSON de Requisição = {'method': 'user/is_admin', 'email': 'administrador@ufv.br'};
2021-09-17 19:56:47.186349 :: [DEBUG] Enviando JSON de Resposta = {'error': 0};
```

Figura 1. Processo servidor sendo executado com a efetuação do login de um administrador.

```
mateus@DESKTOP-54EOGTO: /mnt/c/Users/mateu/Desktop/rock-album-v1$ python client/main.py
Você deseja criar cadastro? [N/s]
Digite o seu email: administrador@ufv.br
Digite a sua senha: administrador

---- Menuzinho ----
0 - [Administrador] Criar novo cartão presente;
1 - [Administrador] Criar nova figurinha;
2 - [Administrador] Sortear prêmio da sorte;
3 - [Administrador] Tornar um jogador um administrador;
4 - [Administrador] Remover um jogador de administrador;
5 - [Álbum] Ver seu próprio álbum;
6 - [Álbum] Ver suas figurinhas livres (que não estão coladas nem a venda);
7 - [Álbum] Colar uma figurinha;
8 - [Mercado da comunidade] Comprar uma figurinha;
9 - [Mercado da comunidade] Ver preço de uma figurinha;
10 - [Mercado da comunidade] Ver suas figurinhas à venda;
11 - [Mercado da comunidade] Colocar uma figurinha à venda;
12 - [Usuário] Ver suas moedas;
13 - [Usuário] Resgatar cartão-presente;
14 - [Mercado oficial] Comprar pacote de figurinhas;
15 - Sair;

Digite uma opção válida: █
```

Figura 2. Processo cliente sendo executado com a efetuação do login de um administrador.

## 2. Como executar

Para executar os processos do sistema distribuído, é necessário ter o interpretador Python 3. É possível executá-los em quaisquer sistemas operacionais que tenham implementações do interpretador citado e a API de Sockets. O trabalho foi testado no Windows 10 e no Linux Ubuntu. Além disso, é preciso que a porta 50007 esteja aberta no Firewall do sistema operacional. Caso não seja possível abri-la por algum motivo, é possível trocá-la no arquivo *config.json* que está na raiz do projeto.

Com os pré-requisitos cumpridos, basta executar o comando **python server/main.py** para executar o processo servidor e **python client/main.py** para o processo cliente. Note que serão necessárias duas instâncias de terminal para isso. Embora seja possível registrar no sistema pelo próprio cliente, criamos os seguintes usuários para facilitar a avaliação da professora/monitor:

Email	Senha	É administrador?
administrador@ufv.br	administrador	Sim
a@ufv.br	a	Não
b@ufv.br	b	Não

## 3. Decisões de implementação

Todo o nosso trabalho foi feito da forma mais simples possível, e nos inspiramos bastante nas APIs Rest e Frameworks Front-end modernos para projetá-lo e desenvolvê-lo. Sobre as opções tecnológicas, optamos por fazer o trabalho na linguagem Python 3 por contar com suporte a API de Socket nativa, bem documentada e de fácil acesso. Além

disso, por ser uma linguagem pouco verbosa, ela nos permitiria codificar rapidamente o trabalho e efetivar mudanças necessárias. Outro fator importante foi o seu suporte nativo tanto a JSONs, utilizados na troca de mensagens, quanto ao banco de dados SQLite 3, utilizado para a persistência. Já a escolha do formato JSON (acrônimo para Notação de Objeto Javascript) foi bastante natural e simples, visto que é o formato utilizado em APIs Rest, e se tornou o formato padrão da internet por permitir enviar mensagens na forma chave-valor e legíveis para humanos. Já o banco de dados SQLite 3 foi escolhido por ser, no nosso entendimento, a forma mais simples para manter uma persistência básica em algum software (ele é muito utilizado em aplicativos de Android para essa finalidade, por exemplo).

Sobre as decisões arquiteturais, optamos por utilizar a arquitetura de rotas tanto para o processo servidor quanto para o processo cliente. Este modelo, que já é adotado para processo servidor há um bom tempo, vem sendo recentemente utilizado também para clientes a fim de manter apenas um padrão de projetos para todo um sistema distribuído. Como Python é uma linguagem de programação multi-paradigma, uma má especificação de arquitetura pode levar facilmente a um software ruim e confuso. Assim, considerando nossas já citadas inspirações, optamos por utilizar apenas programação funcional (pelo menos até onde Python permite isso), evitando ao máximo efeitos colaterais que não fossem, é claro, do próprio banco de dados. Assim, há pouquíssimas variáveis no código e muitas constantes.

```
1 def login():
2     while True:
3         REQUEST = {
4             'method': 'user/login',
5             'email': input('Digite o seu email: '),
6             'password': input('Digite a sua senha: ')
7         }
8
9         RESPONSE = contact_server(REQUEST)
10
11         if RESPONSE['error'] == 1:
12             print(f'Erro! {RESPONSE["error_message"]} Tente novamente
13             ...\\n')
14         else:
15             return REQUEST['email']
```

**Fragmento de código 1. Função de login do processo cliente. Perceba que o Python recomenda que constantes sejam escritas com letras maiúsculas.**

## 4. O banco de dados

Como já dito, utilizamos o banco de dados SQLite 3 por ser simples e fácil de reproduzir. Isto é, não é necessário que um software separado de banco de dados seja instalado, como acontece no MySQL ou PostgreSQL. Além disso, optamos por fazer todas as consultas sem auxílio de nenhuma biblioteca ou Framework, visto a simplicidade do sistema distribuído e o maior controle ao banco que essa decisão trás consigo. O banco de dados conta com apenas três tabelas criadas por nós: a **giftcards**, **stickers** e **users**. Além delas, há uma quarta, a **sqlite\_sequence**, que é gerada e mantida automaticamente pelo SQLite 3 e tem finalidade de cuidar das colunas com *AUTO INCREMENT*. Como ela é bastante

focada no software do banco e não no nosso projeto, optamos por falar apenas das três tabelas criadas por nós.

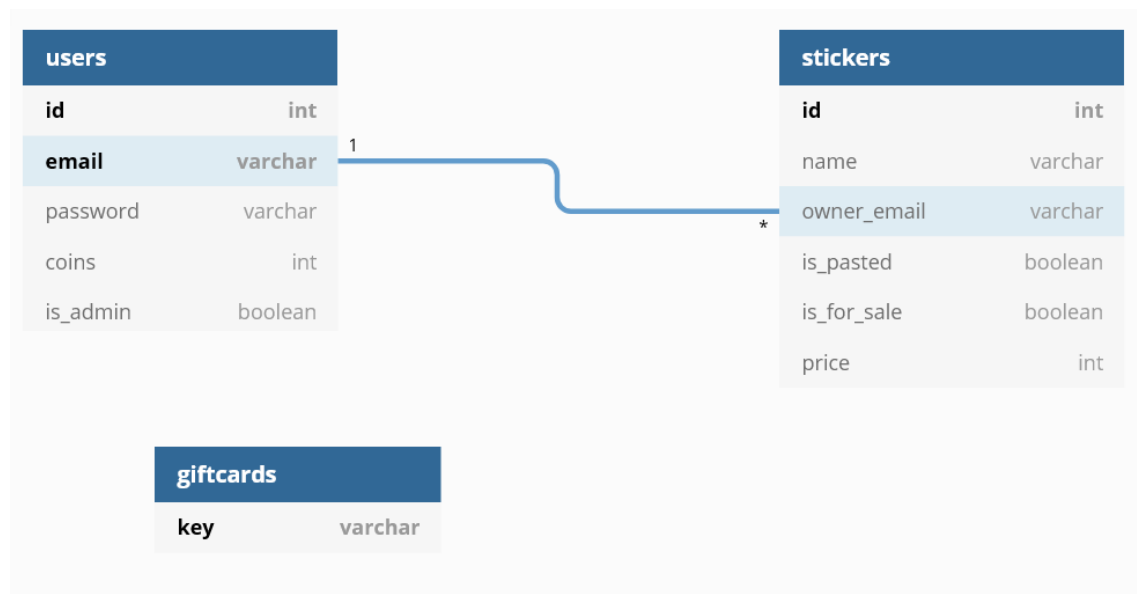


Figura 3. Diagrama entidade-relacionamento do banco de dados.

A tabela **giftcards** gerencia os cartões-presente, e ela contém apenas uma coluna que é a chave do cartão. Quando um cartão é utilizado, sua linha é removida. A tabela **stickers** contém as figurinhas de todos os usuários. Ela contém um identificador, o nome da figurinha, o email do usuário dono (caso ela seja de alguém), booleanos para indicar se ela está colada ao álbum do dono e se está a venda e o seu preço (caso esteja a venda). A tabela **users** persiste os usuários, e contém colunas para o email do usuário, sua senha, o número de moedas e um booleano que indica se o usuário é administrador.

## 5. O processo servidor

O processo servidor do nosso sistema distribuído funciona, em alto nível, o processo servidor recebe requisições em JSON, faz uma operação com banco de dados, e retorna um JSON de resposta para o cliente. Entretanto, para que a comunicação funcione via API de Socket, tais JSON são enviados e recebidos serializados e codificados via UTF-8, e a conexão deve ser iniciada e encerrada a cada iteração do laço. Todas essas operações são feitas em um laço indeterminado, que só para através de interrupção do processo. Assim, o processo servidor pode ser visto como um grande receptor e cuspidor de JSONs. Esse forma de se implementar um processo servidor é o utilizado via API Rests e foi adotado por nós por ser bastante simples e eficiente.

```

2021-09-17 18:28:38.667172 :: [DEBUG] Enviando JSON de Resposta = {'error': 0};
2021-09-17 18:28:43.693338 :: [INFO] Conectado a 127.0.0.1:55144;
2021-09-17 18:28:43.693338 :: [DEBUG] Recebido JSON de Requisição = {'method': 'admin/create_giftcard', 'email': 'administrador@ufv.br', 'giftcard_key': 'cartao2'};
2021-09-17 18:28:43.704941 :: [DEBUG] Enviando JSON de Resposta = {'error': 0};
2021-09-17 18:28:46.912116 :: [INFO] Conectado a 127.0.0.1:55146;
2021-09-17 18:28:46.912116 :: [DEBUG] Recebido JSON de Requisição = {'method': 'admin/create_giftcard', 'email': 'administrador@ufv.br', 'giftcard_key': 'cartao3'};
2021-09-17 18:28:46.924482 :: [DEBUG] Enviando JSON de Resposta = {'error': 0};
  
```

Figura 4. Processo servidor em execução. Note que implementamos diversas mensagens de debugging.

## 5.1. A API de Socket

Tentamos ao máximo modularizar a implementação da API de Socket, a fim de facilitar o nosso trabalho e a adaptação do código no próximo trabalho. Para isso, fizemos uma implementação dessa API que apenas envia e recebe JSONs.

```
1 with sqlite3.connect(CFG['SQLITE_FILE']) as DB, socket.socket(socket.
    AF_INET, socket.SOCK_STREAM) as SCK:
2     SCK.bind((CFG['HOST'], CFG['PORT']))
3     SCK.listen(CFG['LISTEN'])
4     print(f'{datetime.now()} :: [INFO] Servidor online;')
5
6     while True:
7         CONNECTION, ADDRESS = SCK.accept()
8         print(f'{datetime.now()} :: [INFO] Conectado a {ADDRESS[0]}:{
    ADDRESS[1]};')
9
10        REQUEST = json.loads(CONNECTION.recv(CFG['BUFSIZE']).decode(
    CFG['ENCODE_FORMAT']))
11        print(f'{datetime.now()} :: [DEBUG] Recebido JSON de Requisicao
    = {REQUEST};')
12
13        RESPONSE = router(REQUEST, DB)
14
15        CONNECTION.sendall(json.dumps(RESPONSE).encode(CFG['
    ENCODE_FORMAT']))
16        print(f'{datetime.now()} :: [DEBUG] Enviando JSON de Resposta =
    {RESPONSE};')
```

**Fragmento de código 2. Código do servidor que envia e recebe JSONs**

Como é possível observar, o código acima começa fazendo uma conexão ao banco de dados e à API de Socket através do sistema operacional, escolhe o IP e a porta, escuta a conexão e inicia o laço de repetição para enviar e receber cada um dos JSONs. O laço de repetição é indeterminado, e inicia aceitando uma conexão, carregando uma requisição que é decodificada e desserializada em JSON. Depois, é chamado a função roteador (explicada na próxima seção) que trata a requisição e gera uma resposta em JSON. Ao fim, esse JSON é serializado, codificado e reenviado ao cliente.

## 5.2. As rotas

Com os JSONs sendo enviados e recebidos, o único passo que falta é de fato tratar a requisição e gerar uma resposta coerente. Para isso, optamos por utilizar uma pequena função que escolhe a rota a ser utilizada baseado na chave **method** do próprio JSON.

```
1 def router(REQUEST, DATABASE):
2     MW = {
3         'admin/create_giftcard': routes.admin__create_giftcard,
4         'admin/create_stickers': routes.admin__create_stickers,
5         'admin/draw_lucky_prize': routes.admin__draw_lucky_prize,
6         'admin/op': routes.admin__op,
7         'admin/unop': routes.admin__unop,
8
9         'album/get_album': routes.album__get_album,
10        'album/get_free_stickers': routes.album__get_free_stickers,
11        'album/paste_sticker': routes.album__paste_sticker,
```

```

12         'community_market/buy_sticker': routes.
13         community_market__buy_sticker,
14         'community_market/get_sticker_price': routes.
15         community_market__get_sticker_price,
16         'community_market/get_stickers_waiting_for_sale': routes.
17         community_market__get_stickers_waiting_for_sale,
18         'community_market/put_sticker_to_sell': routes.
19         community_market__put_sticker_to_sell,
20
21         'user/get_coins': routes.user__get_coins,
22         'user/is_admin': routes.user__is_admin,
23         'user/login': routes.user__login,
24         'user/register': routes.user__register,
25         'user/retrieve_giftcard': routes.user__retrieve_giftcard,
26
27         'official_market/buy_sticker_pack': routes.
28         official_market__buy_sticker_pack,
29     }
30
31     return MW.get(REQUEST['method'], lambda *_: {'error': 1, '
error_message': 'Metodo nao encontrado!'})(REQUEST, DATABASE)

```

**Fragmento de código 3. O roteador que escolhe a rota certa a ser chamado do processo servidor.**

Cada uma das rotas chamadas faz uma requisição SQL, as trata da forma coerente e retorna um JSON padronizado que sempre possui uma chave **error**, que caso seja zero, o método não retornou erro, e caso seja um, ele retornou um erro. Caso esse método tenha retornado erro, ele ainda possui um **error\_message**, que é um string explicando o erro de forma amigável. Por exemplo, a função de login de usuário faz uma consulta procurando se há um usuário com o email e a senha passados pelo cliente. Caso exista, não é retornado erro. Caso não exista, é retornado um erro e uma mensagem. Todos os outros métodos das rotas seguem esse mesmo padrão, podendo retornar também outras informação a mais, mas nunca informações a menos.

```

1 def user__login(REQUEST, DATABASE):
2     DB_CUR = DATABASE.cursor()
3     if list(DB_CUR.execute(f'SELECT COUNT(*) FROM users WHERE email="{
REQUEST["email"]}" AND password="{REQUEST["password"]}"'))[0][0] ==
4         1:
5         return {'error': 0}
6     else:
7         return {'error': 1, 'error_message': 'Usuario nao encontrado!'}

```

**Fragmento de código 4. Rota de login do processo servidor.**

## 6. O processo cliente

O processo cliente do nosso sistema distribuído funciona, em alto nível, o processo cliente recebe informações do terminal, envia requisições em JSON, recebe respostas em JSON e as trata para exibir de forma amigável ao usuário. Da mesma forma que o processo servidor, tais JSONs são serializados e codificados com UTF-8. O processo cliente é bastante influenciado por Frameworks Front-end modernos, também usando rotas como o processo servidor.

```
Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
PS C:\Users\mateu\Desktop\rock-album-v1> python .\client\main.py
Você deseja criar cadastro? [N/s] n
Digite o seu email: a@ufv.br
Digite a sua senha: a

---- Menuzinho ----
0 - [Álbum] Ver seu próprio álbum;
1 - [Álbum] Ver suas figurinhas livres (que não estão coladas nem a venda);
2 - [Álbum] Colar uma figurinha;
3 - [Mercado da comunidade] Comprar uma figurinha;
4 - [Mercado da comunidade] Ver preço de uma figurinha;
5 - [Mercado da comunidade] Ver suas figurinhas à venda;
6 - [Mercado da comunidade] Colocar uma figurinha à venda;
7 - [Usuário] Ver suas moedas;
8 - [Usuário] Resgatar cartão-presente;
9 - [Mercado oficial] Comprar pacote de figurinhas;
10 - Sair;

Digite uma opção válida: █
```

Figura 5. Processo cliente em execução.

## 6.1. A API de Socket

Toda a API de Socket foi abstraída em uma função chamada **contact\_server(REQUEST)**. Ou seja, a cada par de envio e recebimento de JSONs, uma nova conexão via Socket é criada. Esse método funciona iniciando uma chamada a API de Socket no sistema operacional, conectando via IP e porta, envia o JSON serializado e codificado em UTF-8, e recebe e retorna o JSON desserializado e decodificado em UTF-8. Você pode conferir o código a seguir:

```
1 def contact_server(REQUEST):
2     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as SCK:
3         SCK.connect((CFG['HOST'], CFG['PORT']))
4         SCK.sendall(json.dumps(REQUEST).encode(CFG['ENCODE_FORMAT']))
5
6         return json.loads(SCK.recv(CFG['BUFSIZE']).decode(CFG['ENCODE_FORMAT']))
```

**Fragmento de código 5. Envia o JSON de requisição e recebe o JSON de resposta do processo cliente.**

## 6.2. As rotas

Com a possibilidade de enviar e receber JSONs, o único passo que falta é criar esses JSONs de envio e exibir de forma amigável os JSONs de resposta. Para isso, utilizamos uma função de roteamento que escolhe a rota certa para criar o JSON certo coletando os dados de forma amigável do usuário para criar a requisição e exibindo de forma amigável sua resposta. Além disso, a função cria um menu bonito, porém resolvemos não mostrar isso na documentação por não ser o foco.

```
1 def router(EMAIL, IS_ADMIN):
2     ADMIN = [
```



```

3      ('[Administrador] Criar novo cartao presente', routes.
admin__create_giftcard),
4      ('[Administrador] Criar nova figurinha', routes.
admin__create_stickers),
5      ('[Administrador] Sortear premio da sorte', routes.
admin__draw_lucky_prize),
6      ('[Administrador] Tornar um jogador um administrador', routes.
admin__op),
7      ('[Administrador] Remover um jogador de administrador', routes.
admin__unop),
8      ]
9
10     NON_ADMIN = [
11         ('[Album] Ver seu proprio album', routes.album__get_album),
12         ('[Album] Ver suas figurinhas livres (que nao estao coladas nem
a venda)', routes.album__get_free_stickers),
13         ('[Album] Colar uma figurinha', routes.album__paste_sticker),
14         ('[Mercado da comunidade] Comprar uma figurinha', routes.
community_market__buy_sticker),
15         ('[Mercado da comunidade] Ver preco de uma figurinha', routes.
community_market__get_sticker_price),
16         ('[Mercado da comunidade] Ver suas figurinhas a venda', routes.
community_market__get_stickers_waiting_for_sale),
17         ('[Mercado da comunidade] Colocar uma figurinha a venda',
routes.community_market__put_sticker_to_sell),
18         ('[Usuario] Ver suas moedas', routes.user__get_coins),
19         ('[Usuario] Resgatar cartao-presente', routes.
user__retrieve_giftcard),
20         ('[Mercado oficial] Comprar pacote de figurinhas', routes.
official_market__buy_sticker_pack),
21         ('Sair', lambda *_: quit())
22     ]
23
24     ROUTES = ADMIN + NON_ADMIN if IS_ADMIN else NON_ADMIN

```

**Fragmento de código 6. O roteador que escolhe o método certo a ser chamado do processo cliente.**

Todas as rotas do processo cliente seguem o mesmo padrão, necessitado do email do usuário para ser enviado juntamente com a requisição e não retornam nada. Seu funcionamento interno também é padronizado. Primeiro, é criada a requisição coletando os dados do usuário, envia essa requisição para o servidor através da abstração da API de Socket explicada na seção anterior, e exibe a resposta de forma amigável.

```

1 def admin__op(EMAIL):
2     REQUEST = {
3         'method': 'admin/op',
4         'email': EMAIL,
5         'target_email': input('Insira o email a ser tornado
administrador: '),
6     }
7
8     RESPONSE = aux.contact_server(REQUEST)
9
10    if RESPONSE['error'] == 1:
11        print(f'Erro! {RESPONSE["error\message"]}')

```



```
12     else:
13         print('Feito!')
```

**Fragmento de código 7. Método de tornar um usuário administrador do processo cliente.**

## **7. Considerações finais**

Através deste trabalho, pudemos perceber o empoderamento que a API de Socket oferece. Porém, como tudo na computação de baixo nível, é necessário bastante conhecimento sobre o funcionamento da API e a implementação tende a ser verbosa. Assim, para sistemas que necessitam de maior performance ou flexibilidade, tal API tende a ser uma escolha muito boa.

Com relação a especificação, pudemos implementar toda a nossa própria especificação do Rock Album vol 1 sem maiores problemas. Criamos um sistema distribuído de processos portáteis e reproduzíveis, capaz de ser testado em diferentes sistemas operacionais e sem muitos pré-requisitos.