分布式系统作业

第5次作业

姓名：郝裕玮

班级：计科1班

学号：18329015

理论题：

1、请描述一个用于显示刚更新的Web页面的写读一致性

的简单实现。

答：让浏览器发送一个请求到Web服务器来检查它所显示的页面是否为最新版本。

2、使用Lamport逻辑时钟的全序多播不能扩展的原因。

答：Lamport 的全序多播方式要求所有服务器都启动并运行，而当其中一个服务器运行缓慢或崩溃时，则会降低其性能。这必须被所有其他服务器检测到。

同时随着服务器数量的增加，这个问题会越来越严重。

实验题：

3、请实现一个支持多播的RPC的简单系统。假设系统有多个复制的服务器，每个客户可以通过RPC与一个服务器通信。但是处理复制时，客户需要向每一个副本发送一个 RPC请求。设计客户程序，使得客户好像只往应用程序发送单个RPC。假设复制的目的是为了提高性能，而那些服务器可能容易出故障（程序实现）。

一、解决方案

（1）实验环境：Windows10

（2）在cmd窗口中执行以下代码，配置环境：

pip install grpcio

pip install protobuf

pip install grpcio-tools

（3）桌面新建proto文件夹并在文件夹中新建msg.proto，文件内容如下（具体分析已全部包含在代码注释中）：

syntax = "proto3";

//规定语法，这里使用的是proto3的语法

//使用service关键字定义服务

service MsgService {

    rpc GetMsg (MsgRequest) returns (MsgResponse){}

    //简单RPC，即客户端发送一个请求给服务端，从服务端获取一个应答，就像一次普通的函数调用

}

//定义message内部需要传递的数据类型

message MsgRequest {

    //消息定义中，每个字段都有唯一的一个数字标识符

    //这些标识符是用来在消息的二进制格式中识别各个字段的，一旦开始使用就不能够再改变

    string text = 1;//发送内容

}

message MsgResponse {

    int32 id = 1;//服务器id

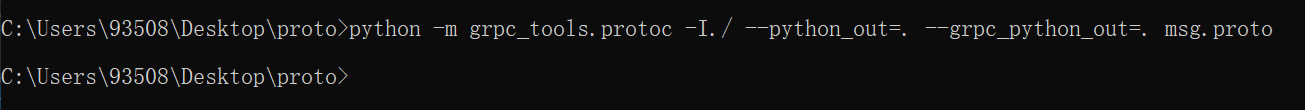
    string result = 2; //返回结果

}

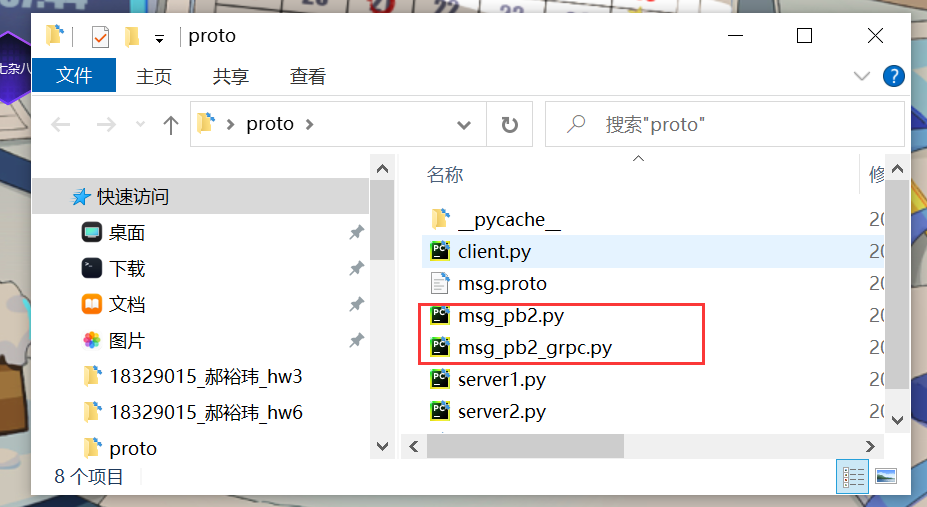
（4）在proto文件夹中打开cmd窗口，执行如下代码对msg.proto进行编译运行：

python -m grpc\_tools.protoc -I./ --python\_out=. --grpc\_python\_out=.

msg.proto



（5）编译运行后发现proto文件夹中多出两个文件：



这两个文件是为后续的客户端和服务器端所用（包含了许多可调用的变量，类和函数）。

（6）在proto文件夹中新建client.py，server1.py，server2.py，server3.py，其中server(1-3)三个文件的代码几乎一致，只需要修改监听端口即可（会在下文说明），所有代码具体如下（具体思路和详细分析均已包含在代码注释中）：

在展示代码前，我先解释下我的实现思路。

本次的重点为多播RPC，需要具有复制和容错（retry）功能。

所以我参考了之前的hw3，并对代码进行了一定程度上的修改。

我设置了3个服务器和1个客户端，运行时，我只需要在客户端输入1次信息，即可向3个服务器广播我的信息。同时我的客户端也会对3个服务器返回的内容进行比较，若不同。则会向返回错误信息的服务器重发n次源信息。

具体思路可见以下代码：

对于client.py：

#coding=gbk

from \_\_future\_\_ import print\_function

import grpc

import msg\_pb2

import msg\_pb2\_grpc

import copy

def run():

    times = 1 #记录重发次数

    # 客户端很好理解,网络连接得到一个channel,拿channel去实例化一个stub,通过stub调用RPC函数

    # 这里连接3个服务器，所以需要3个channel

    channel1 = grpc.insecure\_channel('localhost:50051')

    channel2 = grpc.insecure\_channel('localhost:50052')

    channel3 = grpc.insecure\_channel('localhost:50053')

    # 使用grpc.insecure\_channel('localhost:50051')进行连接服务端,接着在这个channel上创建stub

    # 同样分别在3个服务器上创建不同的stub

    stub1 = msg\_pb2\_grpc.MsgServiceStub(channel1)

    stub2 = msg\_pb2\_grpc.MsgServiceStub(channel2)

    stub3 = msg\_pb2\_grpc.MsgServiceStub(channel3)

    # 在msg\_pb2\_grpc里可以找到MsgServiceStub这个类相关信息。这个stub可以调用远程的GetMsg函数

    text1 = input() #输入发送给3个服务器的内容

    #复制内容,便于发往服务器2,3

    text2 = copy.deepcopy(text1)

    text3 = copy.deepcopy(text1)

    #用于验证重发功能

    text1 = "Different!"

    #text2 = "Different!"

    #text3 = "Different!"

    print('\n')

    # response为服务器端发来的内容

    # MsgRequest中的内容即msg.proto中定义的数据。在回应里可以得到msg.proto中定义的msg

    response1 = stub1.GetMsg(msg\_pb2.MsgRequest(text = text1))

    response2 = stub2.GetMsg(msg\_pb2.MsgRequest(text = text2))

    response3 = stub3.GetMsg(msg\_pb2.MsgRequest(text = text3))

    # 比较从3个服务器收到的结果是否一致，若不一致则将不一致的消息进行重发

    # 服务器1出错的情况

    if response2.result == response3.result and response1.result != response3.result:

        #客户端重发10次

        while times<=10:

            #对服务器1进行重发

            response1 = stub1.GetMsg(msg\_pb2.MsgRequest(text = text1))

            #打印重发后服务器1返回的信息

            print("客户端第{}次重发，服务器{}返回的消息为: {}\n".format(times,response1.id,response1.result))

            #若某次重发后结果与其他两个服务器返回一致则不再重发

            if response1.result == response3.result:

                break

            else:#反之times+1,继续重发

                times+=1

    #服务器2出错的情况

    elif response1.result == response3.result and response2.result != response3.result:

        while times<=10:

            response2 = stub2.GetMsg(msg\_pb2.MsgRequest(text = text2))

            print("客户端第{}次重发，服务器{}返回的消息为: {}\n".format(times,response2.id,response2.result))

            if response2.result == response3.result:

                break

            else:

                times+=1

    #服务器3出错的情况

    elif response1.result == response2.result and response3.result != response1.result:

        while times<=10:

            response3 = stub3.GetMsg(msg\_pb2.MsgRequest(text = text3))

            print("客户端第{}次重发，服务器{}返回的消息为: {}\n".format(times,response3.id,response3.result))

            if response3.result == response1.result:

                break

            else:

                times+=1

    #服务器1,2,3均出错的情况

    elif response1.result != response2.result and response2.result != response3.result and response1.result != response3.result:

        while times<=10:

            response1 = stub1.GetMsg(msg\_pb2.MsgRequest(text = text1))

            print("客户端第{}次重发，服务器{}返回的消息为: {}\n".format(times,response1.id,response1.result))

            response2 = stub2.GetMsg(msg\_pb2.MsgRequest(text = text2))

            print("客户端第{}次重发，服务器{}返回的消息为: {}\n".format(times,response2.id,response2.result))

            response3 = stub3.GetMsg(msg\_pb2.MsgRequest(text = text3))

            print("客户端第{}次重发，服务器{}返回的消息为: {}\n".format(times,response3.id,response3.result))

            if response1.result == response2.result and response2.result == response3.result:

                break

            else:

                times+=1

    #重发结束后

    if response1.result == response2.result and response2.result == response3.result:

        if times == 1:

            print("无需重发,3个服务器返回结果一致!\n")

        else:

            print("重发{}次后,3个服务器返回结果一致!\n".format(times-1))

    else:

        print("重发{}次后,3个服务器返回结果仍不一致!\n".format(times-1))

    print("最终,服务器{}返回的消息为: {}\n".format(response1.id,response1.result))

    print("最终,服务器{}返回的消息为: {}\n".format(response2.id,response2.result))

    print("最终,服务器{}返回的消息为: {}\n".format(response3.id,response3.result))

if \_\_name\_\_ == '\_\_main\_\_':

    run()

对于server1.py:

#coding=gbk

from concurrent import futures

import time

import grpc

import msg\_pb2

import msg\_pb2\_grpc

\_ONE\_DAY\_IN\_SECONDS = 60 \* 60 \* 24

# 导入RPC必备的包，以及刚才生成的两个文件(grpc,msg\_pb2,msg\_pb2\_grpc)

# 因为RPC应该长时间运行，考虑到性能，还需要用到并发的库(time,concurrent)

# 在服务器端代码中需要实现proto文件中定义的服务接口（MsgService）,并重写处理函数（GetMsg）

# Python gRPC的服务实现是写一个子类去继承proto编译生成的userinfo\_pb2\_grpc.UserInfoServicer

# 并且在子类中实现RPC的具体服务处理方法，同时将重写后的服务类实例化以后添加到grpc服务器中

class MsgService(msg\_pb2\_grpc.MsgServiceServicer):

# 工作函数

    def GetMsg(self, request, context):

    # 在GetMsg中设计msg.proto中定义的MsgResponse

    # 对收到的request的内容进行读取

        str = request.text

        msg = "从客户端收到的信息为: {}\n".format(request.text)

        # 在服务器端打印从客户端收到的内容并打印结果，用于检验接收的结果是否正确

        print(msg)

        # 将结果返回给客户端

        return msg\_pb2.MsgResponse(id = 1,result = str)

# 通过并发库，将服务端放到多进程里运行

def serve():

# gRPC 服务器

    # 定义服务器并设置最大连接数,corcurrent.futures是一个并发库，类似于线程池的概念

    server = grpc.server(futures.ThreadPoolExecutor(max\_workers=2))# 创建一个服务器server

    msg\_pb2\_grpc.add\_MsgServiceServicer\_to\_server(MsgService(), server)# 在服务器中添加派生的接口服务（自己实现的处理函数）

    server.add\_insecure\_port('[::]:50051')# 添加监听端口,注意保证3个服务器的监听端口不同

    print("服务器已打开，正在等待客户端连接...\n")

    server.start() # 启动服务器，同时start()不会阻塞，如果运行时无事发生，则循环等待

    try:

        while True:

            time.sleep(\_ONE\_DAY\_IN\_SECONDS)

    except KeyboardInterrupt:

        server.stop(0)# 关闭服务器

if \_\_name\_\_ == '\_\_main\_\_':

    serve()

而对于server2.py和server3.py，仅需修改server1.py中第26行和第34行的代码即可：

第26行修改id值：

        return msg\_pb2.MsgResponse(id = 1,result = str)

第34行修改监听端口（与client.py中的端口号保持一致）

server.add\_insecure\_port('[::]:50051')# 添加监听端口,注意保证3个服务

器的监听端口不同

client.py中端口号对应部分代码如下所示：

第13-15行：

    channel1 = grpc.insecure\_channel('localhost:50051')

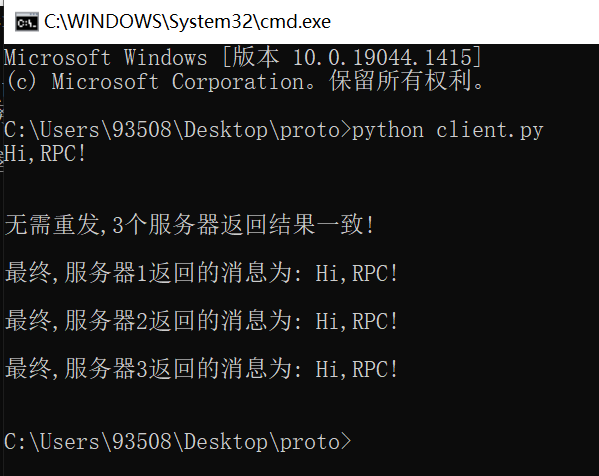
    channel2 = grpc.insecure\_channel('localhost:50052')

    channel3 = grpc.insecure\_channel('localhost:50053')

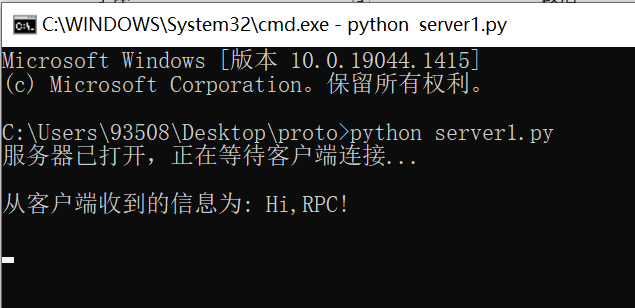
二、实验结果

（1）3个服务器返回结果均正确

①客户端（见下页）：



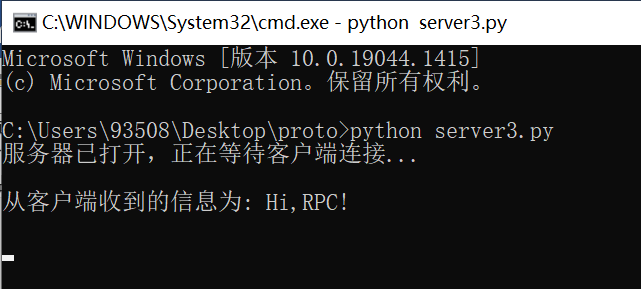
②服务器1：



③服务器2：

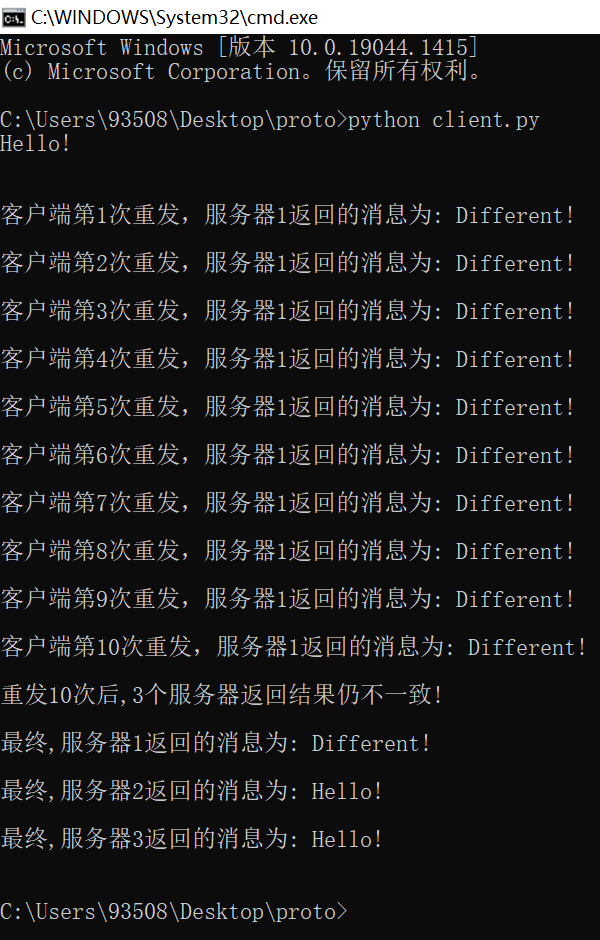


④服务器3：

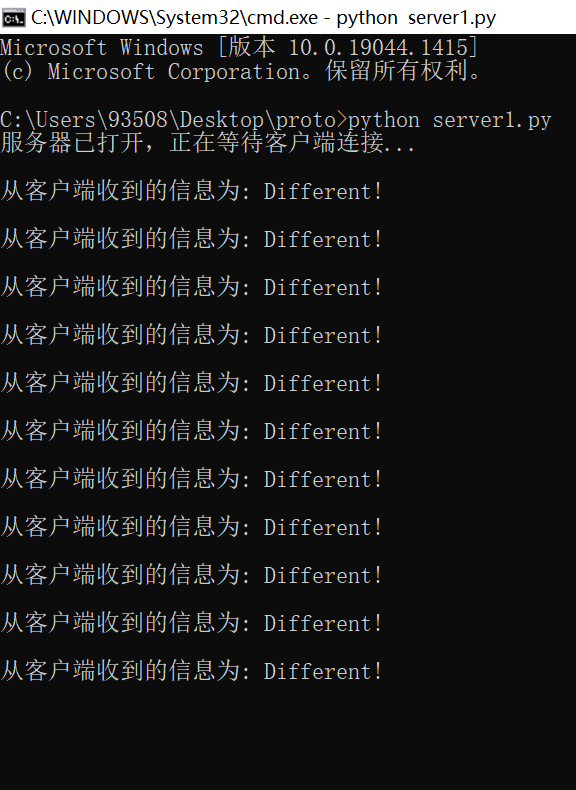


（2）服务器1返回结果错误，另2个服务器返回结果正确

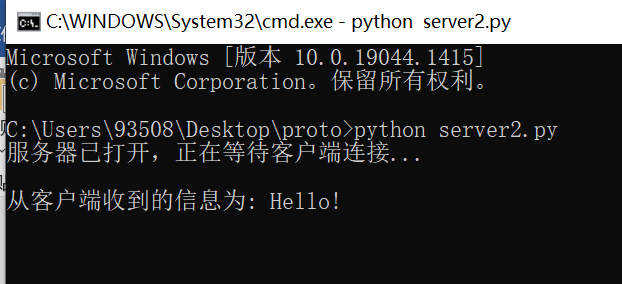
①客户端：



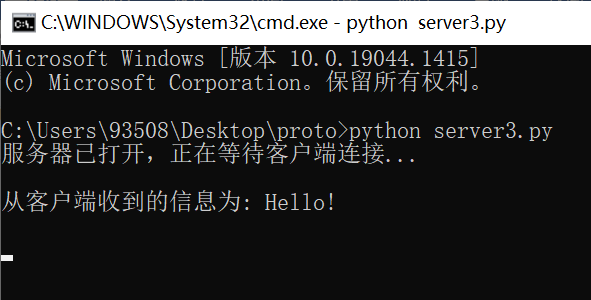
②服务器1（见下页）：



③服务器2：



④服务器3：



结果分析：由于都是在客户端和3个服务器都是在本机上跑的，所以并不存在现实生活中由于传输导致的丢包或者信息错位使得传输信息有误。

所以每次重发的结果都会保持一致，而不会产生变化。但是放在现实生活中，若某次传输发生丢包或信息错误，那么该程序的结果比较和重发机制就会起到作用。

三、遇到的问题及解决方法

难以模拟真实的丢包场景来验证RPC多播的可靠性，但程序逻辑大体上没错。