



# Universidade Federal do Ceará Campus Quixadá

## **TRABALHO FINAL DE SISTEMAS DISTRIBUÍDOS “TAMAGOTCHI”**

Larissa da Silva Matos  
Paula Araújo Feitosa

Trabalho final referente à disciplina de  
Sistemas Distribuídos, ministrada pelo  
professor Marcos Dantas Ortiz.

Quixadá  
2023

## Introdução

Este relatório descreve um serviço remoto para gerenciar e interagir com pets virtuais. O serviço oferece uma série de métodos remotos para brincar, alimentar e colocar pets virtuais para dormir. Cada método remoto possui diferentes entradas e saídas. Vale ressaltar que este projeto foi feito em Python - tanto o cliente quanto o servidor - e para realizar a serialização e desserialização dos dados, foi utilizado o JSON. O link para o código está disponível em [Link do Trabalho](#).

## Desenvolvimento

O projeto possui 6 diferentes arquivos:

- client.py
- proxy.py
- server.py
- dispatcher.py
- skeleton.py
- servant.py

O **arquivo client.py** importa a classe Proxy do arquivo proxy.py e cria uma instância dela, passando o endereço do servidor como parâmetro. Em seguida, o cliente pode invocar métodos do objeto remoto usando o método *invoke\_method* do proxy, que recebe o nome do método e os argumentos.

Abaixo temos uma descrição detalhada do código:

### Instanciação do Proxy:

- A linha `proxy = Proxy('localhost', 8080)` cria uma instância da classe Proxy, especificando um endereço de localhost ('localhost') e a porta 8080.

### Chamadas de Métodos Remotos:

- As linhas subsequentes fazem chamadas de método usando o objeto `proxy.invoke_method()`. Cada chamada passa o nome do método remoto a ser invocado e seus parâmetros, que são passados como argumentos separados para o método *invoke\_method*.

Esse método pode chamar os três métodos de interação com o cliente citados anteriormente, como mostra a Figura 1.

Figura 1: Métodos remotos chamados pelo *invoke\_method*.

```
result_eat = proxy.invoke_method("eat", "banana", "juice")
print("Result of eat method:", result_eat)

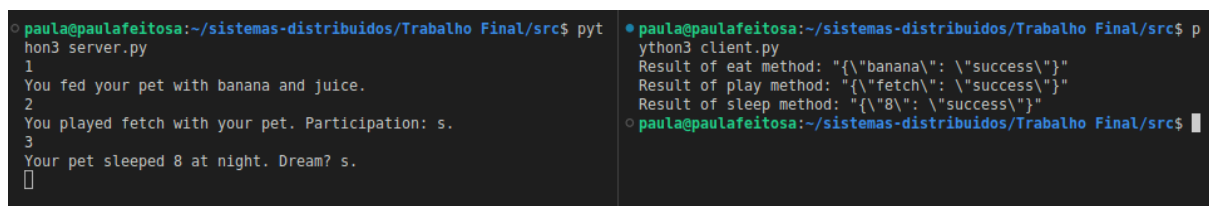
result_play = proxy.invoke_method("play", "fetch", "s")
print("Result of play method:", result_play)

result_sleep = proxy.invoke_method("sleep", "8", "s")
print("Result of sleep method:", result_sleep)
```

## Exibição dos Resultados:

- Após cada chamada de método remoto, o código imprime os resultados obtidos. Atualmente o cliente possui mensagens padrão a serem enviadas explorando os três métodos existentes no serviço remoto. O resultado é composto pelo nome de um dos argumentos e a mensagem “success” para indicar que a chamada e a resposta ocorreram como esperado. Esse resultado pode ser visto na Figura 2.

Figura 2: Terminal após executar o cliente e o servidor.



```
o paula@paulafeitosa:~/sistemas-distribuidos/Trabalho Final/src$ pyth
hon3 server.py
1
You fed your pet with banana and juice.
2
You played fetch with your pet. Participation: s.
3
Your pet slept 8 at night. Dream? s.
o paula@paulafeitosa:~/sistemas-distribuidos/Trabalho Final/src$ p
ython3 client.py
Result of eat method: "{\"banana\": \"success\"}"
Result of play method: "{\"fetch\": \"success\"}"
Result of sleep method: "{\"8\": \"success\"}"
o paula@paulafeitosa:~/sistemas-distribuidos/Trabalho Final/src$
```

## Arquivos do Projeto

O **arquivo proxy.py** define a classe Proxy, que é responsável por enviar e receber mensagens do servidor usando sockets UDP. O método *invoke\_method* cria uma mensagem em formato JSON, que contém o tipo, o id, a referência do objeto, o nome do método e os argumentos. Essa mensagem é codificada em bytes e enviada para o servidor. Em seguida, o proxy espera receber uma resposta do servidor, que também é uma mensagem em formato JSON, e a decodifica em um objeto Python.

Abaixo está sendo apresentada uma descrição detalhada do código:

### Classe Proxy:

- `__init__(self, server_address)`: O construtor da classe recebe um endereço de servidor (`server_address`) como parâmetro e inicializa as seguintes variáveis:
  - `self.server_address`: Armazena o endereço do servidor ao qual as solicitações serão enviadas.
  - `self.sock`: Cria um socket UDP (`socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`) para comunicação com o servidor.
  - `self.sock.settimeout(5.0)`: Define um tempo limite de 5 segundos para as operações de socket.
  - `self.request_id`: Inicializa um contador para gerar IDs de requisição.
- `invoke_method(self, method_name, *args)`: Método para invocar métodos remotos no servidor. Ele recebe o nome do método (`method_name`) e argumentos variáveis (`*args`) que serão passados para o método remoto.

- Incrementa o `self.request_id` para gerar um ID único para cada requisição.
- Prepara uma mensagem JSON estruturada contendo o tipo de requisição, ID da requisição, referência do objeto ("servant"), ID do método e argumentos passados.
- Codifica a mensagem JSON em bytes e a envia para o servidor remoto usando `self.sock.sendto(payload, self.server_address)`.
- O código faz um loop para tentar receber uma resposta do servidor remoto, tratando possíveis timeouts e reenviando a requisição em caso de falha.
- Se a resposta for recebida com sucesso, ela é decodificada de JSON para um objeto Python e retornada como resultado da chamada de método remoto.
- A lógica dentro do método `invoke_method` garante que a comunicação com o servidor remoto seja robusta, lidando com timeouts e reenviando a requisição em caso de falha na recepção da resposta. Um exemplo de resposta de falha está sendo exibido na Figura 3.

Figura 3: Terminal após o *timeout* do cliente expirar.

```
File "/home/paula/sistemas-distribuidos/Trabalho Final/src/proxy.py", line 51, in invoke_method
    raise Exception("Servidor não respondeu após {} tentativas".format(max_tries))
Exception: Servidor não respondeu após 3 tentativas
```

O **arquivo server.py** define a classe `UDPServer`, que é responsável por criar e gerenciar o socket do servidor. O servidor cria uma instância da classe `Servant` do arquivo `servant.py`, que representa o objeto remoto que será invocado pelo cliente. O servidor também cria uma instância da classe `Skeleton` do arquivo `skeleton.py`, que é responsável por chamar os métodos do `servant`. Além disso, o servidor cria uma instância da classe `Dispatcher` do arquivo `dispatcher.py`, que é responsável por despachar as requisições do cliente para o `skeleton`. O servidor fica em um *loop* infinito, esperando receber mensagens do cliente. Quando recebe uma mensagem, ele a passa para o despachante, que retorna o resultado da invocação do método. O servidor então envia o resultado de volta para o cliente.

Abaixo temos uma descrição detalhada do código:

### Classe `UDPServer`:

- `__init__(self, server_address)`: O construtor recebe um endereço de servidor (`server_address`) e inicializa variáveis importantes:
  - `self.server_address`: Armazena o endereço do servidor.

- *self.sock*: Cria um socket UDP (`socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`) e o vincula ao endereço do servidor usando `self.sock.bind(self.server_address)`.
- *self.history*: Inicializa um dicionário vazio (`dict()`) para manter o histórico de requisições e respostas dos clientes.
- *start(self)*: Método principal para iniciar o servidor:
  - Cria instâncias dos objetos *Servant*, *Skeleton* e *Dispatcher*.
  - Inicia um loop infinito para aguardar requisições dos clientes.
  - Recebe dados (mensagens) e o endereço do cliente (*client\_address*) usando `self.sock.recvfrom(4096)`. Os dados recebidos são decodificados de JSON para um dicionário (*payload*) usando `json.loads(data.decode('utf-8'))`.
  - Obtém o ID da requisição (*request\_id*) a partir do *payload*.
  - Verifica se o cliente já fez requisições anteriores. Se sim, procura a resposta correspondente no histórico (*self.history*) com base no endereço do cliente e no ID da requisição.
  - Se não houver histórico para o cliente ou se a requisição não foi encontrada no histórico, despacha a requisição para o *Dispatcher*, obtém a resposta e a armazena no histórico.
  - Envia a resposta de volta ao cliente usando `self.sock.sendto(json.dumps(result).encode('utf-8'), client_address)`.

O **arquivo dispatcher.py** define a classe *Dispatcher*, que é responsável por despachar as requisições do cliente para o *skeleton*. O método *dispatch\_request* recebe uma mensagem em formato JSON, que contém o nome do método e os argumentos. Ele usa a função `json.loads` para converter a mensagem em um objeto Python, e usa a função `getattr` para obter o método correspondente do *skeleton*. Ele então chama o método com os argumentos, e retorna o resultado em formato JSON.

Abaixo temos uma descrição detalhada do código:

### Classe *Dispatcher*:

- *\_\_init\_\_(self, skeleton)*: O construtor recebe um objeto *skeleton* como parâmetro e o armazena em *self.skeleton*.
- *dispatch\_request(self, payload)*: É um método que despacha uma requisição recebida para o método correspondente no objeto *Skeleton*.
  - Converte a carga útil (*payload*), que é uma mensagem JSON, em um dicionário Python usando `json.loads(payload)`. Esta mensagem JSON deve conter informações sobre o método a ser chamado e seus argumentos.

- Extrai o nome do método a ser chamado (*method\_name*) e os argumentos a serem passados para o método do dicionário extraído.
- Usa a função *getattr()* para obter o método correspondente no objeto *Skeleton* com base no *method\_name*.
- Chama o método obtido do esqueleto com os argumentos fornecidos e armazena o resultado em *result*.
- Converte o resultado de volta para um formato JSON usando *json.dumps(result)* para preparar a resposta que será enviada de volta ao solicitante.

O **arquivo *skeleton.py*** define a classe *Skeleton*, que é responsável por chamar os métodos do *servant*. O método *serve\_method* recebe o nome do método e os argumentos, e usa a função *getattr* para obter o método correspondente do *servante*. Ele então chama o método com os argumentos, e retorna o resultado em formato JSON. O *skeleton* também define os métodos *eat*, *play* e *sleep*, que são os mesmos nomes dos métodos do *servant*, e que chamam o método *serve\_method* com os mesmos argumentos.

Abaixo temos uma descrição detalhada do código:

### Classe *Skeleton*:

- *\_\_init\_\_(self, servant)*: O construtor recebe um objeto *servant* como parâmetro e o armazena em *self.servant*.
- *serve\_method(self, method\_name, \*args)*: É um método interno que recebe o nome de um método (*method\_name*) e seus argumentos (*\*args*) e chama esse método no objeto *servant*.
  - Usa a função *getattr()* para obter o método correspondente no objeto *servant* com base no *method\_name*.
  - Chama o método obtido no objeto *servant* com os argumentos fornecidos e armazena o resultado em *result*.
  - Converte o resultado para um formato JSON usando *json.dumps(result)* para preparar a resposta que será enviada de volta ao Dispatcher.
- Métodos de atalho: *eat*, *play*, *sleep*:
  - São métodos que chamam o método *serve\_method* passando o nome do método desejado e seus argumentos.
  - Cada um desses métodos específicos do *Skeleton* serve como uma interface para chamar métodos correspondentes no *Servant*, simplificando a chamada de métodos do *Servant* por meio do Dispatcher.

- Métodos de chamada direta: *eat*, *play*, *sleep*:
  - Cada um desses métodos (*eat*, *play*, *sleep*) representa um método específico que se deseja chamar no objeto *Servant*, e eles utilizam *serve\_method* para encaminhar a chamada para o *Servant* e obter os resultados de volta.

O **arquivo servant.py** define a classe *Servant*, que representa o objeto remoto que será invocado pelo cliente. O *servant* tem um atributo *pet*, que é uma instância da classe *Pet*, que representa um animal de estimação. O *servant* define os métodos *eat* e *play*, que recebem os argumentos *food* e *drink*, e *game* e *join*, respectivamente. Esses métodos chamam os métodos correspondentes do *pet*, que imprimem uma mensagem na tela e retornam um dicionário com o status da operação. O arquivo *servant.py* também define a classe *Pet*, que tem atributos como *alive*, *clean*, *energy* e *hungry*, que representam o estado do animal. Esses atributos ainda não foram implementados, mas servem para indicar uma possível continuação do código e explicita melhor o intuito do código. Abaixo temos uma descrição detalhada do código:

### Classe *Pet*:

- *\_\_init\_\_(self, energyMax, hungryMax, cleanMax)*: O construtor inicializa os atributos do *pet*, como níveis máximos de energia, fome e limpeza.
- *eat(self, food, drink)*: Método que simula alimentar o *pet* com comida e bebida.
  - Imprime uma mensagem indicando que o *pet* foi alimentado com comida e bebida.
  - Retorna um dicionário simples indicando o sucesso da operação de alimentação.
- *play(self, game, join)*: Método que simula o *pet* brincando com um jogo e uma opção para participar ou não.
  - Imprime uma mensagem indicando o jogo que foi jogado e se o proprietário do *pet* participou ou não.
  - Retorna um dicionário simples indicando o sucesso da operação de jogar.
- *sleep(self, hours, dream)*: Método que simula o *pet* dormindo por um número de horas, com ou sem sonhos.
  - Imprime uma mensagem indicando quantas horas o *pet* dormiu e se ele sonhou ou não.
  - Retorna um dicionário simples indicando o sucesso da operação de dormir.

### **Classe Servant:**

- `__init__(self)`: O construtor inicializa um objeto Pet dentro do Servant.
- Métodos *eat*, *play* e *sleep*: Cada um desses métodos no Servant atua como uma interface para chamar métodos correspondentes no objeto Pet.
  - Cada método no Servant chama diretamente o método correspondente no objeto Pet e retorna o resultado.

### **Conclusão**

Ao realizar esse trabalho, foi possível compreender melhor os processos que ocorrem quando um usuário, em um contexto cliente-servidor, faz alguma requisição remota. Isso se deu principalmente pela separação do código em diversos arquivos e a determinação de um conjunto de ações específicas para cada um. Além disso, esse projeto melhorou o entendimento de como é possível que clientes e servidores, num contexto de heterogeneidade das tecnologia, ainda sejam capazes de se comunicar.