



UNIVERSIDADE FEDERAL DE VIÇOSA - UFV - CAMPUS FLORESTAL

SISTEMAS DISTRIBUÍDOS E PARALELOS

Trabalho Prático 1 - Parte 4

Implementação usando Middleware RMI

Emily Lopes Almeida - [Emily-Lopes](#)

Ingred Fonseca de Almeida - [ingredalmeida1](#)

Iury Martins Pereira - [iurymartins46](#)

Letícia Oliveira Silva - [leticiasilvaa](#)

Florestal - MG

2024

Sumário

1. Introdução	3
1.1. Como Executar	3
2. Desenvolvimento	4
2.1. Organização do Código	4
2.2. Decisões de Implementação	6
2.2.1. Modificando a comunicação	6
2.2.2. Mudanças no servidor de banco de dados	6
2.2.3. Mudanças no servidor de aplicação	7
3. Conclusão	10
4. Referências	10

1. Introdução

O sistema distribuído desenvolvido, denominado `EmojiCards`, consiste em um jogo de cartas destinado a envolver os usuários na exploração das emoções humanas. O objetivo é proporcionar aos jogadores a oportunidade de colecionar cartas, compor seus próprios baralhos e competir em um ambiente virtual colaborativo. Como incentivo adicional, os vencedores das partidas são recompensados com uma nova carta para expandir sua coleção.

O sistema foi inicialmente implementado usando `API de Sockets`, e nesta parte do trabalho foi implementado utilizando o middleware `Pyro5`, que é um middleware específico para a linguagem Python. Como o middleware é uma camada de software que adiciona abstração, nesta estratégia de implementação as entidades arquitetônicas são representadas por objetos e a comunicação é feita utilizando o paradigma de invocação remota. Além disso, toda comunicação entre processo foi feita por fluxo `TCP`, que é o protocolo utilizado pelo middleware.

1.1. Como Executar

Para facilitar a execução do sistema distribuído foi utilizado o `Docker`. Portanto, antes de executar é preciso ter o `Docker` configurado na máquina.

No Sistema Operacional `Linux`, primeiramente, é preciso executar o comando abaixo antes de executar os containers para liberar o acesso do display ao `Docker`:

```
$ xhost +local:*
```

Se for a primeira execução do sistema, é preciso usar o comando abaixo para preparar as imagens remotas:

```
$ docker compose build
```

Como o sistema utiliza `Pyro5` para gerenciar a comunicação entre os processos distribuídos, é preciso que o servidor `Pyro5` esteja em execução antes de subir os outros containers. Para isso, execute o comando a seguir para iniciar o `Pyro5`:

```
$ docker compose up -d rmi-ns
```

Feito isso, basta utilizar o comando abaixo para iniciar os demais containers descritos no ``docker-compose.yml``:

```
$ docker compose up
```

Com isso, o terminal `Docker` será iniciado e para encerrá-lo basta fazer `Ctrl + C`. Já para destruir os containers é preciso utilizar o comando:

```
$ docker compose down
```

O arquivo ``docker-compose.yml`` tem um contêiner para o servidor de banco de dados, um contêiner para o servidor de aplicação e três contêineres para três clientes, já que para jogar uma partida é preciso ter no mínimo três jogadores. Portanto, ao executar os comandos descritos, os dois servidores estarão disponíveis e três jogadores serão iniciados e por isso, três janelas com a interface gráfica serão abertas e a primeira tela exibida é a tela de login.

Também para facilitar, no banco de dados já existem três usuários cadastrados:

username	senha
thais	12345678
henrique	12345678
aluno	12345678

2. Desenvolvimento

2.1. Organização do Código

Com o objetivo de desenvolver um sistema robusto e modularizado, foi definido um modelo de arquitetura baseado em três camadas físicas e cinco camadas lógicas. As camadas físicas são distribuídas em apresentação, aplicação e dados, garantindo uma separação clara entre a interface do usuário, a lógica de negócios e o armazenamento de informações. As camadas lógicas, por sua vez, incluem: apresentação, negócio, comunicação, middleware e plataforma, sendo que, vale destacar que a camada lógica de apresentação, que corresponde à interface gráfica do usuário, está presente apenas no cliente, permitindo a interação direta do usuário com o sistema.

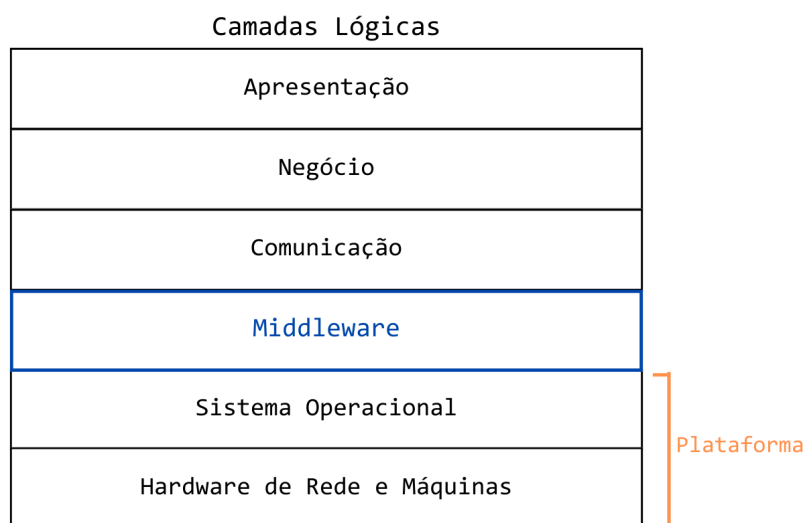


Figura 1 - Diagrama Representativo das Camadas Lógicas

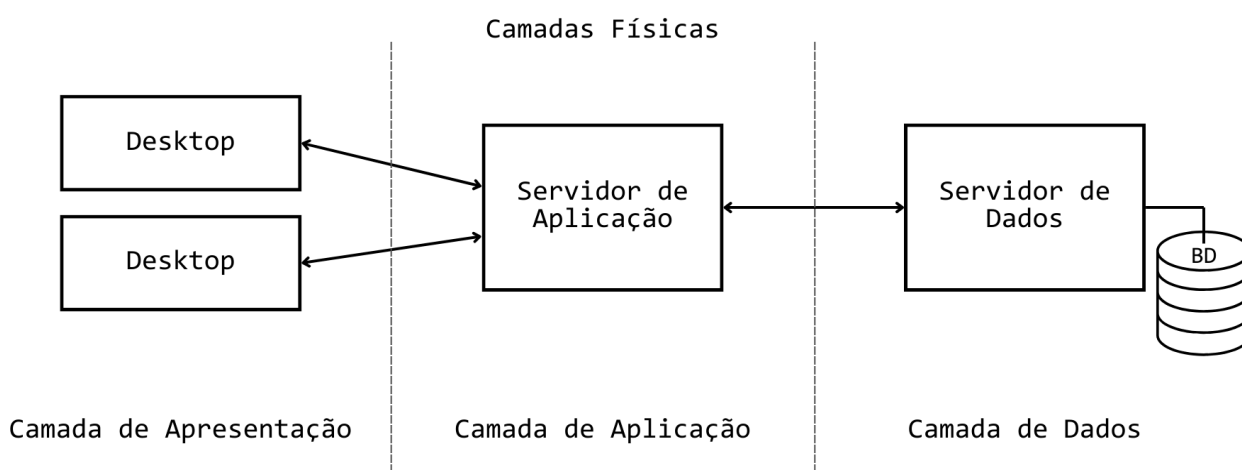


Figura 2 - Diagrama Representativo das Camadas Físicas

Nesse sentido, como mostra a Figura 3, o código foi organizado de forma a ter uma pasta representando o servidor de aplicação (/server-app), uma pasta representando o servidor de banco de dados (/server-bd), uma pasta representando um cliente (/cliente) e uma pasta para a definição do container Docker responsável pela a inicialização do middleware Pyro5 (/pyro).

Com relação às camadas lógicas, a camada de comunicação do servidor de aplicação, onde é definida a classe que pode ser instanciada como um objeto para disponibilizar os métodos do servidor de forma remota através do middleware, é representada pelo arquivo `server.py` e o arquivo `funcionalidades.py` representa a camada de negócio.

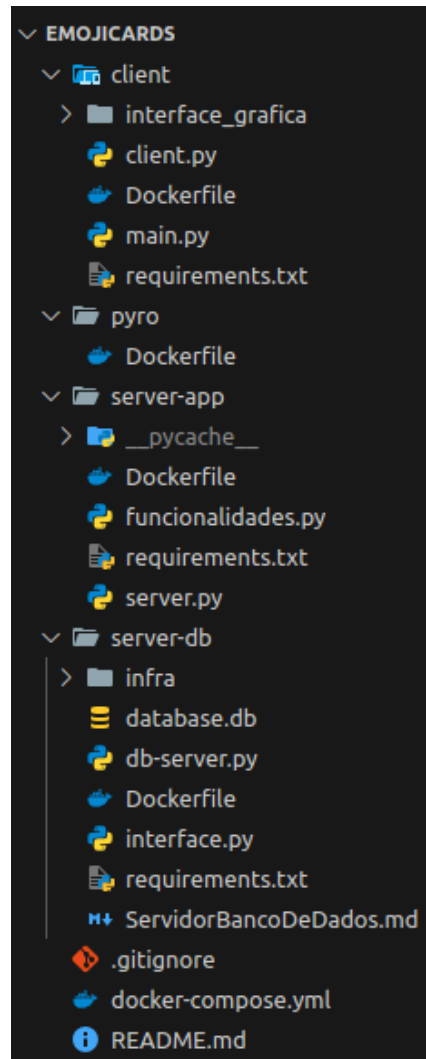


Figura 3 - Arquivos e Pastas do Projeto

No servidor de banco de dados, a camada de comunicação é representada pelo arquivo `db-server.py` e a camada de negócio é representada pelo arquivo `interface.py` e pela pasta `/infra`.

Já no cliente, a camada de comunicação e de negócio são representadas pelo arquivo `client.py`, onde são feitas invocações remotas aos métodos do servidor e a apresentação é representada pela pasta `/interface_gráfica`.

2.2. Decisões de Implementação

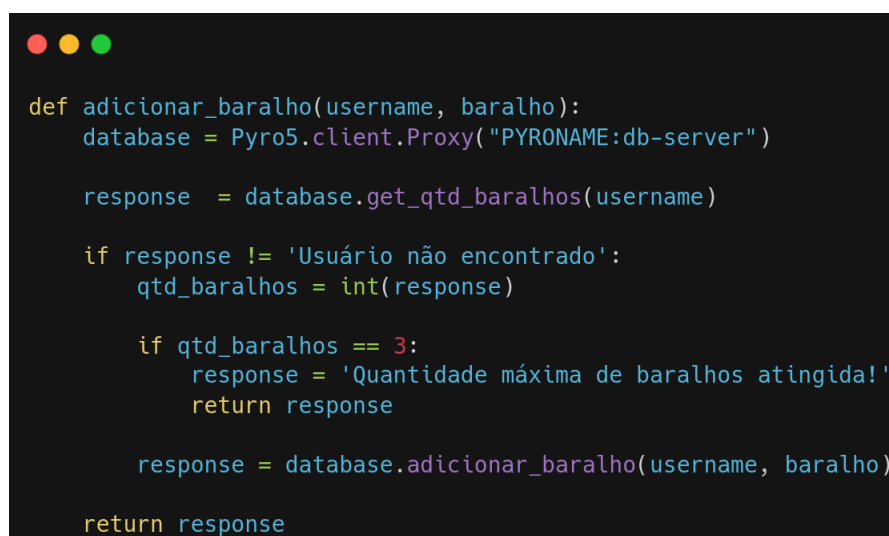
2.2.1. Modificando a comunicação

No trabalho anterior, a comunicação direta entre processos foi implementada utilizando API de `Sockets`. Para fazer uso do middleware, decidiu-se alterar apenas os arquivos referentes à comunicação entre os servidores e entre o servidor e o cliente, uma vez que o middleware abstrai toda a implementação com `Sockets` e lida com os desafios inerentes à comunicação. Dessa forma, cada arquivo responsável pelo gerenciamento do recebimento e envio de requisições nas camadas de banco de dados, servidor de aplicação e cliente foi modificado para utilizar o `Pyro5`, convertendo as entidades em objetos com métodos que podem ser invocados remotamente. Dessa forma, tanto a interface quanto às questões relacionadas ao negócio puderam ser mantidas da mesma forma que na implementação do trabalho anterior.

2.2.2. Mudanças no servidor de banco de dados

No servidor de banco de dados, foi necessário alterar apenas o arquivo responsável pelo gerenciamento do recebimento das requisições provenientes do servidor de aplicação. Nele foi definida a classe `Database` que é exposta usando o modificador `@Pyro5.server.expose`. Dessa forma, quando a classe é instanciada, a instância é registrada no `Name Server` do `Pyro` com o nome `db-server`, o que permite que os métodos da instância sejam invocados remotamente por outros componentes do sistema, como o servidor de aplicação e que o objeto seja facilmente encontrada através apenas do seu nome.

Dessa forma, para o servidor de aplicação invocar um método do banco de dados remotamente basta ele criar um `proxy` para o objeto remoto e, em seguida, chamar o método desejado. A figura 4 ilustra um exemplo desse processo.



```
def adicionar_baralho(username, baralho):
    database = Pyro5.client.Proxy("PYRONAME:db-server")

    response = database.get_qtd_baralhos(username)

    if response != 'Usuário não encontrado':
        qtd_baralhos = int(response)

        if qtd_baralhos == 3:
            response = 'Quantidade máxima de baralhos atingida!'
            return response

        response = database.adicionar_baralho(username, baralho)

    return response
```

Figura 4 - Exemplo de Invocação Remota de um Método do Servidor de Banco de Dados

Ao comparar o código acima, que mostra como a invocação remota de um método é realizada usando o Pyro5, com o código da implementação anterior utilizando a API de Sockets, como mostra a Figura 5 disponível abaixo, fica evidente o quanto o middleware simplifica e abstrai o processo de comunicação.

```
def adicionar_baralho(username, baralho):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as bd_client:
        bd_client.connect((db_host, db_port))
        request = f"get_qtd_baralhos {username}"
        bd_client.send(request.encode('utf-8'))
        response = bd_client.recv(1024).decode('utf-8')

        if response != 'Usuário não encontrado':
            qtd_baralhos = response

            if qtd_baralhos == 3:
                response = 'Quantidade máxima de baralhos atingida!'
                return response

            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as bd_client:
                bd_client.connect((db_host, db_port))
                request = f"adicionar_baralho {username} {baralho}"
                bd_client.send(request.encode('utf-8'))
                response = bd_client.recv(1024).decode('utf-8')

    return response
```

Figura 5 - Exemplo de como a comunicação entre o Servidor de Aplicação e o Servidor de Banco de Dados era feita com uso da API de Sockets

2.2.3. Mudanças no servidor de aplicação

Na implementação com Sockets, o servidor de aplicação possuía dois arquivos principais e fundamentais: um responsável pela iniciação do servidor e pelas chamadas das funções para gerenciar o recebimento e envio de mensagens, e outro contendo as funcionalidades relativas ao gerenciamento do jogo. Com o uso do Pyro5, o gerenciamento das requisições foi alterado para ser realizado por um objeto `Server` (neste caso, um objeto local que é exposto pelo middleware como um objeto remoto), que contém todos os métodos disponíveis para a comunicação via cliente e servidor.

Já para a implementação das funcionalidades, em vez de utilizar os métodos `send` e `receive` do Sockets para se comunicar com o servidor de banco de dados, com o uso do middleware, basta criar um `proxy` para o objeto `Database`, como mostra a figura 6, para acessar remotamente os métodos disponíveis para processar as requisições.

```
def adicionar_carta_colecao(username, carta):
    database = Pyro5.client.Proxy("PYRONAME:db-server")

    response = database.adicionar_carta_na_colecao(username, carta)

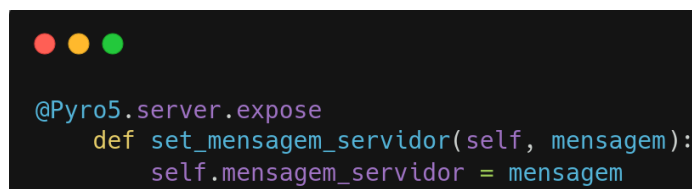
    return response
```

Figura 6 - Exemplo de Invocação Remota de um Método do Servidor de Banco de Dados

2.2.3. Mudanças no cliente

Na implementação com `Sockets`, ao realizar o login, o cliente iniciava uma `thread` que escutava em uma porta conhecida tanto por ele quanto pelo servidor de aplicação. Essa `thread` era responsável por receber requisições relacionadas a convites para jogar partidas ou sobre uma partida específica.

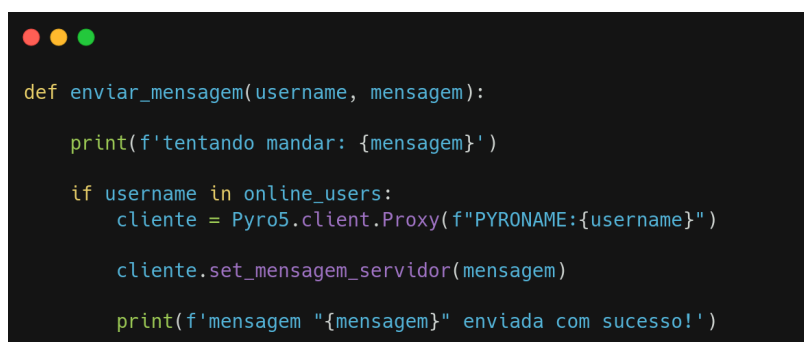
Com o uso do middleware `Pyro5`, o processo se tornou mais simples. Quando um usuário realiza o login, o cliente inicia um servidor `Pyro` com o nome do usuário, que expõe um único método remotamente, o `set_mensagem_servidor()`. Isso permite que o cliente atue como servidor em momentos específicos. Com isso, para que o servidor de aplicação envie mensagens ao cliente, ele só precisa criar um `proxy` para representar o objeto cliente e fazer uma chamada remota ao método disponível. Dessa forma, em vez de depender do endereço IP e da porta do cliente, o servidor só precisa conhecer o nome de usuário, que é uma informação do negócio.



```
@Pyro5.server.expose
def set_mensagem_servidor(self, mensagem):
    self.mensagem_servidor = mensagem
```

Figura 7 - Método exposto como um serviço remoto para o Servidor se comunicar com um Cliente

Nas funcionalidades do servidor, para adaptar o envio de mensagens ao uso do middleware, foi necessário alterar apenas o método de envio de mensagens. As figuras 8 e 9 ilustram a comparação entre a implementação original usando a API de `Sockets` e a nova abordagem com o middleware `Pyro5`. O conteúdo e o formato das mensagens foram mantidos, garantindo que a interface não precisasse ser modificada.



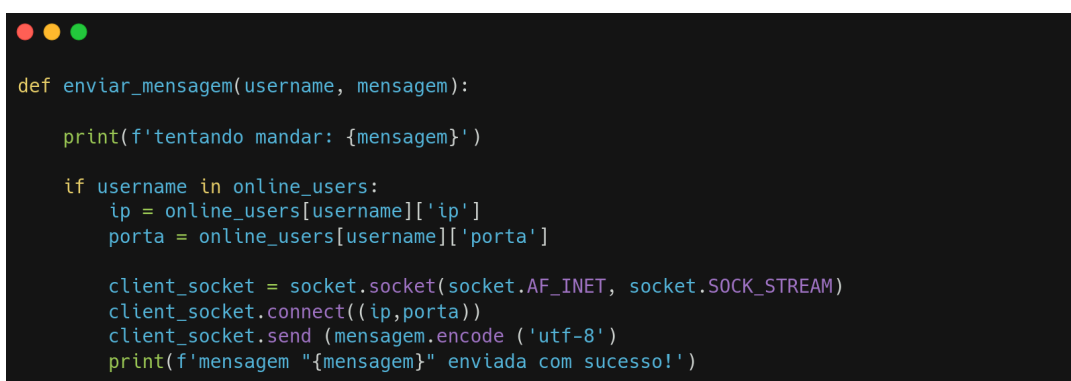
```
def enviar_mensagem(username, mensagem):
    print(f'tentando mandar: {mensagem}')

    if username in online_users:
        cliente = Pyro5.client.Proxy(f"PYRONAME:{username}")

        cliente.set_mensagem_servidor(mensagem)

        print(f'mensagem "{mensagem}" enviada com sucesso!')
```

Figura 8 - Envio de Mensagens do Servidor de Aplicação para o Cliente com Middleware



```
def enviar_mensagem(username, mensagem):
    print(f'tentando mandar: {mensagem}')

    if username in online_users:
        ip = online_users[username]['ip']
        porta = online_users[username]['porta']

        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client_socket.connect((ip, porta))
        client_socket.send(mensagem.encode('utf-8'))
        print(f'mensagem "{mensagem}" enviada com sucesso!')
```

Figura 9 - Envio de Mensagens do Servidor de Aplicação para o Cliente com API de Sockets

As figuras abaixo ilustram o processo de inicialização do servidor do lado do cliente. Quando o usuário realiza o login, e a autenticação é bem-sucedida, o cliente inicia um servidor local em uma thread separada por meio do método `start_listen_server()`. Este método configura um daemon Pyro5, que é registrado no serviço de nomes do Pyro5 utilizando o nome de usuário que realizou o login. Assim, o método `set_mensagem_servidor()`, exposto como um serviço remoto via Pyro5 através do modificador `@Pyro5.server.expose`, fica disponível para que o servidor de aplicação o utilize para enviar convites para partidas ou atualizações de status diretamente ao cliente.

```
def start_listen_server(self):
    try:
        self.daemon = Pyro5.server.Daemon(host="localhost")
        ns = Pyro5.core.locate_ns()
        uri = self.daemon.register(self)
        ns.register(self.username, uri)
        print(f'Ready {self.username}. Object uri = {uri}')
        self.daemon.requestLoop()
    except Exception as e:
        print(f'Erro ao iniciar o servidor de escuta: {str(e)}')

def login(self, username, senha):
    try:
        server = Pyro5.client.Proxy("PYRONAME:server")
        response = server.login(username, senha)

        if response == "Login feito com sucesso!":
            self.username = username
            # Inicia o servidor de escuta em uma thread separada
            listen_thread = threading.Thread(target=self.start_listen_server)
            listen_thread.start()

            return True, response
        return False, response
    except Exception as e:
        print(f'Erro ao realizar login:{str(e)}')
        return False, f'Servidor Indisponível: Reinicie o Sistema!"
```

Figura 10 - Implementação do método `start_listen_server()`

O daemon do cliente é armazenado como um atributo para permitir que ele seja encerrado quando o cliente realiza logout, pois dessa forma, o daemon pode ser finalizado de maneira controlada utilizando o método `shutdown()`.

```
def logout(self):
    try:
        server = Pyro5.client.Proxy("PYRONAME:server")
        response = server.logout(self.username)
        print(response)

        if response == 'Logout feito com sucesso!':
            # Encerrar o daemon do servidor do cliente
            if self.daemon:
                self.daemon.shutdown()
                print(f'Servidor cliente {self.username} encerrado.")

            return True, response
        return False, response

    except Exception as e:
        print(f'Erro ao realizar logout:{str(e)}')
        return False, f'Erro ao realizar logout:{str(e)}"
```

Figura 11 - `logout()`

3. Conclusão

Durante o desenvolvimento do sistema `EmojiCards` com programação via middleware `Pyro5`, foram enfrentadas algumas dificuldades relacionadas ao uso do middleware em si, uma vez que a programação orientada à comunicação entre objetos distribuídos em diferentes máquinas era uma experiência nova para o grupo. A principal dificuldade foi compreender como o middleware funcionava, especialmente no que diz respeito à publicação de objetos para torná-los acessíveis por meio de nomes no sistema de nomes do `Pyro5`. No início, foram enfrentados problemas para configurar corretamente os objetos dos servidores de aplicação e banco de dados para que eles pudessem ser registrados e localizados pelo cliente.

Outra dificuldade foi a necessidade de adaptar o fluxo do cliente para que ele pudesse atuar simultaneamente como cliente e servidor. Antes, a implementação era realizada diretamente com `threads` e `sockets`, permitindo um controle mais direto da comunicação. A transição para o uso do `Pyro5` exigiu uma mudança na abordagem, especialmente para garantir que o cliente pudesse responder a chamadas remotas enquanto continuava a fazer suas próprias requisições. Para resolver essa questão, foi necessário instanciar um servidor dentro do cliente no momento do login, permitindo que o usuário recebesse mensagens específicas diretamente do servidor de aplicação. Esse objeto foi, então, publicado pelo middleware, possibilitando a comunicação direta e eficaz entre o cliente e o servidor.

Apesar desses desafios, conseguimos implementar com sucesso o sistema distribuído proposto, utilizando o middleware `Pyro5`. Após as dificuldades iniciais, o sistema não sofreu alterações significativas, pois foi necessário modificar apenas os arquivos relacionados ao gerenciamento de requisições, enquanto o restante da estrutura permaneceu semelhante ao trabalho anterior. Além disso, com o uso do middleware, os códigos ficaram menores e mais concisos, graças à abstração proporcionada pelo `Pyro5`, que simplificou a comunicação entre objetos distribuídos. Cada obstáculo enfrentado contribuiu para o aprimoramento das habilidades dos membros do grupo e para o desenvolvimento de um sistema mais robusto e funcional. Por fim, alcançamos o objetivo principal do trabalho, demonstrando nossa capacidade de lidar com sistemas distribuídos e o potencial do `Pyro5` em facilitar a comunicação entre objetos distribuídos.

4. Referências

[1] Documentação `Pyro5`. Disponível em:
<https://pyro5.readthedocs.io/en/latest/intro.html>