



## **Trabalho Prático de Sistemas Distribuídos e Paralelos - Parte 5**

Marcos Veniciu de Sá Barbalho - 3016  
Guilherme Corrêa Melos - 3882  
Samuel Aparecido Delfino Rodrigues - 3476

Florestal - MG  
09 de Julho de 2023

Marcos Veniciu de Sá Barbalho 3016  
Guilherme Corrêa Melos - 3882  
Samuel Aparecido Delfino Rodrigues - 3476

## **Trabalho Prático de Sistemas Distribuídos e Paralelos - Parte 5**

Florestal - MG  
09 de Julho de 2023

# Sumário

<b>Introdução</b>	<b>4</b>
<b>Desenvolvimento</b>	<b>4</b>
Execução	4
Servidor	4
Servidor remoto	6
Web service	7
Cliente	8
Objetos	8
Telas	10
<b>Conclusão</b>	<b>11</b>
<b>Anexos</b>	<b>12</b>
Anexo 1: Tela de login	12
Anexo 2: Perfil usuário	13
Anexo 3: Lista de receitas do usuário	14
Anexo 4: Perfil amigo	15
Anexo 5: Adicionar amigo	16
Anexo 6: Mostra receita	17
Anexo 7: Calcular calorias (web service)	18

# Introdução

Nesta parte do projeto de desenvolvimento de uma rede social para o compartilhamento de receitas entre amigos, o app Vó Maria. O objetivo era a utilização do *Middleware Pyro5* para realizar a comunicação entre o cliente e o objeto remoto e a adição de uma *web service rest*, utilizando o flask.

## Desenvolvimento

### Execução

Para executar o programa é necessário abrir o web service, name server do pyro5, o servidor e o cliente. Para isso basta abrir a pasta vomaria e abrir o terminal nesta pasta e executar o makefile, nele há três funções: ***install***, ***run*** e ***run\_dependencies***.

Para instalar o ambiente virtual basta executar o seguinte comando no terminal:

***make install***

Para executar o web service, o name server, o servidor e o cliente basta executar o comando:

***make run***

Existe a possibilidade de que bibliotecas como Tkinter e Sqlite precise ser instaladas no sistema operacional, para isso use o comando:

***make run\_dependencies***

Para poder testar o sistema, pode ser feito o login com a seguinte conta:

usuário: **canna**

senha: **1234**

Um possível amigo a ser adicionado seria: ***algodaodoce***.

Para a receita pode ser adicionado qualquer coisa.

Para testar o web service, basta informar a quantidade (em gramas) e o nome do ingrediente.

Existe uma lista de ingredientes que a API tem suporte: ***arroz, fermento biologico, creme de leite, ovos, batata, oleo, queijo parmesao, polvilho doce, pimenta do reino***.

## Servidor

O servidor foi modelado utilizando uma arquitetura de repositório, contendo as seguintes camadas:

- **models.py** - Camada que modela as classes. Contém as seguintes classes:
  - **User**: Define o usuário do aplicativo
  - **Recipe**: Define uma receita
  - **Relationship**: Define o relacionamento entre dois usuários. Um usuário pode seguir um segundo usuário e vice-versa.
- **repository.py** - Define a camada de repositório para cada um dos modelos, acessa diretamente o banco de dados, realiza operações de atualização e pesquisa no banco de dados.

- service.py - Define a camada de serviço para cada um dos modelos propostos, entra em contato direto com a camada de repositório para requisitar serviços.
- database.py - Define o banco de dados, implementado com a biblioteca sqlite3
- rmi.py - Classe que representa o servidor remoto responsável por fornecer serviços ao cliente remotamente.

Por simplicidade, não utilizamos uma camada de controle, a camada de serviço é a camada que usuário entrará em contato pela interface de usuário.

```
class User:
    def __init__(self, uuid, username, password, description) -> None:
        self.uuid = uuid
        self.username = username
        self.password = password
        self.description = description
        self.recipes = []
        self.followers = []
        self.following = []

class Recipe:
    def __init__(self, uuid, user_uuid, title, recipe, likes, created_at) -> None:
        self.uuid = uuid
        self.user_uuid = user_uuid
        self.title = title
        self.recipe = recipe
        self.likes = likes
        self.created_at = created_at

class Relationship:
    def __init__(self, uuid: str, user_uuid1: str, user_uuid2: str) -> None:
        self.uuid = uuid
        self.user_uuid1 = user_uuid1
        self.user_uuid2 = user_uuid2
```

Imagem 1. Camada de modelo, models.py

```
class RecipeRepository:
    def __init__(self) -> None: ...
    def create(self, recipe: Recipe): ...
    def find_by_user_uuid(self, user_uuid): ...
    def find_by_uuid(self, uuid): ...
    def find_by_name(self, name): ...
    def like_recipe(self, name): ...

class UserRepository:
    def __init__(self) -> None: ...
    def find_user_by_username(self, username): ...
    def create(self, user: User): ...

class RelationshipRepository:
    def __init__(self) -> None: ...
    def create(self, relationship: Relationship): ...
    def find(self, user_id): ...
```

Imagem 2. Camada de repositório, repository.py

```

class RelationshipService:
    def __init__(self):~

    def add(self, username_1: str, username_2: str):~

    def user_follows(self, username):~

class UserService:
    def __init__(self):~

    def create_account(self, username: str, password: str, description: str):~

    def login(self, username: str, password: str):~

    def user_description(self, username):~
class RecipeService:
    def __init__(self) -> None:~

    def add(self, username, title, recipe):~

    def get_recipe_by_user(self, username):~

    def get_recipe_by_title(self, title):~

    def like_recipe(self, title):~

```

Imagem 3. Camada de serviço, service.py

## Servidor remoto

Durante o desenvolvimento do trabalho, o servidor, desenvolvido no arquivo rmi.py, foi modelado diretamente para suportar receber invocações de métodos remotos do cliente, sendo assim o trabalho possui apenas dois objetos distribuídos se comunicando: o servidor e o cliente. O servidor se conecta unicamente com a camada de serviço de cada objeto definido e lida com os procedimentos remotos. Sendo assim, precisamos de criar apenas um nameserver para as chamadas de procedimentos remotos, definido como nameserver “servidor”.

```

1  import Pyro5.api
2  import threading
3  from service import UserService, RelationshipService, RecipeService
4
5
6  @Pyro5.api.expose
7  class RemoteServer():
8  >   def __init__(self) -> None:~
13 >   def login(self, username, password):~
15 >   def get_friends(self, username):~
17 >   def get_recipes(self, username) -> list:~
25 >   def get_recipe_by_title(self, title):~
27 >   def get_ingredients_list(self, title):~
37 >   def get_preparation_mode(self, title):~
42 >   def get_likes(self, title):~
45 >   def get_user_description(self, username):~
47 >   def follow_user(self, username, user_to_follow):~
49 >   def add_recipe(self, username, title, ingredients, preparation_mode):~
60 >   def like_recipe(self, name):~
62 >   def run(self):~
74 >   def thread(self):~
78
79  if __name__ == '__main__':
80   server = RemoteServer()
81   server.thread()

```

Imagem 4. Servidor remoto e seus métodos

Utilizamos a biblioteca *threading* para realizar a implementação de threads, permitindo assim que múltiplos usuários façam requisições ao servidor.

```

def run(self):
    daemon = Pyro5.api.Daemon()

    uri = daemon.register(self)

    ns = Pyro5.api.locate_ns()

    ns.register("servidor", uri)

    print("Servidor aguardando conexões...")
    daemon.requestLoop()
def thread(self):
    server_thread = threading.Thread(target=self.run)
    server_thread.start()
    server_thread.join()

if __name__ == '__main__':
    server = RemoteServer()
    server.thread()

```

Imagem 5. Implementação de threads do servidor.

## Web service

```

1 from flask import Flask
2
3 calories_100g = {"arroz": 500, "fermento biologico" : 50,
4                  "creme de leite": 70, "ovos": 400, "batata": 300,
5                  "oleo": 170, "queijo parmesao": 230,
6                  "polvilho doce": 103, "pimenta do reino": 114,
7                  }
8
9 app = Flask(__name__)
10
11 @app.route('/hello/<string:ingrediente>/<int:quantidade>')
12 def calorias(ingrediente, quantidade):
13     return str(calories_100g[str.lower(ingrediente)] * quantidade / 100)
14

```

Imagem 6. Implementação do web service.

O web service que escolhemos fazer, é um serviço que informa a quantidade de calorias que uma determinada quantidade de um ingrediente possui, ele recebe o nome do ingrediente e a quantidade em gramas do ingrediente que será usado. Ele vai retornar a quantidade de calorias

proporcionalmente ao que foi passado. Usamos um dicionário para ter o nome dos ingredientes e a quantidade de calorias que tem em 100 gramas.

Dentro do aplicativo, tem a tela de calcular calorias, em que o usuário informa a quantidade e o nome do ingrediente, e a medida que mais ingredientes são passados é gerada a lista de ingredientes e a quantidade de calorias, e na parte inferior tem a quantidade total de calorias.

## Cliente

### Objetos

- usuario.py

```
4
3 class Usuario:
4     """
9     def __init__(self):
10         """
13
14         self.nome = ""
15         self.descricao = ""
16         self.lista_amigos = []
17         self.lista_receita = []
18
19     def get_usuario(self):--
28
29     def get_objeto_remoto(self):--
33
34     def _update_lista_amigos(self):--
40
41     def _update_lista_receitas(self):--
47
48     def get_nome(self):--
53
54     def get_descricao(self):--
65
66     def get_lista_amigos(self):--
71
72     def get_lista_receitas(self):--
77
78     def add_amigo(self, usuario, name):--
89
90     def add_receita(self, usuario, receita_name, ingredientes, modo_preparo):--
97
98     def login(self, nome, senha):--
```

Imagem 7. Implementação da classe usuário

A classe **usuario** é responsável por gerenciar os dados do usuário logado, tendo como atributos o nome, a descrição do perfil, a lista de amigos e a lista de receitas. Para isso, ela tem a operação *login()* que verifica se o nome e senha informado corresponde a algum usuário cadastrado.

```
def get_objeto_remoto(self):
    # Obtém uma referência ao objeto remoto registrado no Name Server
    uri = Pyro5.api.locate_ns().lookup("servidor")
    return Pyro5.api.Proxy(uri)
```

imagem 8. Função que conecta ao objeto remoto.

A função *get\_objeto\_remoto()* é responsável por conectar ao objeto remoto.



```

8
9     def get_usuario(self):
10         """
11         Monta o objeto usuario, buscando os seus atributos no servidor
12         """
13         objeto_remoto = self.get_objeto_remoto()
14
15         self.descricao = objeto_remoto.get_user_description(self.nome)
16         self.lista_amigos = objeto_remoto.get_friends(self.nome)
17         self.lista_receita = objeto_remoto.get_recipes(self.nome)
18

```

imagem 9. Função `get_usuario`, que recupera os dados do usuário.

A função `get_usuario()` é responsável por utilizar as operações do objeto remoto e solicitar os dados do usuário.

As funções `add_amigo()` e `add_receita()` são responsáveis por adicionar novos amigos e novas receitas no banco de dados. Para isso, elas utilizam o objeto remoto para adicionar os dados no banco de dados.

As funções `update_lista_amigos()` e `update_lista_receitas()` são usadas para atualizar a lista de amigos e a lista de receitas sempre que é realizada uma alteração em alguma delas.

As demais são para retornar os dados no objeto usuário para a interface.

- amigo.py

```

You, há 15 horas | 2 authors (You and others)
class Amigo():
    def __init__(self):
        self.nome = ""
        self.descricao = ""
        self.lista_receitas = []

> def get_objeto_remoto(self): ...
> def get_amigo(self, username): ...
> def get_nome(self): ...
> def get_descricao(self): ...
> def get_lista_receitas(self): ...

```

imagem 10. Implementação da classe `amigo`.

A classe **Amigo** é responsável por gerenciar os dados de um amigo. Sempre que o perfil de um dos amigos da lista de amigos é selecionado, os dados desse usuário são carregados para o objeto amigo. Os dados dos amigos que são exibidos são o nome, a descrição do perfil e a lista de receitas.

A função `get_objeto_remoto()` é responsável por conectar ao objeto remoto.

A função `get_amigo()` é responsável por utilizar as operações do objeto remoto e solicitar os dados do amigos selecionado.

- receita.py

```

Guilherme Correa Melos, há 14 horas | 2 authors (You and others)
class Receita():
    def __init__(self):
        self.titulo = ""
        self.lista_ingredientes = []
        self.modos_preparo = ""
        self.garfadas = 0

    def get_objeto_remoto(self): ...

    def get_receita(self, title): ...

    def get_titulo(self): ...

    def get_lista_ingredientes(self): ...

    def get_modos_preparo(self): ...

    def get_garfadas(self): ...

    def set_garfada(self): ...

    def _update_garfadas(self): ...

```

imagem 11. Implementação da classe receitas.

A classe **receita** é responsável por gerenciar os dados de uma receita. Sempre que uma receita é selecionada, os dados dessa receita são carregados para o objeto receita. Os dados das receitas que são exibidos são o título, a lista de ingredientes, o modo de preparo e a quantidade de garfadas que a receita tem.

A função `get_objeto_remoto()` é responsável por conectar ao objeto remoto.

A função `get_receita()` é responsável por utilizar as operações do objeto remoto e solicitar os dados da receita selecionada.

## Telas

- Tela de Login  
Na tela de login, o usuário informa um nome de usuário e a senha. Ao clicar no botão logar ele pode receber o aviso de login invalido caso tenha digitado algo invalido, caso contrário vai para a tela de perfil do usuário logado. Como mostrado no Anexo 1
- Tela de perfil  
Na tela de perfil, possui o nome do usuário, uma descrição do seu perfil e um botão para a lista de receitas do próprio usuário, uma lista de amigos e um botão para adicionar amigos como mostrado no Anexo 2, e ao clicar no botão minha receita, é mostrada uma tela como a do perfil do amigo, com a diferença de ter a opção de adicionar receita, como mostrado no Anexo 3. Ao clicar no nome de um dos amigos ele é enviado para o perfil desse amigo, que contém o nome, a descrição do perfil e a lista de receitas desse amigo, e um botão para voltar para a tela do próprio usuário, como mostrado no Anexo 4.

- Tela adicionar amigo  
Nessa tela tem a opção de digitar um nome para ser adicionado à lista de amigos. Caso o usuário não esteja cadastrado é exibido um aviso. Como mostrado no Anexo 5.
- Tela Adicionar receita  
Nessa tela é exibida as informações sobre a receita selecionada, sendo o título, a lista de ingredientes e o modo de preparo, e também um botão que mostra o número de garfadas que essa receita teve. Como mostrado no anexo 6.
- Tela Calcular calorias  
Nesta tela há dois campos para o usuário digitar, a primeira é a quantidade do ingrediente que ele quer saber as calorias, nesse campo ele deve informar a quantidade de gramas que ele vai usar. O segundo campo é o nome do ingrediente que ele quer saber as calorias. Ao informar esses dois campos será exibido na tela a quantidade de calorias que a quantidade de ingrediente informada possui, e na parte inferior terá a quantidade total de calorias, que é o somatório de calorias de todos os ingredientes informados. Como pode ser visto no anexo 7.

## Conclusão

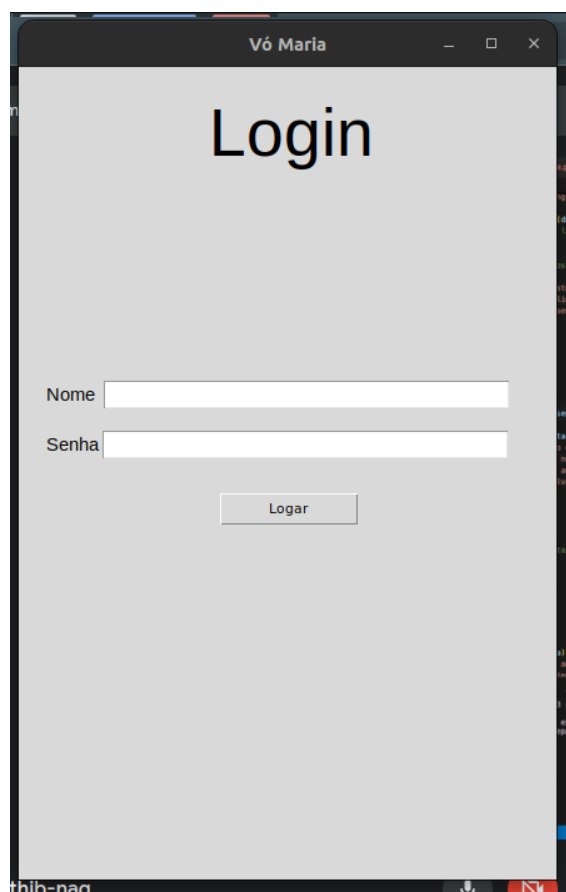
Neste trabalho, pudemos implementar um sistema distribuído para o compartilhamento de receitas entre usuários que usasse o Middleware Pyro5. Em relação ao anterior que utilizava API socket e multithreading, a implementação usando um middleware foi bem mais simples incluindo adicionar o multithreading. O uso de objetos remotos tornou o desenvolvimento do cliente bem mais simples e intuitivo do que antes, a programação pode ser feita com se estivesse utilizando um objeto local. Ao adicionarmos o web service, ele foi algo bem simples de implementar e adicionar ao app já pronto.

Ao comparar os três métodos, o API Socket foi o mais complicado de usar, por nele tivemos de usar strings para poder enviar as solicitações e receber os resultados. Para enviar tivemos de colocar o identificador e os parâmetros na string e para receber, precisamos juntar os dados e depois tivemos de separar os dados na string. O web service foi bem simples de implementar e usar, mas se tivéssemos usado mais operações além do envio de calorias ele poderia ter se tornado algo mais trabalhoso de implementar no app. Nesse sentido, o uso do Middleware Pyro5 foi o melhor de trabalhar, já que no servidor implementamos os objetos e no cliente usamos como se fossem objetos locais, o que tornou mais simples de usar e pareceu mais simples de adicionar e usar mais funcionalidade que o web service.

Neste trabalho dividimos em duas partes principais, a escolha e implementação do web service e a criação da tela e utilização do serviço. Para isso dividimos, o marcos ficou responsável pela tela e inserir o serviço dentro do app, enquanto o guilherme e o samuel ficaram responsáveis pela escolha e implementação do web service.

# Anexos

## Anexo 1: Tela de login



The image shows a screenshot of a web application window titled "Vó Maria". The window contains a login form with the following elements:

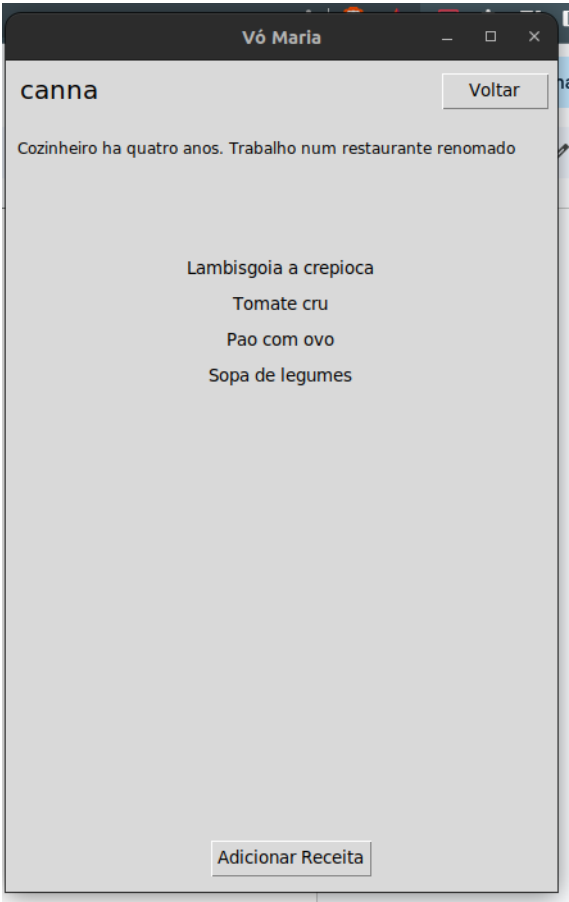
- A large heading "Login" at the top.
- A label "Nome" followed by a text input field.
- A label "Senha" followed by a text input field.
- A button labeled "Logar" centered below the input fields.

The background of the form is light gray, and the window has a dark title bar with standard minimize, maximize, and close buttons.

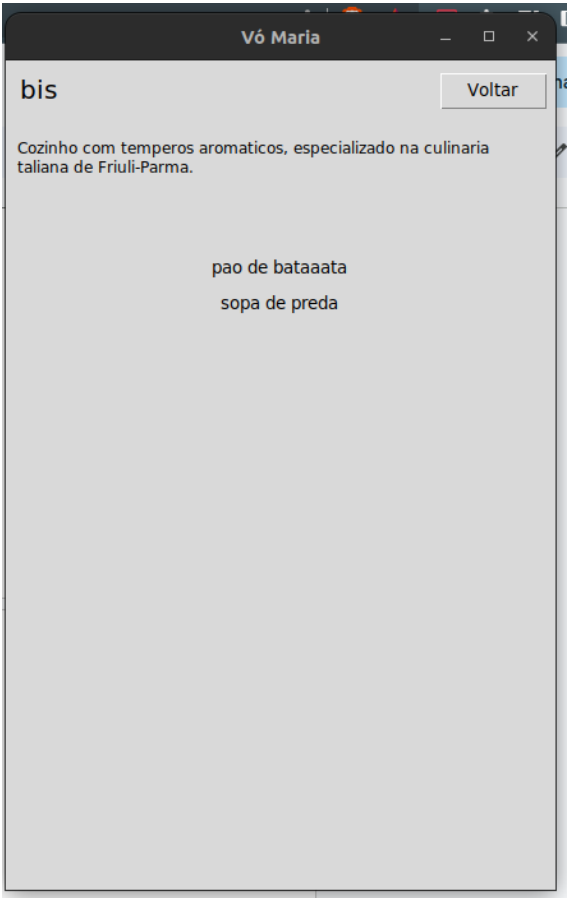
Anexo 2: Perfil usuário



Anexo 3: Lista de receitas do usuário



Anexo 4: Perfil amigo



Anexo 5: Adicionar amigo

Vó Maria

Adicionar Amigo

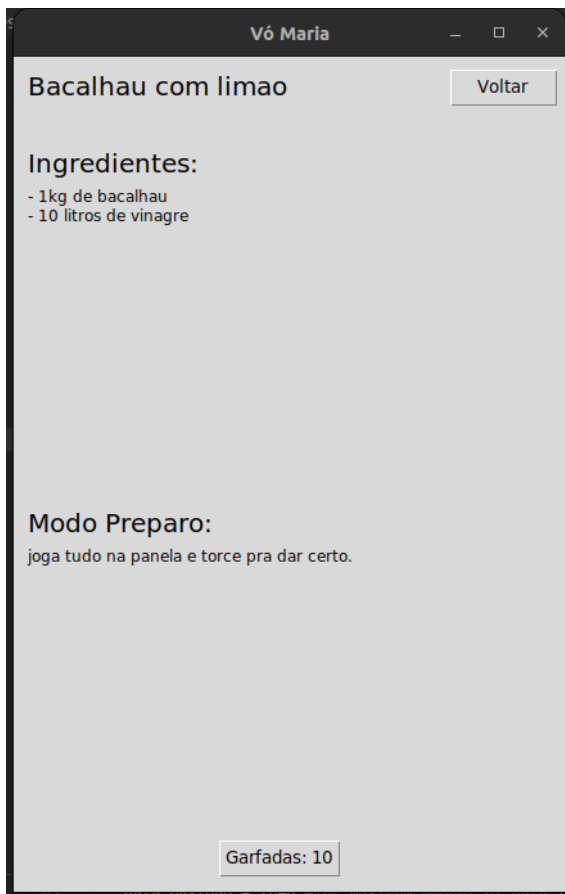
Voltar

Nome

Adicionar Amigo



## Anexo 6: Mostra receita



Anexo 7: Calcular calorias (web service)

Vó Maria

Calcular Calorias

Voltar

Quantidade (gramas)

Nome ingrediente:

Add

- 100g arroz: 500.0 calorias  
- 200g batata: 600.0 calorias  
- 50g pimenta do reino: 57.0 calorias  
- 150g ovos: 600.0 calorias

Total de calorias: 1757.0