Ciências ULisboa

Aplicações Distribuídas
2022/23

Guião de apoio 3
Organização de Clientes e Servidores: RPC

---

1. Introdução ao tema

A concretização estruturada de um sistema cliente-servidor envolve a definição e construção de uma série de módulos ou componentes que devem interagir para que a aplicação distribuída funcione. Nesta aula vamos realizar alguns exercícios relacionados com a definição destes módulos e sua concretização.

Conforme visto nas aulas TP, uma abordagem muito comum para simplificar o acesso a recursos remotos (e.g., servidores) é o uso de RPCs (*Remote Procedure Calls*). Nesta abordagem, a aplicação cliente acede a um serviço remoto usando uma interface muito parecida com a que seria usada se o serviço fosse local. Assim, todas as trocas de mensagens e gestão de ligações e *sockets* são escondidas dentro de componentes responsáveis pela comunicação em baixo nível.

Esta interação está exemplificada na Figura 1. Nela, um conjunto de clientes interagem com um servidor que mantém uma estrutura de dados (e.g., um dicionário ou uma lista). Cada cliente é composto pelo código da aplicação e um *stub* que contém todo o código para o acesso à estrutura de dados remota. Do lado do servidor existe um programa principal "Servidor" que recebe mensagens e as envia a um *skeleton* que interpreta as mensagens, invoca a operação pedida na estrutura de dados e responde apropriadamente.

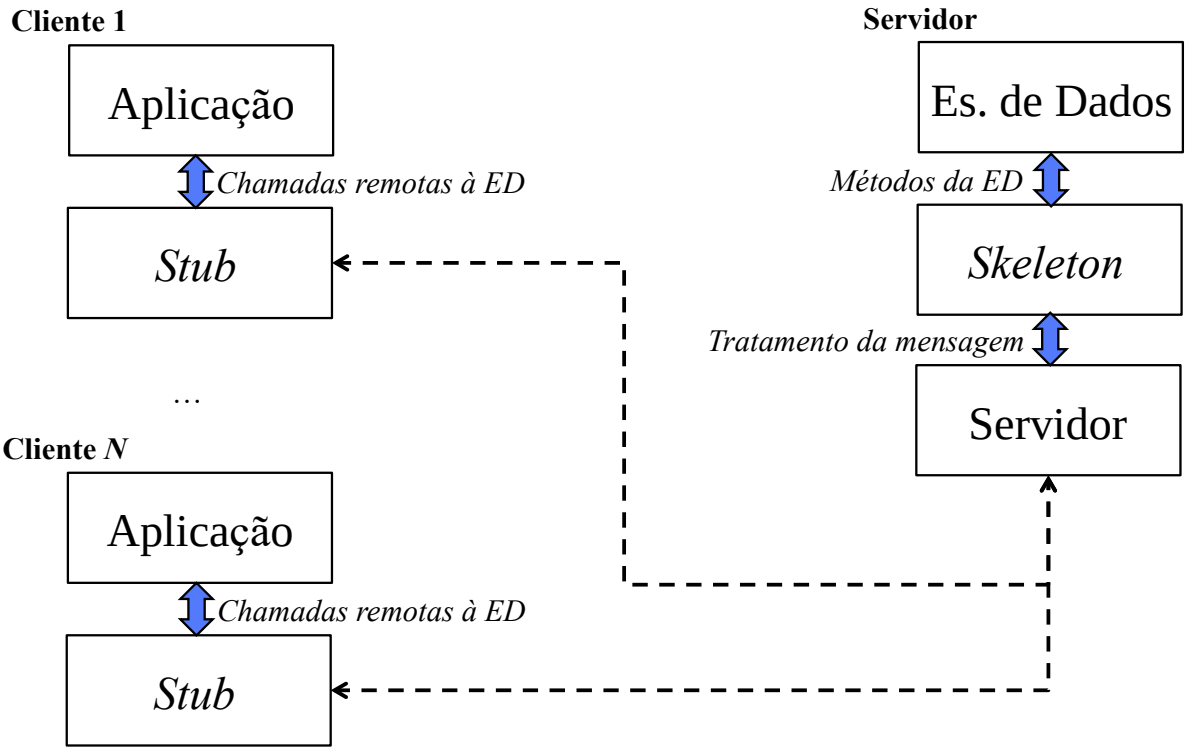

Figura 1. Arquitetura de um sistema cliente/servidor com RPC.

Nas secções seguintes, detalhamos algumas responsabilidades dos módulos *Stub* e *Skeleton*, bem como a sua ligação com a serialização e desserialização de mensagens.

2. Módulos *Stub* e *Skeleton*

2.1. *Stub*

Uma classe *stub* contém sempre um conjunto de métodos para ligar-se a um servidor, desligar-se de um servidor ao qual o *stub* tenha sido previamente ligado e invocar as operações suportadas pelo servidor. Para além disso, toda a gestão de erros relativos à utilização de sockets deve ser tratada nessa classe.

Como exemplo, considere um servidor que implementa uma lista (como a construída nos guiões anteriores). Neste sistema, a interface de um *stub* para o "serviço de lista" terá uma interface da seguinte forma.

```
class ListStub:

      def __init__(self):
            self.conn_sock = None

      def connect(self, host, port):
            # código para estabelecer uma ligação,
            # i.e., tornando self.conn_sock válida

      def disconnect(self):
            # Fecha a ligação conn_sock

      # Métodos tradicionais de um objeto do tipo lista
      def append(self, element):

      def list(self):

      def clear(self):

      # Outros métodos possíveis
```

Uma vez que o método *connect* trata da criação e ligação da socket, os métodos do serviço (e.g., *append*) têm apenas algumas poucas tarefas a fazer (ver slides das TPs para mais detalhes):

1) verifica se o socket é válido;
2) criar uma mensagem correspondente à operação pedida pela aplicação
3) enviar essa mensagem usando o socket (e.g., através do *sendall*)
4) esperar a resposta usando o socket (e.g., através do *recv*)
5) converter a resposta recebida num resultado a ser retornado pelo método
6) retornar o resultado

Além disso, estes métodos devem estar preparados para lidar com erros na ligação e fechar a socket, sinalizando a condição inesperada para a aplicação. Isto é feito através de cláusulas *try/except* e o lançamento de uma exceção, por exemplo *InvalidCall*, em caso de problemas.

2.2. *Skeleton*

Diferentemente do *stub* do lado do cliente, onde temos múltiplos métodos para tratar das diferentes operações suportadas pelo servidor, um *skeleton* normalmente consiste de um método principal (e alguns auxiliares, quando necessário). Este método compreende toda a lógica de programação necessária para a interpretação das mensagens com os pedidos dos clientes, a invocação dos métodos requeridos no serviço local ao servidor e a construção de uma mensagem a ser enviada em resposta ao cliente. A seguir apresentamos um *Skeleton* que trata de mensagens codificadas como listas, com o primeiro elemento como o nome do método e os seguintes com os parâmetros.

```
class ListSkeleton:

      def __init__(self):
            self.servicoLista = [];

      def processMessage(self, msg_bytes) :
            pedido = bytesToList(msg_bytes)

            resposta = [];

            if pedido == None or len(pedido) == 0 :
                  resposta.append('INVALID MESSAGE')
            else :
                  if pedido[0] == 'append' and len(pedido) > 1 :
                              self.servicoLista.append(pedido[1])
                              resposta.append('OK')
                  elif pedido[0] == 'list' :
                        resposta.append(str(self.servicoLista))
                  elif pedido[0] == 'clear' :
                        self.servicoLista = []
                        resposta.append('OK')
                  else :
                        resposta.append('INVALID MESSAGE')

            return listToBytes(resposta)

      #fim do metodo processMessage

      #outros métodos possíveis
```

Existem pelo menos três observações importantes a serem feitas sobre a concretização do método *processMessage*. Em primeiro lugar, este método faz uso de métodos auxiliares para converter os bytes recebidos para o formato das mensagens (neste caso, uma lista) e vice-versa. Esses métodos devem tratar das tarefas de serialização e deserialização, vistas na aula anterior. Além disso, é importante observar que este método deve ser capaz de verificar todos os erros possíveis das mensagens, como por exemplo a invocação de uma operação não suportada ou mensagens com parâmetros insuficientes. Finalmente, é importante observar que o número de 'ifs' concretizados neste método depende do número de operações oferecidas aos clientes, e que em cada um desses 'ifs' temos a invocação do método requisitado no serviço local (neste caso, a lista).

3. Exercícios fundamentais

1. Copie os programas cliente e servidor apresentados a seguir para sua área e execute-os no computador.

Cliente (Cliente.py)

```
import sys, socket as s

if len(sys.argv) > 1:
    HOST = sys.argv[1]
    PORT = int(sys.argv[2])
else:
    HOST = '127.0.0.1'
    PORT = 9999

while True:
    msg = input('Mensagem: ');
    if msg == 'EXIT':
        exit()

    sock = s.socket(s.AF_INET, s.SOCK_STREAM)
    sock.connect((HOST, PORT))

    sock.sendall(msg.encode())
    resposta = sock.recv(1024)

    print ('Recebi: %s' % resposta.decode())
    sock.close()
```

Servidor (Servidor.py)

```
import sys, socket as s
HOST = ''
if len(sys.argv) > 1:
    PORT = int(sys.argv[1])
else:
    PORT = 9999
sock = s.socket(s.AF_INET, s.SOCK_STREAM)
sock.setsockopt(s.SOL_SOCKET, s.SO_REUSEADDR, 1)

sock.bind((HOST, PORT))
sock.listen(1)

list = []
while True:
    try:
        (conn_sock, addr) = sock.accept()
        msg = conn_sock.recv(1024)
        resp = 'Ack'

        if msg.decode() == 'LIST':
            resp = str(list)
        elif msg.decode() == 'CLEAR':
            list = []
            resp = "Lista apagada"
        else:
            list.append(msg.decode())

        conn_sock.sendall(resp.encode())

        print ('list= %s' % list)
        conn_sock.close()
    except:
        print ('socket fechado!')
        conn_sock.close()

sock.close()
```

2. Modifique os programas anteriores para que eles sejam organizados como na Figura 1 (ver Secção 1 deste documento). Em particular, crie as classes *Stub* e *Skeleton* em ficheiros separados.

3. Modifique os programas anteriores para que as mensagens sejam agora enviadas como listas, serializadas através do *pickle*.

4. Modifique os programas anteriores para suportar as seguintes novas operações na lista: *size*, *remove(element)*, *remove_all(element)* e *pop*.

---

3. Bibliografia e outro material de apoio

https://docs.python.org/3/library/pickle.html
https://docs.python.org/3/tutorial/introduction.html#lists
https://docs.python.org/3/library/struct.html