# Sistemas Distribuídos - TP3 - Rock Album vol. 1 - Implementação utilizando Middleware RMI

**Leandro Lázaro Araújo Vieira - 3513[1], Mateus Pinto da Silva - 3489[2]**

[1]Ciência da Computação – Universidade Federal de Viçosa (UFV-caf) – Florestal – MG – Brasil

(leandro.lazaro[1], mateus.p.silva[2])@ufv.br

***Resumo.*** *Implementamos a nossa especificação de sistema distribuído de álbum de figurinhas, o Rock Album vol. 1, utilizando Middleware RMI. Optamos por fazer isso da forma mais simplificada o possível, utilizando a linguagem de programação Python, o banco de dados SQLite3 e o framework Pyro5 de Middleware RMI. Muito código do trabalho prático de implementação via Socket foi reaproveitado. 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 as utiliadades que os Middlewares RMI oferecem. Também descrevemos rapidamente as diferenças percebidas entre a implementação via Sockets e via Middleware RMI. Além disso, nosso trabalho é facilmente reproduzível, bastando que se instale o interpretador Python 3 e um ambiente Virtualenv para executá-lo, visto que o framework Pyro5 não é nativo da linguagem, embora o banco de dados SQLite3 seja.*

## 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 dos Middlewares RMI, como utilizá-los e principalmente quando utilizá-los - e também conseguir uma boa nota. Como interface gráfica, tratamentos para segurança, tratamento de erros de comunicação (como queda do servidor, queda de rede) 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 as invocações remotas via o Middleware RMI Pyro5, 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. Além disso, assume-se que o leitor desta documentação já tenha lido a do trabalho prático anterior feito via Sockets (cujo título é "Sistemas Distribuídos - TP3 - Rock Album vol. 1 - Implementação utilizando socket"), portanto partes reaproveitadas de código não serão reexplicadas.

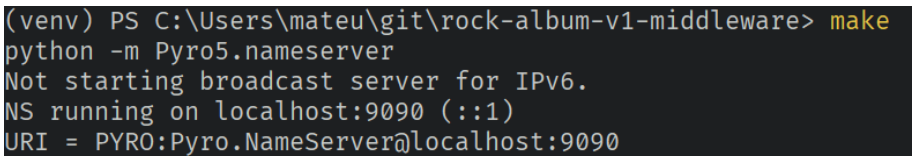


**Figura 1. Vinculador sendo executado (infelizmente ele não possui modo debug).**


**Figura 2. Processo servidor sendo executado, encontrando o vinculador (servidor de nomes) e iniciando seu laço indeterminado que espera clientes.**

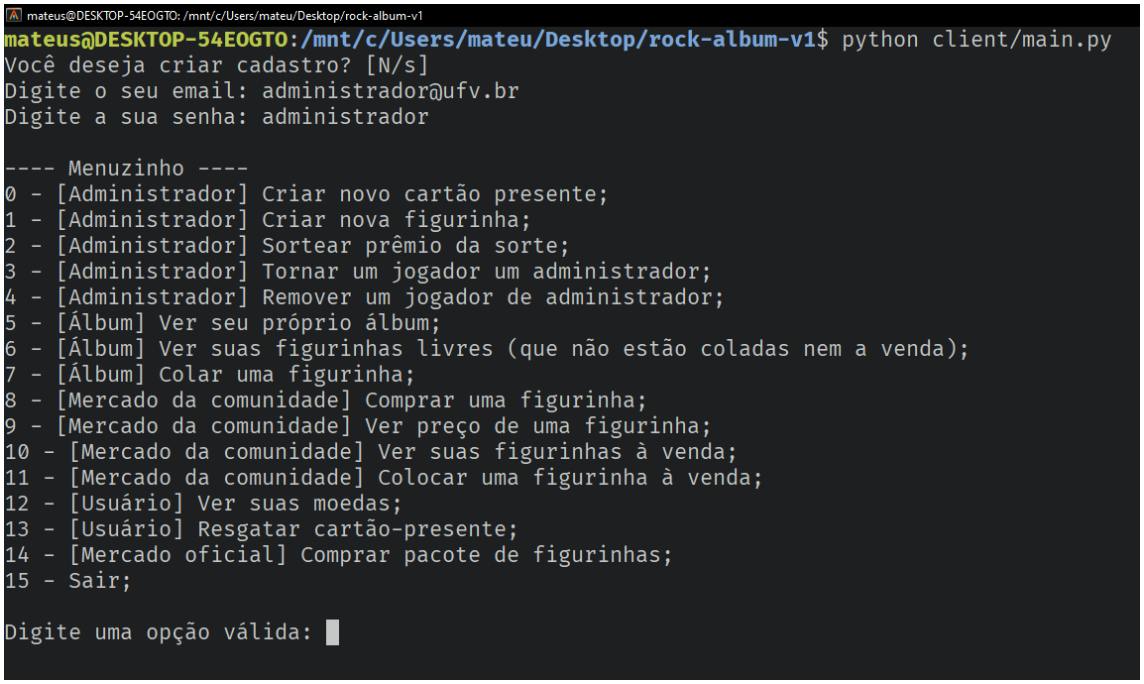


**Figura 3. Processo cliente sendo executado com a efetuação do login de um administrador.**

## 2. Como executar

Para executar os processos do sistema distribuído, é necessário ter o interpretador Python 3 e o gerenciador de ambientes Python 3 Virtualenv. 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 55525 e 9090 estejam aberta no Firewall do sistema operacional.

Antes da execução, é necessário criar um ambiente virtual Python 3 Virtualenv. Para isso, execute o comando **virtualenv venv**. Depois disso, utilize o comando **pip install -r requirements.txt** para instalar o Pyro5 nesse ambiente. Para usar o ambiente virtual no Linux, utilize o comando **source venv/bin/activate** e no Windows, utilize o comando **.\venv\Scripts\activate.ps1** no PowerShell para fazer o mesmo. Para a correta execução e teste do sistema distribuídos, três instâncias do terminal com o ambiente virtual ativado são necessárias, então o último comando (seja em sua versão Linux ou Windows) precisa ser repetido três vezes, uma em cada janela de terminal.

Com os pré-requisitos, finalmente será possível executar o sistema distribuído. Como trata-se de um sistema implementado via Middleware RMI, um vinculador (servidor de nomes) é necessário. Para executá-lo, digite em um dos terminais criados acima o comando **make**. O processo servidor pode ser iniciado em outro terminal criado acima com o comando **make run_server**. O processo cliente pode ser iniciado no último terminal criado acima disponível com o comando **make run_client**. 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 |

A versão do banco de dados é exatamente igual a do último trabalho prático, e toda a interface do cliente também é idêntica, a fim de poupar-nos retrabalho e ajudar na correção.

## 3. Decisões de implementação

Como nosso código do trabalho anterior foi feito com programação funcional, e os Middlewares RMI requerem códigos de programação orientada a objetos no processo servidor, uma conversão foi necessária. Assim, seguindo a literatura, a opção de classe Singleton foi adotada. O processo cliente, por sua vez, precisou de ainda menos adaptação, bastando instanciar um objeto servidor e utilizar os métodos dele.

```
def login(SERVER):
    while True:
        EMAIL = input('Digite o seu email: ')
        PASSWORD = input('Digite a sua senha: ')

        RESPONSE = SERVER.user__login(EMAIL, PASSWORD)

        if RESPONSE['error'] == 1:
            print(f'Erro! {RESPONSE["error_message"]} Tente novamente
    ...\n')
        else:
            return EMAIL
```

**Fragmento de código 1. Função de login do processo cliente adaptada, que recebe um objeto servidor e chama seu método de login.**

## 4. O banco de dados

Não houve nenhuma alteração no banco de dados, nem quanto a suas tabelas nem ao driver utilizado. Assim, todo o modelo, os motivos das decisões de implementação são iguais as do trabalho anterior.

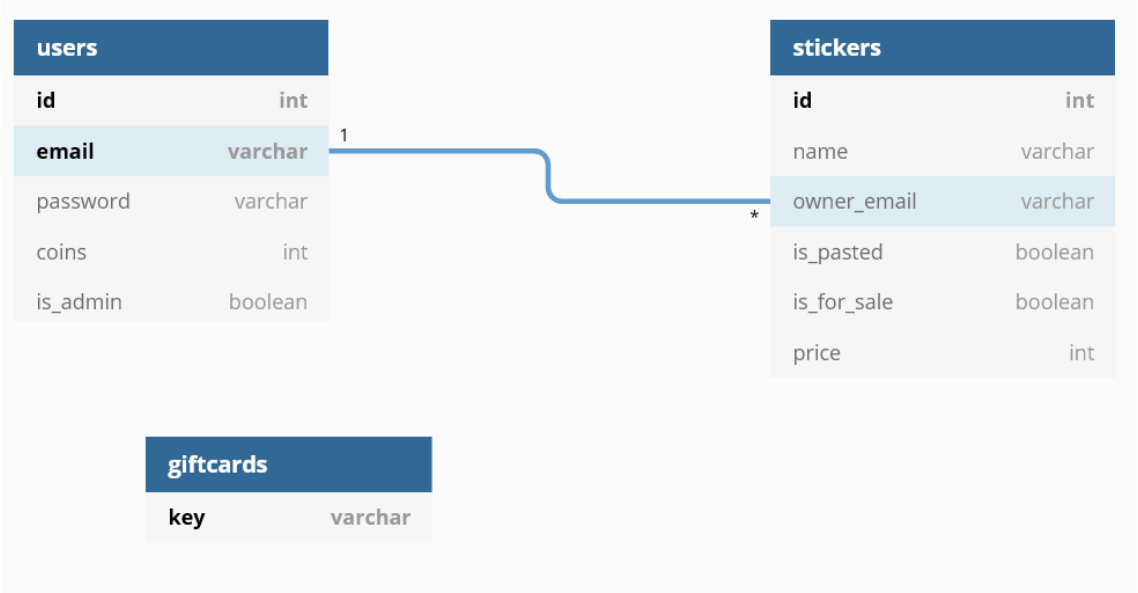


**Figura 4. Diagrama entidade-relacionamento do banco de dados.**

## 5. O vinculador (servidor de nomes)

Utilizamos o vinculador (servidor de nomes) padrão do Pyro5, pois ele foi suficiente e funcionou muito bem em todos os testes. Assim, nenhuma configuração precisou ser alterada nele, e não foi necessário codificar nada para que ele funcionasse, bastando chamá-lo via terminal. Infelizmente ele não contém modo debugging, que seria útil para visualizar mais informações de seu funcionamento interno.

## 6. O processo servidor

O processo servidor do nosso sistema distribuído funciona, em alto nível, como uma classe Singleton que implementa todas as rotas do processo servidor do trabalho anterior como métodos. As rotas principais tornar-se-aram métodos públicos. Por sua vez, rotas auxiliares viraram métodos privados (ou seja, iniciados por "__" no Python). Embora a resposta - isto é, o retorno dos métodos - continue retornando JSONs, a fim de encapsular os erros e evitar que exceções sejam disparadas (e evitar a necessidade de tratá-las com estrutura de try/except no cliente, além de evitar toda a dificuldade de criar classes para representar exceções específicas), a requisição - os parâmetros formais dos métodos - foram modificados. Ao invés dos JSONs, optamos por utilizar tipos primitivos da própria linguagem Python para fins de aumento da legibilidade do código. Por fim, a operação de conexão ao banco de dados foi feita no construtor da classe servidor e a operação de encerramento da conexão ao banco de dados foi feita no destrutor. Assim, quando o primeiro cliente se conecta, o banco de dados é acionado e a conexão é estabelecida. Caso o objeto servidor expire (ou seja, quando nenhuma nova invocação a método do objeto servidor seja feita em um tempo hábil), o objeto será expirado, destruído e a conexão ao banco será encerrada. Caso, depois disso, outro cliente deseje invocar algum método, novamente o construtor será acionado e uma nova conexão ao banco será estabelecida.

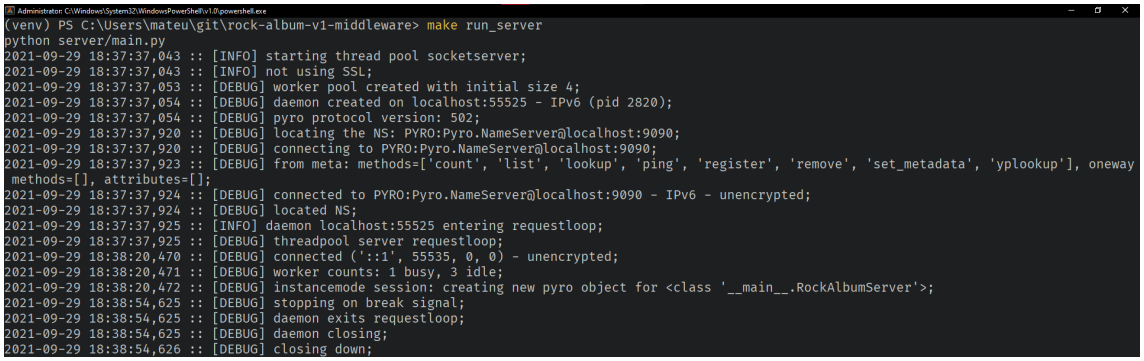


**Figura 5. Processo servidor em execução. Note que usamos o debugging do próprio Pyro5.**

```python
@Pyro5.api.expose
class RockAlbumServer(object):
    def __init__(self):
        CFG = json.load(open('config.json'))
        self.DATABASE = sqlite3.connect(CFG['SQLITE_FILE'])

    def __del__(self):
        self.DATABASE.close()
```

**Fragmento de código 2. Classe do servidor exposta pelo Middleware RMI Pyro5, com a declaração de seu construtor e destrutor.**

## 6.1. O Middleware RMI

Todo o processo servidor foi escrito em apenas um arquivo, pois ele ficou muito simples e dividi-lo apenas pioraria a legibilidade do código. Nele há apenas a definição da classe Singleton do servidor e sua inicialização. Sobre a inicialização, ela começa definindo algumas diretivas de debugging, procurando o endereço do vinculador (servidor de nomes), registrando a classe do servidor e iniciando o laço que aguarda novos clientes. Além disso, foi necessário definir um nome lógico para o objeto. Para isso, seguimos à risca a recomendação da documentação do próprio Pyro5, e o nome adotado foi **rockalbum.server**.

```python
if __name__ == "__main__":
    logging.basicConfig(stream=sys.stdout, level=logging.DEBUG, format='%(asctime)s :: [%(levelname)s] %(message)s;')
    logging.getLogger("Pyro5").setLevel(logging.DEBUG)
    logging.getLogger("Pyro5.core").setLevel(logging.DEBUG)

    with Pyro5.server.Daemon() as DAEMON:
        NAME_SERVER = Pyro5.api.locate_ns()
        URI = DAEMON.register(RockAlbumServer)
        NAME_SERVER.register("rockalbum.server", URI)

        DAEMON.requestLoop()
```

**Fragmento de código 3. Inicialização do processo servidor, com definição das mesagens de debugging e criação do laço para aguardar clientes.**

### 6.2. As rotas

Como já dito, todas as rotas foram implementadas via métodos da própria classe Singleton do servidor. Assim, as rotas auxiliares são os métodos privados e podem retornar diferentes tipos, enquanto as rotas públicas são métodos públicos que retornam JSONs. Rotas auxiliares são úteis para fazer verificações e aumentar a redigibilidade e reuso de código, enquanto as rotas públicas são de fato as chamadas pelos clientes. Além disso, optamos por usar notação de tipos nos parâmetros formais de todos os métodos para aumentar a legibilidade do código.

```
def __is_this_sticker_waiting_for_sale(self, STICKER_ID: int) -> int:
    DB_CUR = self.DATABASE.cursor()
    return 1 if list(DB_CUR.execute(f'SELECT COUNT(*) FROM stickers
    WHERE id="{STICKER_ID}" AND is_for_sale=1'))[0][0] == 1 else 0
```

**Fragmento de código 4. Exemplo de rota auxiliar implementada via método privado.**

```
def user__login(self, EMAIL: str, PASSWORD: str):
    DB_CUR = self.DATABASE.cursor()
    if list(DB_CUR.execute(f'SELECT COUNT(*) FROM users WHERE email="{
    EMAIL}" AND password="{PASSWORD}"'))[0][0] == 1:
        return {'error': 0}
    else:
        return {'error': 1, 'error_message': 'Usuario nao encontrado!'}
```

**Fragmento de código 5. Rota de login do processo servidor implementada via método público.**

## 7. O processo cliente

O processo cliente do nosso sistema distribuído funciona, em alto nível, como um aplicativo qualquer que utiliza o objeto remoto servidor. Conseguimos reaproveitar ainda mais código do trabalho anterior do que no processo servidor. O paradigma de programação continua sendo o funcional, e o único objeto utilizado foi o do servidor em si.

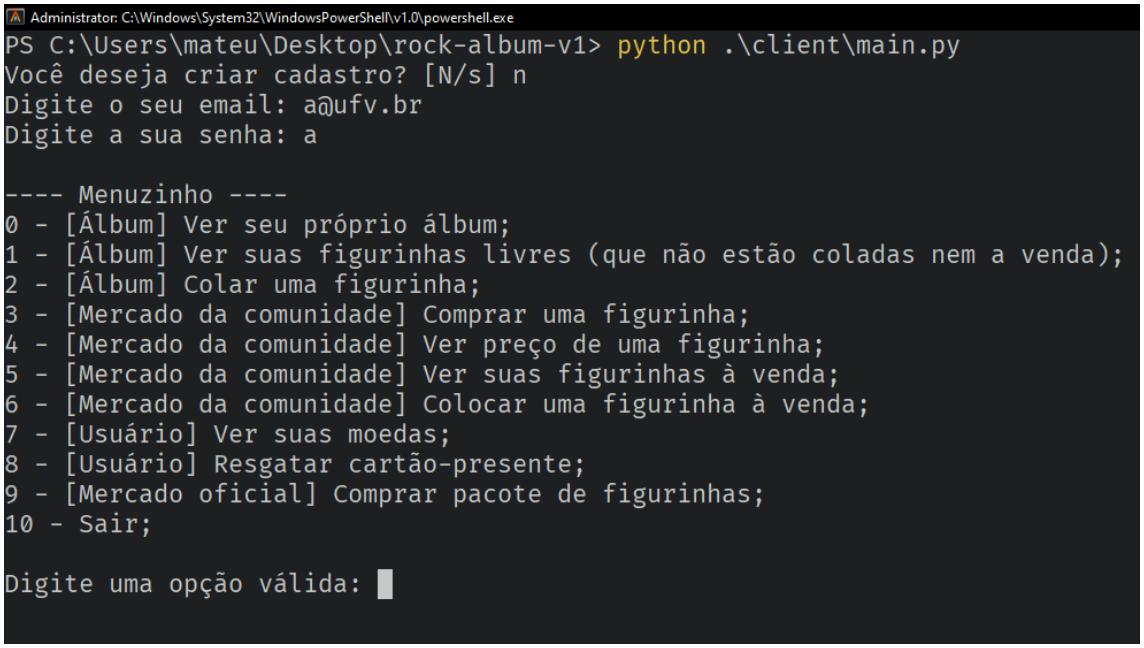


**Figura 6. Processo cliente em execução.**

### 7.1. O Middleware RMI

Middlewares RMI geralmente são feitos para que os clientes consigam utilizar objetos de forma bastante abstraída. Felizmente, o Pyro5 segue essa filosofia à risca. Assim, apenas foi necessário utilizar uma função da linguagem que retorna uma referência remota ao objeto servidor e chamar os métodos dele sem maiores problemas. Tal função necessita do nome lógico do objeto servidor. O vinculador (servidor de nomes) é procurado de forma implícita pelo framework. Por padrão, o endereço do vinculador (servidor de nomes) e do processo servidor no Pyro5 são **localhost**, por isso não o declaramos de forma explícita no código.

```
1 if __name__ == "__main__":
2     SERVER = Pyro5.api.Proxy("PYRONAME:rockalbum.server")
3     EMAIL = routes.register(SERVER) if input("Voce deseja criar
      cadastro? [N/s] ").lower() == 's' else routes.login(SERVER)
4     IS_ADMIN = routes.is_admin(SERVER, EMAIL)
5     print()
6     router(SERVER, EMAIL, IS_ADMIN)
```

**Fragmento de código 6. Fragmento do arquivo main.py do processo cliente. Note que a função Pyro5.api.Proxy é utilizada para recuperar o objeto servidor.**

### 7.2. As rotas

Com a possibilidade de enviar e receber informações do servidor via invocação de métodos remotos, o único passo que falta é coletar do usuário os dados para os parâmetros reais e exibir de forma amigável os retornos dos métodos. Isso foi realizado da mesma forma que no trabalho anterior, ou seja, via uma função de roteamento.

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

```
        ('[Administrador] Criar novo cartao presente', routes.admin__create_giftcard),
        ('[Administrador] Criar nova figurinha', routes.admin__create_stickers),
        ('[Administrador] Sortear premio da sorte', routes.admin__draw_lucky_prize),
        ('[Administrador] Tornar um jogador um administrador', routes.admin__op),
        ('[Administrador] Remover um jogador de administrador', routes.admin__unop),
    ]

    NON_ADMIN = [
        ('[Album] Ver seu proprio album', routes.album__get_album),
        ('[Album] Ver suas figurinhas livres (que nao estao coladas nem a venda)', routes.album__get_free_stickers),
        ('[Album] Colar uma figurinha', routes.album__paste_sticker),
        ('[Mercado da comunidade] Comprar uma figurinha', routes.community_market__buy_sticker),
        ('[Mercado da comunidade] Ver preco de uma figurinha', routes.community_market__get_sticker_price),
        ('[Mercado da comunidade] Ver suas figurinhas a venda', routes.community_market__get_stickers_waiting_for_sale),
        ('[Mercado da comunidade] Colocar uma figurinha a venda', routes.community_market__put_sticker_to_sell),
        ('[Usuario] Ver suas moedas', routes.user__get_coins),
        ('[Usuario] Resgatar cartao-presente', routes.user__retrieve_giftcard),
        ('[Mercado oficial] Comprar pacote de figurinhas', routes.official_market__buy_sticker_pack),
        ('Sair', lambda *_: quit())
    ]

    ROUTES = ADMIN + NON_ADMIN if IS_ADMIN else NON_ADMIN
```

**Fragmento de código 7. O roteador que escolhe o método certo a ser chamado do processo cliente.**

Todas as rotas do processo cliente precisam do objeto servidor para serem executadas. Além disso, pode ser que seja necessário o email do usuário. Nas execuções das rotas, primeiro os dados do usuário são coletados, o método correspondente a rota é invocado através do objeto servidor, e a resposta é exibida de forma amigável. Os métodos que não precisam do email do usuário tem, como o último de seus parâmetros formais, o identificador "_", para que todas as funções de rota contenham a mesma aridade, o que permite que elas possam ser trabalhadas como um mesmo tipo de função de primeira classe pela função roteador.

```
def admin__op(SERVER, _):
    TARGET_EMAIL = input('Insira o email a ser tornado administrador: ')

    RESPONSE = SERVER.admin__op(TARGET_EMAIL)

    if RESPONSE['error'] == 1:
        print(f'Erro! {RESPONSE["error_message"]}')
    else:
```

```
9         print('Feito!')
```

**Fragmento de código 8. Método de tornar um usuário administrador do processo cliente.**

## 8. Considerações finais

Através deste trabalho, pudemos perceber o empoderamento que os Middlewares RMI oferecem. Porém, eles apresentam alguns problemas. O principal deles, ao nosso ver, é a obrigatoriedade de que pelo menos uma parte do sistema seja orientado a objetos. No nosso caso, reaproveitamos o código da implementação anterior que é do paradigma funional. Felizmente, converter código do paradigma funcional para o orientado a objeto é bastante simples, e fizemos isso sem problemas. Além disso, caso o programador que trabalhará com algum código de processo cliente não saiba exatamente o que está fazendo, há o risco de que ele esqueça que está lidando com objetos remotos e não trate as exceções que isso pode causar - como falhas no processo servidor, máquina do servidor ou canal de comunicação - prejudicando a confiabilidade do projeto (lembrando que não tratamos tais exceções pois isso não será avaliado).

Além disso, o próprio framework Pyro5 contém problemas. Não é possível importar a classe servidor no cliente, por exemplo, para usar o auto completar ou o *linting* nos editores de código. Assim, a única forma de saber quais métodos estão disponíveis é via documentação do sistema. Isso pode levar a inúmeros problemas de engenharia de software, principalmente considerando que o Python é uma linguagem puramente interpretada e, portanto, não verifica erros em tempo de compilação. Além disso, foi perceptível o aumento do tempo de envio e recebimento das mensagens dessa implementação em comparação a anterior que usava API de Sockets.

Por isso, achamos muito mais simples a implementação via API de Sockets pois, embora existam as desvantagens comentadas no trabalho interior, ela é muito mais explícita. Através deste trabalho, avaliamos que os Middlewares RMI (seja o próprio Pyro5 ou outro framework) são boas soluções para sistemas distribuídos apenas em casos em que já exista um sistema legado local feito com orientação a objetos e deseja-se distribuí-lo. Em contextos em que desempenho é muito desejado ou que seja necessário cuidar de aspectos de baixo nível, a API de Sockets parece ser uma ideia melhor. Em outros casos, através de conhecimento prévio e estudos da disciplina, acreditamos que REST API seja a melhor opção.

Com relação a especificação do sistema distribuído em si, 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.