# Computação Distribuída
# 2019 / 2020

Licenciatura em Engenharia Informática

Lab. 05 – Serviço de JSON-RPC

## Introdução

Nesta ficha iremos implementar um serviço básico de RPC. Este serviço será constituído por um servidor que disponibiliza o acesso remoto a um conjunto de funções e um cliente que acede às funções disponibilizadas pelo servidor.

Para as mensagens iremos usar uma versão mais simplificada do formato JSON-RPC 2.0 (http://www.jsonrpc.org/specification). O seguinte excerto exemplifica a utilização de JSON-RPC para a invocação da função *add(2, 3)*.

```
Cliente → Servidor
{“id”: 1, “jsonrpc”: “2.0”, “method”: “add”, “params”: [2, 3]}

Servidor → Cliente
{“id”: 1, “jsonrpc”: “2.0”, “result”: 5}
```

**Cliente:** O cliente deverá implementar uma interface para as funções disponibilizadas no servidor. Por exemplo, no cliente deverá eventualmente ser possível executar código semelhante ao seguinte:

```
val = add(2,3)
print(val)
```

A invocação da função *add* deverá originar o pedido para o servidor no formato JSON-RPC como descrito no excerto acima.

**Servidor:** Após iniciar o *socket*, o servidor deverá ficar à espera dos pedidos dos clientes. Para cada pedido o servidor deverá obter o nome da função e os respectivos argumentos. Por fim, deverá invocar a função pretendida e retornar o resultado para o cliente (no campo *result*).

Este laboratório tem por base o ficheiro **5-rpc-base.zip** que contém uma implementação inicial do servidor (*server.py*), do cliente (*client.py*) e das funções a disponibilizar pelo servidor (*functions.py*).

Crie um projecto com estes dois ficheiros e execute primeiro o servidor e depois o cliente. Verifique que o servidor faz o eco das mensagens enviadas pelo cliente.

## Nível 1 – Invocação de funções

Considere o código do servidor:

1. Importe o módulo *functions* (*import functions*).

2. Altere a resposta do servidor de modo a retornar o resultado da função *functions.hello()*.

3. Modifique a função *hello()* de modo a retornar 'Hi!'.

```
res = functions.hello()
conn.send(res.encode())
```

## Nível 2 – Nomes das funções

Considerando ainda o código do servidor:

1. No módulo *functions.py* crie uma nova função *hello2()* que deverá retornar 'Hello2'.

2. Modifique o servidor de modo a retornar o resultado da função *hello()* se receber do cliente a string "hello" ou o resultado da função *hello2()* se receber a string "hello2". Caso contrário deverá retornar uma mensagem de erro (função não existente).

```
if msg == 'hello':
    res = functions.hello()
elif msg == 'hello2':
    res = functions.hello2()
else:
    res = 'Error'
```

Coloque o servidor a funcionar e depois execute a aplicação cliente várias vezes modificando a mensagem enviada.

## Nível 3 – Parâmetros

Considere o seguinte excerto:

```
elif msg.startswith('greet'):
    values = msg.split()
    name = values[1]
    res = functions.greet(name)
```

1. Modifique o código do servidor de modo a receber o parâmetro *name* se o cliente invocar a função *greet*.

2. Teste o cliente invocando *client.send('greet Manuel')*.

Execute o servidor e o cliente, envie várias mensagens e verifique que o servidor invoca as funções correctamente até agora.

## Nível 4 – Múltiplos parâmetros

Considere a função *add(x, y)* e o seguinte excerto:

```
elif msg.startswith('add'):
    ..
    a = values[1]
    b = values[2]
    ..
```

1. Altere o código do servidor de modo a tratar correctamente a função *add* e seus parâmetros.

2. Modifique o necessário de forma a que a função *add* some inteiros e que a resposta a enviar pelo *socket* seja do tipo *string*.

Teste o cliente invocando *client.send('add 2 3')* e verifique que obtém a resposta *5*.

## Nível 5 – JSON

Considere o módulo *json* disponível em Python, e o seguinte excerto:

```
# Convert json message to dict
msg = json.loads(msg)
method = msg['method']
params = msg['params']
```

1. Altere o código do servidor de modo a converter o pedido do cliente em formato *json*. Modifique os *ifs* de modo a validar o nome da função usando a variável *method* e obtenha os parâmetros pela variável *params*.

2. Teste a função *add* com *client.send('{"method": "add", "params": [2, 3]}')*. O *json* é simplesmente uma string. Verifique que o cliente envia *json* bem formatado.

3. Teste as outras funções usando os seguintes *jsons* no cliente:
   - *{"method": "hello", "params": []}*
   - *{"method": "greet", "params": ["Manuel"]}*

4. Modifique a resposta do servidor de modo a retornar *json* com *{"result": ..}* ou *{"error": "Method not found"}*. Para evitar *jsons* inválidos (por causa do tipo de dados do resultado) considere criar um dicionário e utilize *json.dumps* para converter para *json*.

## Nível 6 – Parâmetros variáveis

Em Python é possível invocar funções passando uma lista com os argumentos:

```
> params = [2, 3]
> functions.add(*params)
5
```

Modifique a invocação de todas as funções implementadas no servidor de modo a que todas recebam o argumento **params*.

## Nível 7 – Funções como objectos

Em Python, as funções são objectos, e como tal podem ser mapeadas para outros objectos, ex:

```
> fn = functions.add
> type(fn)
<class ‘function’>

> fn(2, 3)
5

method = msg['method']
params = msg['params']
```

Considere então o seguinte código:

```
# Map functions
funcs = {
    'hello': functions.hello,
    'hello2': functions.hello2,
    'greet': functions.greet,
    'add': functions.add,
}

# Execute function
try:
    func = funcs[method]
    res = str(func(∗params))
except KeyError:
    res = 'Error'
```

1. Altere o código do servidor de modo a usar o mapeamento de objectos para associar os nomes do dicionário *funcs* às funções definidas do módulo *functions.py*.

2. Teste a invocação das várias funções no cliente e verifique que tudo continua a funcionar:
   - *{"method": "hello", "params": []}*
   - *{"method": "greet", "params": ["Manuel"]}*
   - *{"method": "add", "params": [2, 3]}*

3. Modifique o código do servidor de modo a que a variável *funcs* passe a ser um atributo da classe.

4. Crie um método *register(name, function)* de modo a associar um nome a uma função na variável *funcs.* Isto tornará possível o seguinte código no servidor.

```
server = JSONRPCServer('0.0.0.0', 8000)
server.register('add', functions.add)
server.start()
```

5. Registe e teste todas as funções implementadas até agora incluindo as funções *sub*, *mul* e *div* do módulo *functions.py.*

**Nível 8 – Cliente**

Considere o seguinte dicionário:

```
req = {
    'method': method,
    'params': params
}
```

1. No cliente, crie o método *invoke(method, params)* que recebe o nome da função e os parâmetros e envia em formato *json* para o servidor. O método deverá retornar o conteúdo do *result* ou *None* caso hajam erros.

2. Na classe do cliente crie o método *add(a, b)*. Este deverá utilizar o método *invoke* para comunicar com o servidor. No código, utilize *client.add(2, 3)* e deverá obter a resposta correcta.

3. Por fim, remova o método *add* e inclua o seguinte excerto de código. Este excerto faz uso de um mecanismo que invoca o método __getattr__ quando o interpretador de Python não encontro um método ou atributo de uma classe.

   ```
   def __getattr__(self, name):
       """Invokes a generic function."""
       def inner(*params):
           return self.invoke(name, params)
       return inner
   ```

   Invoque os métodos *client.add(..)* e *client.mul(..)* e verifique que continua tudo a funcionar correctamente.

4. Modifique o método *invoke* para lançar a excepção *AttributeError('Remote method unavailable')* quando existir erros, independente do que o servidor informar.

(fim de enunciado)