YA-RPC: Yet-Another RPC Framework 设计报告

1 引言

1.1 编写目的

通过学院开设的分布式系统课程学习了分布式系统的本原理，了解了远程过程调用(RPC: Remote Procedure Call)的基本概念，此次课程大作业选择的题目为“YA-RPC: Yet-Another RPC Framework”，此设计报告文档旨在说明实现该题目的整个过程，包括相关原理、项目体系结构、重点代码、测试结果，以便使读者理解整个系统的结构和功能。

1.2 背景知识

RPC 由三部分组成：RPC Server、RPC Client、Registry。RPC Server 作为服务提供方，暴露服务；RPC Client 作为服务消费方，调用远程服务；Registry 实现服务的注册与发现。这三部分协作实现 RPC 的简要过程为：RPC Server 在 Registry 中注册服务，RPC Client 在 Registry 中订阅服务，Registry 将服务的信息传给 RPC Client，最终 RPC Client 调用 RPC Server 提供的服务。

1.3 说明

分布式系统大作业选题：YA-RPC: Yet-Another RPC Framework

开发语言：Python

开发环境：PyCharm，Anaconda

设计与开发者：邓棋（202222080416）

2 系统设计与实现

2.1 需求分析

开发的 YA-RPC 框架需支持三种基本数据类型：int, float, string；支持 At-least-once 语义；需要实现两个 API： float sum(float a, float b)，string uppercase(str)；并不少于 2 个客户端，1 个服务端。

由此可分析得到该系统的非功能性需求有：

（1） 访问透明性：客户端调用函数时就像调用本地函数一样。

（2） 并发要求：至少实现并发数为 2 的并发调用。

2.2 概要设计

要实现 RPC 需要解决 3 个主要问题：

（1）Call ID 映射。如何确定客户端到底调用的是哪个函数？在本地调用中，函数体是直接通过函数指针来指定的，但是在远程调用中，函数指针是不行的，因为两个进程的地址空间是完全不一样的。所以，在 RPC 中所有的函数都必须有自己的一个 ID，这个 ID 在所有进程中都是唯一确定的。客户端在做远程过程调用时，必须附上这个 ID。然后我们还需要在客户端和服务端分别维护函数与 Call ID 的对应表。当客户端需要进行远程调用时，它就查一下这个表，找出相应的 Call ID，然后把它传给服务端，服务端也通过查表，来确定客户端需要调用的函数，然后执行相应函数的代码。

（2）序列化和反序列化。客户端怎么把参数值传给远程的函数呢？在本地调用中，我们只需要把参数压到栈里，然后让函数自己去栈里读就行。但是在远程过程调用时，客户端跟服务端是不同的进程，不能通过内存来传递参数。这时候就需要客户端把参数先转成一个字节流，传给服务端后，再把字节流转成自己能读取的格式。这个过程叫序列化和反序列化。同理，从服务端返回的值也需要序列化反序列化的过程。

（3）网络传输。客户端和服务端通过网络连接的，所有的数据都需要通过网络传输，因此需要有一个网络传输层把 Call ID 和序列化后的参数字节流传给服务端，然后再把序列化后的调用结果传回客户端。

因此，实现 RPC 的基本工作原理为：部署在不同服务器上的客户端想调用服务端提供的服务，由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。因此需要客户端把参数转换成字节流，传给服务端，然后服务端将字节流转换成自身能读取的格式，这是一个序列化和反序列化的过程；数据准备好了之后，通过网络传输层把序列化后的参数传给服务端，然后把计算好的结果序列化传给客户端。该过程如图 1 所示。

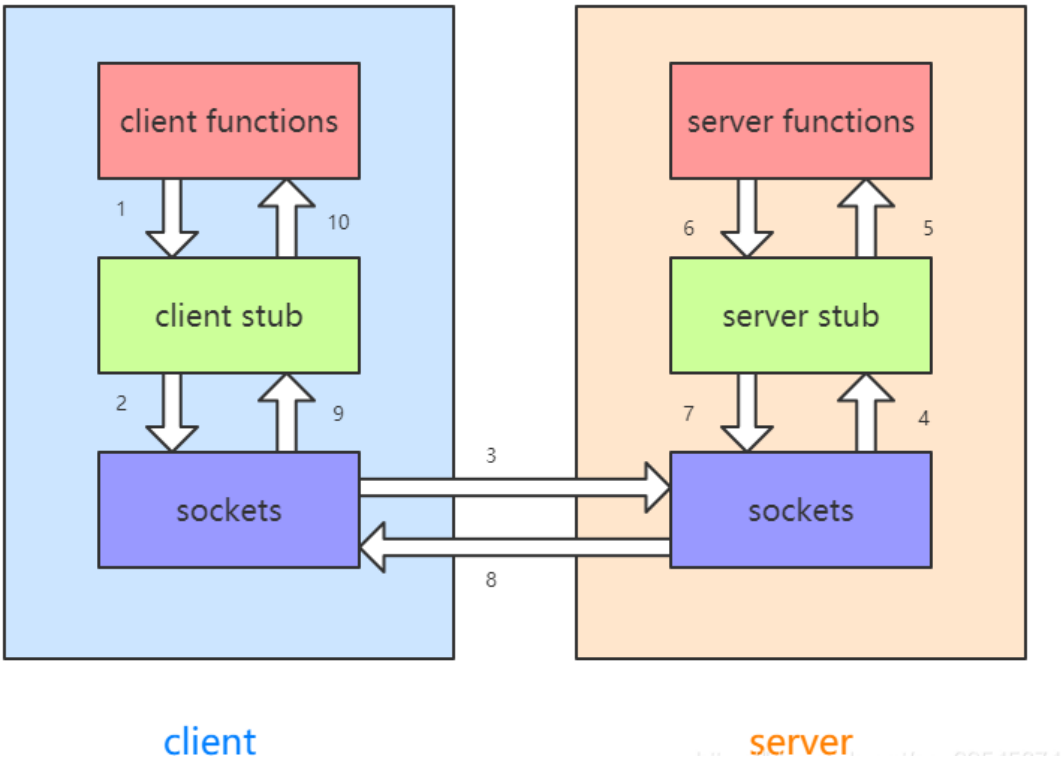

图 1 RPC 调用过程

2.3 详细设计

本课程设计采用 Python 语言实现 YA-RPC 框架，使用 socket、struct、json 这三个 Python 内置库分别实现 RPC 服务的网络通信功能、字节转换功能和消息序列化功能。

网络通信的内容是字节序列，消息序列化的目标是将 Python 的数据结构转换成字节序列，而用于界定消息边界的信息长度也是消息的一部分，它需要将 Python 的整型数组转换成字节数组，这部分工作由 struct 库来完成。

Socket 库是 Python 内置的网络编程类库，方便用户编写 Tcp/Udp 相关代码。客户端和服务端进程需要通信时，可以通过 socket 套接字来传递数据。Struct 库是 Python 内置的二进制解码编码库，用于将各种不同的类型的字段编码成二进制字节串，通过 struct 包将消息的长度整数编码成 byte 数组。Json 库是 Python 内置的序列化库，它用 dumps 方法将内存的对象序列化成 json 字符串，用 loads 方法将字符串反序列化成 Python 对象。

消息协议：使用长度前缀法来确定消息边界，消息体使用 json 序列化。每个消息都有相应的名称，请求的名称使用 in 字段表示，请求的参数使用 params 字段表示，响应的名称使用 out 字段表示，响应的结果使用 result 字段表示。将

请求和响应使用 json 序列化成字符串作为消息体，然后通过 Python 内置的 struct 包将消息体的长度整数转成 4 个字节的长度前缀字符串。

系统调用过程的流程图如图 2 所示。

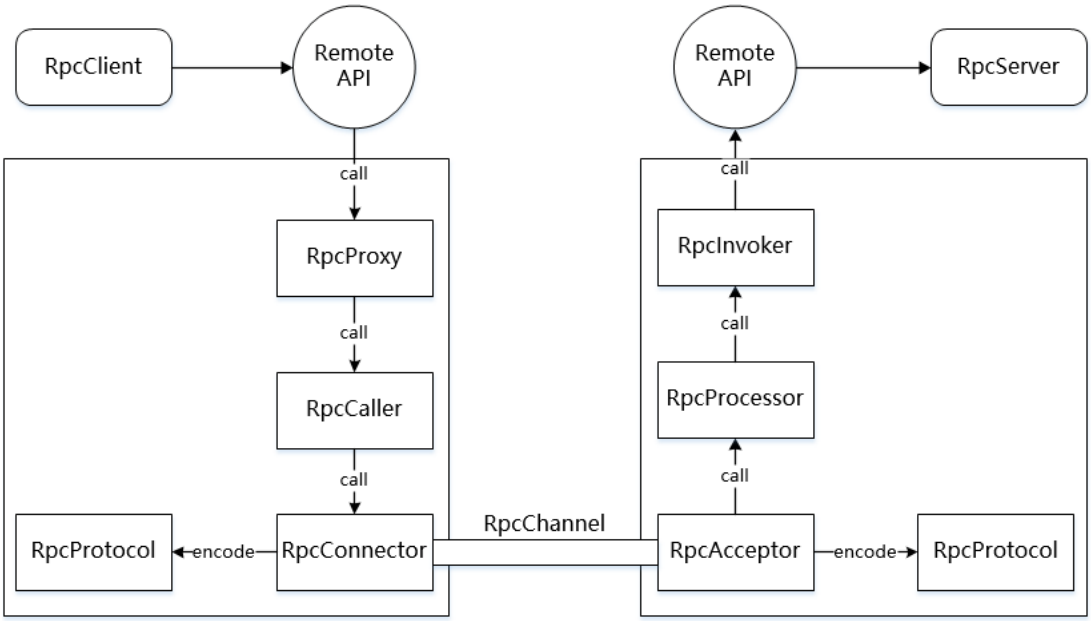

图 2 RPC 调用流程图

2.4 系统实现与测试

2.4.1 sum 函数实现与测试

sum 函数获取两个参数后根据变量的类型进行计算，并返回结果。

```
    def sum(conn, params, args):
        try:
            # 以逗号分隔获取两个加数
            a, b = str(params).split(',')
            if args == 'float':
                # 保留 6 位小数
                res = round(float(a) + float(b), 6)
            elif args == 'int':
                res = int(a) + int(b)
            send_result(conn, 'sum_result', res)
        except:
            send_result(conn, 'error', 'error')
```

客户端请求调用 sum 函数的代码如下。

```
    print('测试 sum')
    out, result = rpc(s, 'sum', '6,6', 'int')
    out, result = rpc(s, 'sum', '7,8.8888888888', 'float')
    out, result = rpc(s, 'sum', '3,4', 'int')
    out, result = rpc(s, 'sum', '2,9.12345678', 'float')
```

客户端控制台结果如图 3.1 所示，服务端控制台结果如图 3.2 所示。

```
测试sum
send data: b'{"in": "sum", "params": "6,6", "args": "int"}'
receive: b'{"out": "sum_result", "result": 12}'
send data: b'{"in": "sum", "params": "7,8.8888888888", "args": "float"}'
receive: b'{"out": "sum_result", "result": 15.888889}'
send data: b'{"in": "sum", "params": "3,4", "args": "int"}'
receive: b'{"out": "sum_result", "result": 7}'
send data: b'{"in": "sum", "params": "2,9.12345678", "args": "float"}'
receive: b'{"out": "sum_result", "result": 11.123457}'
```

图 3.1 sum 函数功能测试，客户端控制台

```
listen
client: ('127.0.0.1', 53779) connect
recv : {'in': 'sum', 'params': '6,6', 'args': 'int'}
recv : {'in': 'sum', 'params': '7,8.8888888888', 'args': 'float'}
recv : {'in': 'sum', 'params': '3,4', 'args': 'int'}
recv : {'in': 'sum', 'params': '2,9.12345678', 'args': 'float'}
client: ('127.0.0.1', 53779) close
```

图 3.2 sum 函数功能测试，服务端控制台

2.4.2 uppercase 函数实现与测试

uppercase 函数直接使用 str.upper()方法即可实现该函数功能。

```
1.     def uppercase(conn, params, args):
2.         # print(str.upper(params))
3.         res = str.upper(params)
4.         send_result(conn, 'upper_result', res)
```

客户端请求调用 uppercase 函数的代码如下。

```
1.     print('测试 Uppercase')
2.     out, result = rpc(s, 'uppercase', 'dengqi', 'str')
3.     out, result = rpc(s, 'uppercase', 'ya-rpc', 'str')
```

客户端控制台结果如图 4.1 所示，服务端控制台结果如图 4.2 所示。

```
测试Uppercase
send data: b'{"in": "uppercase", "params": "dengqi", "args": "str"}'
receive: b'{"out": "upper_result", "result": "DENGQI"}'
send data: b'{"in": "uppercase", "params": "ya-rpc", "args": "str"}'
receive: b'{"out": "upper_result", "result": "YA-RPC"}'
```

图 4.1 uppercase 函数功能测试，客户端控制台

```
listen
client: ('127.0.0.1', 53845) connect
recv : {'in': 'uppercase', 'params': 'dengqi', 'args': 'str'}
recv : {'in': 'uppercase', 'params': 'ya-rpc', 'args': 'str'}
client: ('127.0.0.1', 53845) close
```

图 4.2 uppercase 函数功能测试，服务端控制台

2.4.3 At-least-once 实现与测试

实现部分代码如下。

```
    # 等待接受响应
        try:
            # 设置响应时间以实现 At-least-once 语义
            sock.settimeout(1)
            # 接受响应并且得到响体
            length_prefix = sock.recv(4)
            length, = struct.unpack('I', length_prefix)
            # 响应消息体
            body = sock.recv(length)
            print('receive:', body)
            response = json.loads(body)
            # 返回响应类型和结果
            return response['out'], response['result']
        # 响应时间超过时、重新发送请求
        except Exception as e:
            print(e)
            return rpc(sock, in_, params, args)
```

将客户端的 timeout 设置为 1s，服务端 1.5s 后才发送响应，此时客户端超时应重新发送请求。

客户端控制台结果如图 5.1 所示，服务端控制台结果如图 5.2 所示。

```
测试At-least-once语义
send data: b'{"in": "uppercase", "params": "test the at-least-once", "args": "str"}'
timed out
send data: b'{"in": "uppercase", "params": "test the at-least-once", "args": "str"}'
receive: b'{"out": "upper_result", "result": "TEST THE AT-LEAST-ONCE"}'
```

图 5.1 At-least-once 语义功能测试，客户端控制台

```
listen
client: ('127.0.0.1', 53912) connect
recv : {'in': 'uppercase', 'params': 'test the at-least-once', 'args': 'str'}
recv : {'in': 'uppercase', 'params': 'test the at-least-once', 'args': 'str'}
client: ('127.0.0.1', 53912) close
```

图 5.2 At-least-once 语义功能测试，服务端控制台

2.4.4 并发实现与测试

设置两个客户端同时对服务端发起请求，测试请求代码与上文类似，服务端的控制台输出结果如图 6 所示。

```
listen
client: ('127.0.0.1', 53935) connect
recv : {'in': 'sum', 'params': '1,2', 'args': 'int'}
recv : {'in': 'sum', 'params': '6.666,7.777', 'args': 'float'}
client: ('127.0.0.1', 53935) close
client: ('127.0.0.1', 53936) connect
recv : {'in': 'sum', 'params': '6,6', 'args': 'int'}
recv : {'in': 'sum', 'params': '7,8.8888888888', 'args': 'float'}
recv : {'in': 'sum', 'params': '3,4', 'args': 'int'}
recv : {'in': 'sum', 'params': '2,9.12345678', 'args': 'float'}
recv : {'in': 'uppercase', 'params': 'dengqi', 'args': 'str'}
recv : {'in': 'uppercase', 'params': 'ya-rpc', 'args': 'str'}
client: ('127.0.0.1', 53936) close
```

图 6 并发测试，服务端控制台

3 总结

通过本次大作业实现的 YA-RPC 框架，加深了自己对 RPC 的理解，也对分布式系统有了更加直观的体验。目前国内使用较多的分布式方案是 Spring Cloud Alibaba，如果要设计大型的分布式项目，注册中心和配置中心可以使用其中的 Nacos 组件，调用远程服务可以使用 Feign 组件，包括调用链监控的 Sleuth 组件，分布式事务解决方案 Seata 组件等等。现实生活中软件的并发请求越来越大，应用分布式具有广阔的前景。