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
}
```

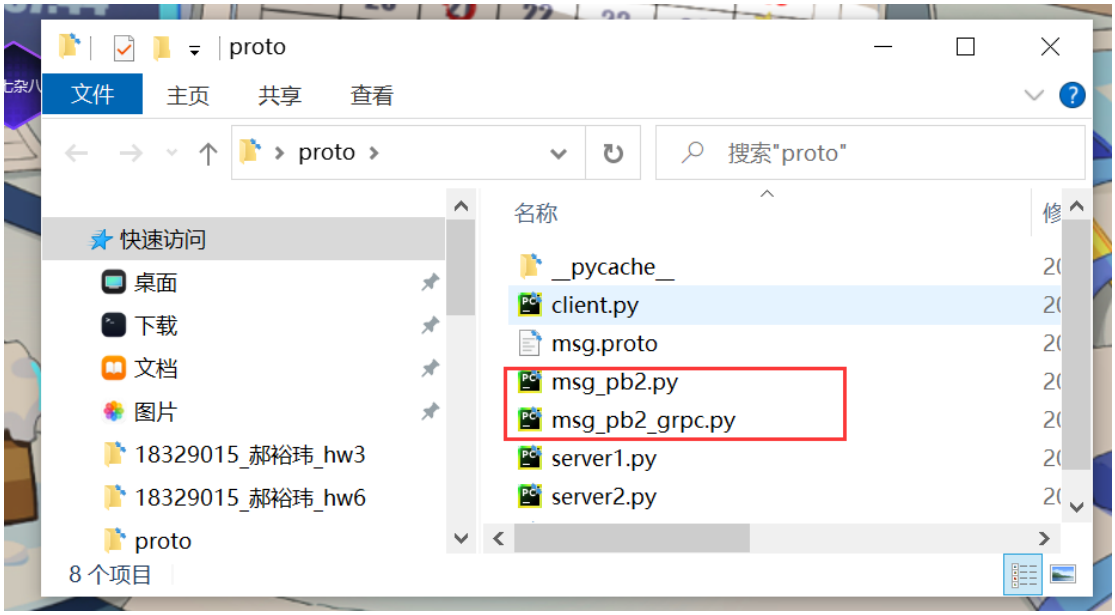
（4）在 proto 文件夹中打开 cmd 窗口，执行如下代码对 msg.proto 进行编译运行：

```
python -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=.
msg.proto
```

```
C:\Users\93508\Desktop\proto>python -m grpc_tools.protoc -I./ --python_out=. --grpc_python_out=. msg.proto

C:\Users\93508\Desktop\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）：


C:\WINDOWS\System32\cmd.exe
Microsoft Windows [版本 10.0.19044.1415]
(c) Microsoft Corporation。保留所有权利。
C:\Users\93508\Desktop\proto>python client.py
Hi,RPC!
无需重发,3个服务器返回结果一致!
最终,服务器1返回的消息为: Hi,RPC!
最终,服务器2返回的消息为: Hi,RPC!
最终,服务器3返回的消息为: Hi,RPC!
C:\Users\93508\Desktop\proto>

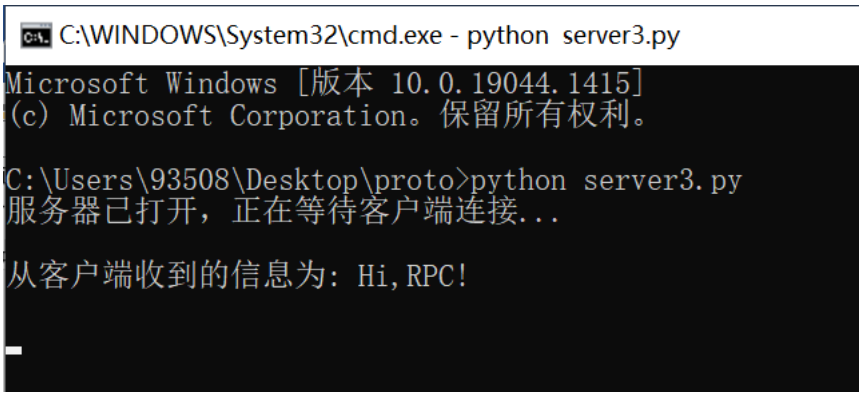
②服务器 1：


C:\WINDOWS\System32\cmd.exe - python server1.py
Microsoft Windows [版本 10.0.19044.1415]
(c) Microsoft Corporation。保留所有权利。
C:\Users\93508\Desktop\proto>python server1.py
服务器已打开，正在等待客户端连接...
从客户端收到的信息为: Hi,RPC!

③服务器 2：

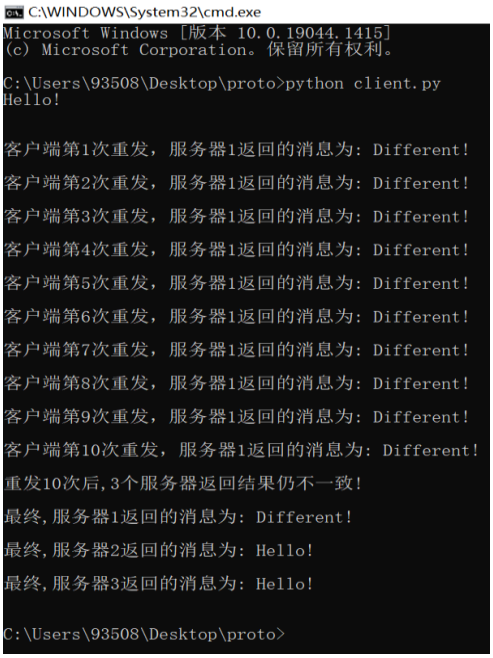

C:\WINDOWS\System32\cmd.exe - python server2.py
Microsoft Windows [版本 10.0.19044.1415]
(c) Microsoft Corporation。保留所有权利。
C:\Users\93508\Desktop\proto>python server2.py
服务器已打开，正在等待客户端连接...
从客户端收到的信息为: Hi,RPC!

④服务器 3：

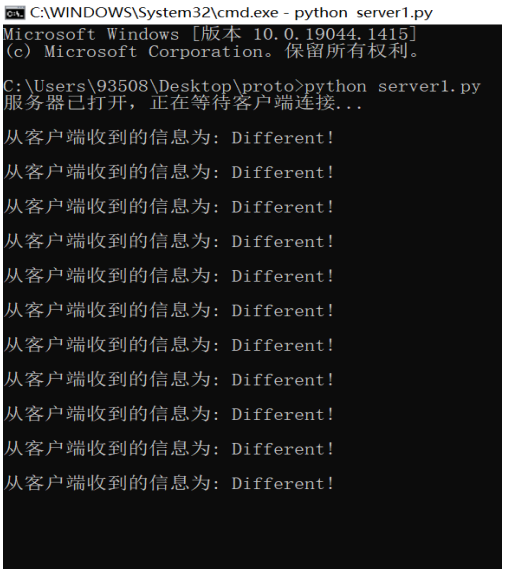


（2）服务器 1 返回结果错误，另 2 个服务器返回结果正确

①客户端：

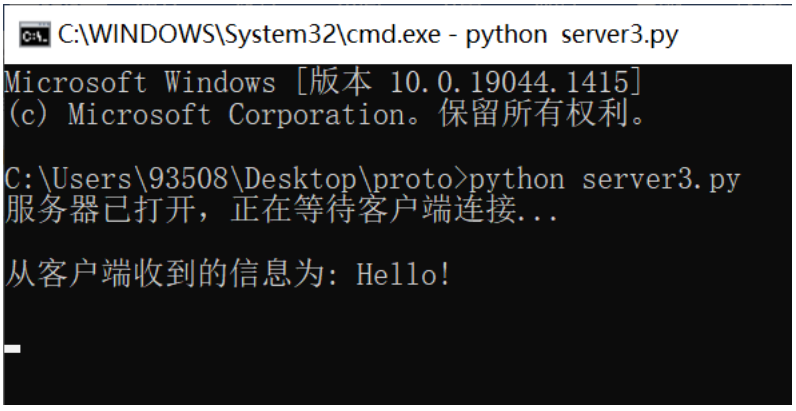


②服务器 1（见下页）：



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。