IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal

Curso :

Engenharia Informática

Ano Letivo 2017/2018

Docente: João Ventura

Relatório Técnico - Computação Distribuída

Relatório realizado por:

-Tomás Santos 160221032

-Tiago Neto 160221086

-Hugo Ferreira 160221089

Turma 1

Introdução

Foi-nos proposto elaboração de um modelo básico de RPC através do uso de sockets TCP/IP. Em suma, trata-se de um servidor que deve responder aos pedidos dos clientes executando as funções que tem conhecimento.

Funções opcionais de implementação

- Deteção de métodos inexistentes e Deteção de parâmetros inválidos

```
def getInformation(self,dicionary):
    method = dicionary["method"]
    if self.isValidMethod(method):
        if verifyLetter(dicionary) == False:
            return self.encode("Parametros Inválidos", dicionary["id"])
        x, y = verifyLetter(dicionary)
        return self.encode(self.methods[method](x, y), dicionary["id"])
    else:
        return self.encode("Metodo desconhecido",dicionary["id"])
```

```
def verifyLetter(decoded):
    x = decoded["params"]["x"]
    y = decoded["params"]["y"]

    try:
        value = int(x)
        value2=int(y)
        return value,value2
    except ValueError:
        return False
```

- Mecanismos para “registo” de funções no servidor

```
def register(self,function):
    self.methods[function.__name__]=function

class RPCServer:
    def __init__(self):
        self.methods={}
        self.port=8000
        self.host='0.0.0.0'
```

```
rpcserver=rpcserver.RPCServer()
rpcserver.register(myLib.add)
rpcserver.register(myLib.div)
rpcserver.register(myLib.mul)
rpcserver.register(myLib.sub)
rpcserver.start()
```

- Várias invocações de funções num único pedido

```
def decode(self,information):
    decoded = json.loads(information)
    if isinstance(decoded, (list,)):
        newList = []
        for item in decoded:
            newList.append(self.singleDecode(item))
        return newList
    else:
        return self.singleDecode(decoded)
```

```
list= ['mul(1, a)', 'sub(2,4)', 'ads(10,2)', 'mul(4,2)']
x=client2.sdf(list)
print(x)
```

- Implementação de notificações json-rpc

```
def notificate(self,content):
    dicionary={
        "jsonrpc": "2.0",
        "method": content
    }
    self.client_socket.send(json.dumps(dicionary).encode())
```

- Reutilização de ligações tcp/ip

```
def listenClient(self,client_connection):
    while True:
        # Print message from client
        msg = client_connection.recv(1024).decode()
        print('Received:', msg)
        try:
            result = self.decode(msg)
            if result != "Exit":
                client_connection.send(result.encode())
            else:
                break
        except:
            client_connection.send(self.notificate("Erro!").encode())
    client_connection.close()
```

- Múltiplos clientes em paralelo

```
def start(self):
    # Create socket
    server_socket=self.createServerSocket()
    print('Listening on port %s ...' % self.port)

    try:
        while True:
            # Wait for client connections
            client_connection, client_address = server_socket.accept()
            client_connection.settimeout(10)
            _thread.start_new_thread(self.listenClient, (client_connection,))
    except timeout:
        # Close socket
        server_socket.close()
        print('Servidor encerrado!')
```

- Outras
    - Utilizámos um módulo Library onde são armazenadas todas as funções, utilizando, uma função register que regista as mesmas no servidor, organizadas numa lista.
    - No cliente, não houve necessidade de implementar nenhuma função. Utilizando o método getattr, criámos uma função que através de um conjunto de parâmetros recebidos, neste caso, dois valores ou uma lista, procede ao envio do nome da função e dos argumentos para o servidor.

```
def add(x, y):
    return x + y

def sub(x, y):
    return x - y

def div(x, y):
    return x / y

def mul(x, y):
    return x * y
```

```
def __getattr__(self, item):
    def calc(*args):
        if self.online == True:
            if len(args) == 1 and isinstance(args[0], (list,)):
                info = self.encode(args[0])
            elif len(args) == 1:
                info = self.encode("%s,%s,%s" % (item, args[0],0))
            else:
                info = self.encode("%s,%s,%s" % (item, args[0], args[1])
            return self.sendReceive(info)
        else:
            return "Servidor offline!"
    return calc
```

Dificuldades

A maiores dificuldades passaram por compreender o funcionamento do mecanismo json rpc, funções não estarem definidas no cliente e mesmo assim serem enviadas para o servidor (método getattr) e trabalhar com as threads em geral.

Gráficos Ilustrativos de funcionamento

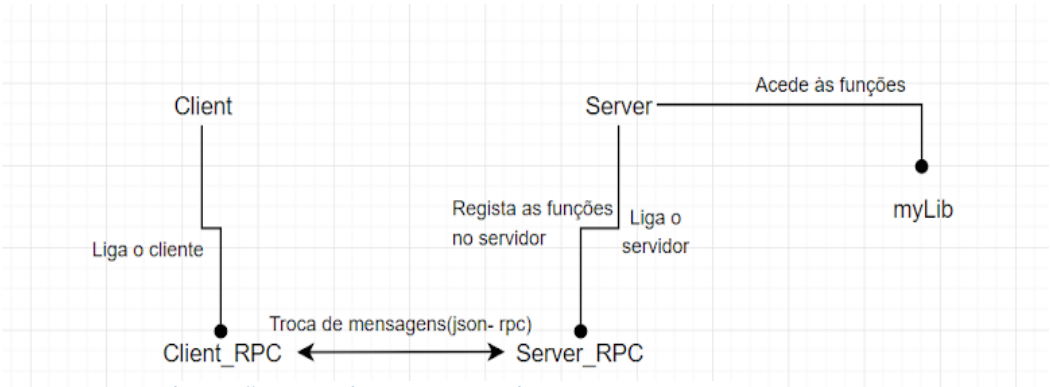

Ilustração 1 Funcionamento geral

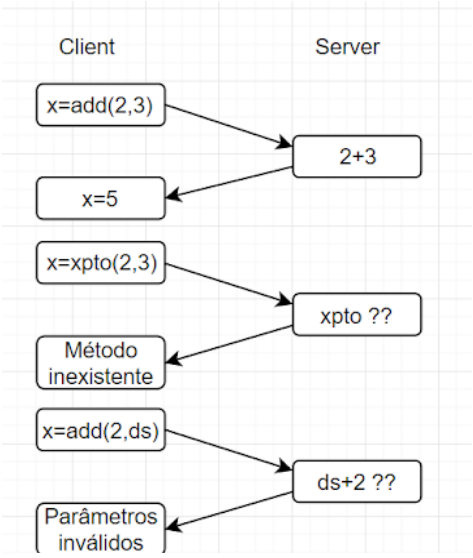

Ilustração 2 Troca de mensagens

Melhorias

Deteção de métodos inexistentes e deteção de parâmetros inválidos implementados com o formato de erro padrão do Json-RPC

```
def createError(self,code,message,id):
    error={
        "jsonrpc": "2.0",
        "error": {
            "code":code,
            "message":message
        },
        "id":id
    }
    return error
```