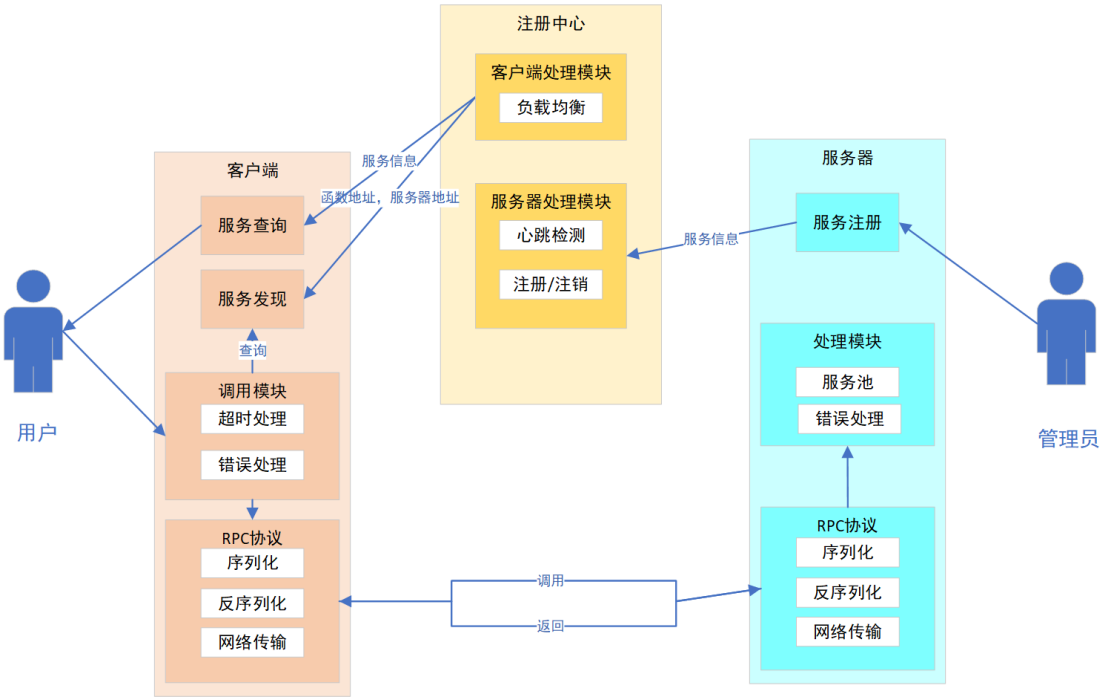
RPC 框架说明

杨鼎文

一、整体架构与使用方法

1、RPC 框架图：



2、代码启动教程：

作业文件夹中包含两个版本，支持 windows 系统和 linux 系统。windows 系统需进入 win 文件夹，mac/linux 系统需进入 mac 文件夹.打开终端，输入 make all，即能够编译出注册中心、客户端、服务器的可执行文件，需要先运行注册中心，才能运行其他程序。

客户端启动参数：-i 注册中心 ip 地址，-p 注册中心端口号，-h 帮助。

服务器启动参数：-l 监听的 ip 地址（为空默认 0.0.0.0），-p 监听的端口号，-h 帮助。

注册中心启动参数：-l 监听的 ip 地址（为空默认 0.0.0.0），-p 监听的端口号，-h 帮助。

示例:./Serverstart -l 0.0.0.0 -p 11111

如果启动参数缺少，则程序运行时会要求输入缺少的参数。当程序成功启动后，可以输入 help 查看可用命令。

由于此项目在 MacOS 系统上进行开发，代码量较大，将程序移植到 windows 系统的过程中遇到了许多困难，如 C++11 标准库 thread，mutex 在 mingw 开发环境不被支持，socket 绑定方法，获取 ip 地址方法也有很大区别，为解决 thread，mutex 不能正常使用的问题，我在 github 上找到了一个库（见 win\rely），可以使其能正常运行。虽然 windows 版本的代码可以被正常的编译运行，但其中可能出现一些错误如 ipv6 地址访问不稳定等，建议在 Linux 或 MacOS 环境下进行代码测试。

3、整体架构：

远程过程调用协议通常对性能有较高的要求，故选择性能较好的 C++语言进行开发。项目采用面向对象的编程思想，客户端，服务端，注册中心的所有方法分别封装在三个类（Server、Client、Register）中，程序运行时，均创建一个对应类型的全局对象，对象中包含了服务器，客户端，注册中心的所有必要信息，并封装了涉及到网络通信处理的所有方法。

注册中心：

```
class Register {
public:
    Register();
    //初始化注册中心，包括监听的 IPv4 地址和端口号，监听的 IPv6 地址和端口号
    void init(string portnum,string portnum6,string listenipnum,string
listenipnum6);
    //准备开始提供服务.
    void readyToServe(bool ipv4=1,bool ipv6=1);
    //监听 IPv4 的客户端连接
    void readytohear(int sock,bool ok);
    //监听 IPv6 的客户端连接
    void readytohear6(int sock,bool ok);
    //处理客户端/服务器请求，根据请求类型调用下面三个函数中的其中一个，得到
string 结果，返回给客户端/服务器
    void handleclient(int sock);
```

```
    //处理服务器注册信息，将服务器信息加入到 serverList 中
    string addServer(string serverinfo);
    //向客户端展示服务器信息，如果 functionname 为空，则展示所有服务器信息
    string showtoClient(string functionname="");
    //客户端准备调用服务，需要得到必要的服务器信息（包括 ip 地址，端口号，函数地址，参数表）
    //如果 ipnum 为空，则按负载均衡选择服务器
    string getServer(string functionname,string ipnum="");
    //监听服务器的心跳
    void hearingheartbeat();
    //处理心跳信息，并将回应发送给服务器
    string handleheartbeat(string heartbeatinfo);
    //计时器，用于检测服务器是否在线
    void resttimecount();
    //展示注册中心自身信息
    void showself(bool ok4,bool ok6);
    //得到服务器数量
    int getnumofserver();
private:
    shared_mutex registerLock;//读写锁
    string listenipnum;//监听的 IPv4 地址
    string listenipnum6;//监听的 IPv6 地址
    string portnum;//监听的端口号
    string portnum6;//监听的端口号
    map<int,int> dic;//心跳包发来的是标识创建时间，为减少查询时间，使用 map 将创建时间映射到服务器 ID
    vector<Server_info> serverList;//服务器列表
    vector<int> resttime;//服务器剩余时间
};
```

服务器：

```
class Server{
public:
    Server();
    int id;//服务器 id，由注册中心分配，若注册失败则为-1
    //添加函数，将函数信息加入到 functionInfo 中
    void addfunction(string functionname,string functioninfo,
vector<string> arglist, string returntype, u64 address);
    //初始化服务器，包括监听的 ipv4 地址、端口号、ipv6 地址、端口号、自身 ipv4
地址、ipv6 地址、创建时间
    void init(string acceptipnum,string acceptipnum6, string
portnum,string portnum6, string selfipnum, string selfipnum6,int
create_time);
    //注册服务器，将服务器信息发送给注册中心，注册成功则将 id 赋值为注册中心分
配的 id，失败则为-1
    void regist(string reg_ipnum, string reg_portnum ,int iptype);
    //服务器准备就绪，开始监听，ok4 和 ok6 分别表示是否监听 ipv4 和 ipv6
    //将开启 acceptclient 和 acceptclient6 两个线程，分别监听 ipv4 和 ipv6
    void readyToServe(bool ok4,bool ok6);
    //接受客户端连接，ok 为 true 表示监听 ipv4，false 表示不监听
    void acceptclient(int sock,bool sign);
    //接受客户端连接，ok 为 true 表示监听 ipv6，false 表示不监听
    void acceptclient6(int sock,bool sign);
    //处理客户端请求，包括接收客户端数据，解析数据，调用函数，返回结果
    void handleclient(int sock);
    //心跳，每 5 秒向注册中心发送心跳包，若 3 次未收到回复则停止心跳
    void heartbeats();
    //下线，向注册中心发送下线信息
    void deadhb();
    //显示服务列表
```

```
    void showavailable();
    //注册或注销函数，sign 为 1 表示注册，0 表示注销
    //当接收到正确返回信息时，再将本地函数状态改变
    void beavailableornot(string functionname,bool sign);
private:
    Server_info info;//服务器信息，包含函数列表、ip 地址、端口号、等
    mutex serverLock;//服务器锁，用于锁客户端数量
    string reg_ipv4;//注册中心 ipv4 地址
    string reg_port4;//注册中心 ipv4 端口号
    int id;//服务器 id，由注册中心分配，若注册失败则为-1
};
```

客户端：

```
class Client {
public:
    Client();
    //初始化客户端，并尝试连接注册中心，返回是否成功
    bool init(string reg_ipnum,string reg_portnum, int iptype);
    //查询服务，若 functionname 为空则查询所有服务，否则查询指定服务，返回信息
    string checkServer(string functionname="");
    //调用服务前的准备工作，向注册中心查询服务，返回服务 ip 地址、端口号、函数
地址、参数个数、参数列表
    //若 ipnum 不为空则查询指定 ip 的服务
    handel findServer(string functionname,string ipnum="");
    //调用服务，向服务端发送数据，返回服务端处理结果
    //hand 包含服务端 ip 地址、端口号、函数地址、参数列表，data 包含可序列化的
数据
    Rpc_data callServer(handel hand,Rpc_data data);
private:
    int reg_iptype;//注册中心的 ip 类型
    string reg_ipnum;//注册中心的 ip 地址
```

```
    string reg_portnum;//注册中心的端口号
};
```

二、消息传递及序列化消息

客户端/服务器与注册中心之间的通信不涉及服务调用，参数传递等，故无需使用序列化消息，而是直接传递字符串，根据字符串内容进行服务注册、服务发现。客户端与服务器之间的通信则使用了序列化消息，便于传递大量参数、简化函数接口。

所有涉及网络通信的命令执行步骤如下：

i、用户输入命令

ii、发送端请求建立 TCP 连接

iii、发送端等待接收端处理

iv、发送端接收到返回的信息

v、关闭 TCP 连接

vi、输出用户可见的结果

每次建立的 TCP 连接只负责发送一条请求，接收一条返回结果（类似于 TCP 非持续性连接），这样不仅保证了消息的正确，解决了粘包问题，同时避免长时间保持 TCP 连接所需要的额外开销。

用空格将信息进行分隔，以便使用 stringstream 对数据进行处理。“”中的消息不存在空格。

1、服务器与注册中心间的通信：

（1）、服务器向注册中心进行注册的消息格式（于 Server.h 函数 void Server::regist()）：

Server “监听地址” “监听端口号” ……

将服务器的全部信息发送给注册中心，包括服务器所在 ip 地址，服务器监听的 ip 地址和端口号，同时将服务器内可用服务（函数）的信息逐个发送，包括函数地址，**参数列表**，返回类型。最后向注册中心发送服务器创建时间，作为服务器的唯一标识。

这里的参数表不同于平常的参数表，采用了自定义的消息类型，形如 5*int，

2*double/float，1*string，目的是方便指导用户进行手动输入参数。

（2）、注册中心返回服务器注册请求消息格式（于 Register.h 函数 string Register::addServer()）：

OK “服务器在注册中心上的 id”“注册中心 ipv4 地址”“注册中心 ipv4 端口号”

注册中心被设定为强制支持 ipv4 地址访问，且心跳包发送部分是使用 ipv4 地址进行发送，所以这里返回了注册中心的 ipv4 地址与端口号，以便于让服务器发送心跳包给注册中心。

（3）、服务器向注册中心注册\注销指定函数的消息格式：

Serverchange “服务器创建时间” “函数名”

用于在注册中心上改变函数状态。

（4）、服务器向注册中心发送心跳包消息格式（于 Server.h 函数 void Server::heartbeats()）：

Heartbeat“服务器创建时间”“在此服务器上正在运行的客户端数量”。

同时发送服务器上正在运行客户端的数量以便于注册中心进行负载均衡。

死亡心跳（于 Server.h 函数 void Server::deadhb()）：

Dead “服务器创建时间”

死亡心跳用于快速告知注册中心该服务器已下线，以避免注册中心将客户端请求分配给已下线的服务器。

（5）、注册中心向服务器发送心跳包回复（于 Register.h 函数 string Register::handleheartbeat()）：

OKhb

用于告知服务器注册中心接收到心跳，服务器会依据这个返回消息，判断注册中心是否还保留自己的服务。

2、客户端与注册中心进行通信：

（1）、客户端与注册中心进行试连接（于 Client.h 函数 bool Client::init()）：

Client connect

如果此消息发送失败或者发送超时，代表客户端初始化时指定注册中心的信

息可能有误，需要重新指定注册信息。

（2）、客户端查询所有服务器所有可用服务（于 Client.h 函数 string Client::checkServer()）：

Client all

注册中心返回所有服务器信息作为回应，此消息不需要进行处理，直接在客户端输出即可。

（3）、客户端查询拥有指定服务名的服务器的信息(于Client.h函数string Client::checkServer()）：

Client check “函数名”

注册中心返回有指定服务名的服务器信息作为回应，同样的，此消息不需要进行额外处理。

（4）、客户端准备调用服务，向注册中心请求必要的服务器信息（于 Client.h 函数 handel Client::findServer()）：

用户未手动指定服务器：

Client get withoutip “函数名

用户手动指定服务器：

Client get withip “函数名” “服务器 ip 地址”

（5）、注册中心返回客户端的调用服务器请求（于 Register.h 函数 string Register::getServer()）：

找到该服务：

“服务器 ipv4 地址” “服务器 ipv6 地址” “服务器 ipv4 端口号” “服务器 ipv6 端口号” “函数地址” “函数参数个数” “函数参数” …… “函数信息”

（……表示参数表）

未找到该服务：

Function not found

上文提到自定义参数表消息格式，在这里得到运用，如消息为：

*0.0.0.0 fe80:: 12345 12346 2234234(函数地址) 3 2*int 4*double 1*string 函数信息*

使用这种消息类型，可以在客户端中实现提示用户输入 2 个 int 类型的参数，4 个 double 类型的参数，1 个 string 类型的参数。进行处理后即可生成客户端将要向服务器发送的可序列化消息（于 Client.cpp 函数 Rpc_data handtoRpc_Manual()）。

3、客户端与服务器之间的通信：

由于查找服务，注册服务部分都是在注册中心进行，客户端只需要直接向服务器发送函数地址以及序列化消息即可。

客户端调用服务请求：

“函数编号” “序列化消息”

服务器返回结果：

“序列化消息”

4、序列化消息格式：

C++基本数据类型包括 int、string、double、char、float（可用 double 代替）、bool（可用 int 代替），大多数的 c++函数（不涉及内存操作、如指针等）的参数，返回值都可以用这四种类型表示，故使用以下的消息序列：

“string 类型个数”“字符串 1”“字符串 2”..... “int 类型个数”“整形 1”“整形 2”..... “double 类型个数”“浮点数 1”..... “char 类型个数” “字符 1”.....

用空格对每个信息进行分割，但由于字符串中可能会含有空格，导致序列化消息格式被破坏，这里需要对 stirng 类型的数据进行特殊处理，在项目中使用转义符‘\1’将 string 数据包裹，如字符串“ydwnb”在序列化消息中是“\1ydwnb\1”，这样在序列化反序列化方法中使用额外的方法处理 string 类型的参数，就可以解决以上问题。

序列化消息示例：2 \1nb\1 \1bnb sdhsdl\1 0 2 1.2 4.3 1 c

含义：

2 个 string 分别为“nb”“bnb sdhsd”

0 个 int

2 个 double 1.2，4.3

1 个 char‘c’

在后面将会详细介绍处理序列化消息的类：Rpc_data。

三、项目实现细节：

1、服务器注册服务方法：

组织服务的数据结构：

所有服务均是一个函数，故自定义一个 Function_info 结构体，每个服务器内存有一个 Function_info 矢量，用于存储可用服务信息。

```
struct Function_info{
    string functionname;//函数名
    string functioninfo;//函数信息,描述
    vector<string> arglist;//参数列表
    string returntype;//返回值类型
    u64 address;//函数地址（编号）
    bool available;//是否可用
};
```

在服务端启动程序中，预先编码了许多可用的服务，并已将函数信息全部存储在 server 对象中。

```
server.addfunction("add","两数相加",stringtovec("1*double 1*double"),"double",(u64)add);
server.addfunction("sub","两数相减",stringtovec("2*float"),"double",(u64)sub);
server.addfunction("matrix5_5","计算5*5的行列式值",stringtovec("25*int"),"int",(u64)matrix5_5);
server.addfunction("String_reverse","反转字符串",stringtovec("1*string"),"string",(u64)String_reverse);
server.addfunction("bigintplus","大整数相加",stringtovec("2*string"),"string",(u64)bigintplus);
server.addfunction("quadratic","解一元二次方程，三个参数分别为a,b,c",stringtovec("3*float"),"double string",(u64)quadratic);
server.addfunction("longtimeprocess","模拟长时间处理",stringtovec("1*string"),"string",(u64)longtimeprocess);
server.addfunction("runtimeprocess","模拟超时处理",stringtovec("1*string"),"string",(u64)runtimeprocess);
server.addfunction("testString","测试处理带空格的字符串",stringtovec("1*int 1*int 2*string"),"string",(u64)testString);
server.addfunction("sendfile","接收文件",stringtovec("1*file"),"string",(u64)senfile);
```

服务器初始化时需要指定一个注册中心，服务器将会把所有预先编码的函数信息全部发送给注册中心，之后便开始服务，如果未注册成功将会提示用户进行重新注册，服务器需要向注册中心发送服务器监听的的 ip 地址，如果用户输入监听的 ip 是 0.0.0.0，则服务器获取一个本机上的一个 ip 地址用于发送。

```
server.regist(reg_ip,reg_port,getiptype(reg_ip));
server.readyToServe(ip4ok,ip6ok);
if(server.id!=-1)cout<<"请手动输入命令register，以注册服务函数"<<endl;
```

服务器启动成功后，可以手动输入命令改变函数可用性，函数 beavaliableornot 将会发送更新消息给注册中心，并接受返回 ，接收到注册中心的成功消息后才会改变本地可用性。

```
else if(command=="register"){
    string name;
    cout<<"请输入函数名：<<endl;
    cin>>name;
    server.beavailableornot(name,1);
}
else if(command=="unregister"){
    string name;
    cout<<"请输入函数名：<<endl;
    cin>>name;
    server.beavailableornot(name,0);
}
```

2、服务发现：

客户端初始化时需绑定一个注册中心，所有的服务发现请求均在注册中心上处理。客户端可以输入命令检索所有的可用服务或检索需要的服务，客户端将会发送查询请求给注册中心，注册中心将服务器的可用服务信息（包括服务器 ip 地址等信息）返回给客户端：

```
else if(command=="checkall"){
    string serveinfo = client.checkServer();
    cout<<serveinfo<<endl;
}
else if(command=="check"){
    string sevicename;
    cout<<"请输入服务名：<<endl;
    cin>>sevicename;
    string serveinfo = client.checkServer(sevicename);
    cout<<serveinfo<<endl;
}
```

当客户端选择进行调用服务时，需输入正确的函数名，并可以选择指定 ip 的服务器进行调用，注册中心将会返回必要的调用服务信息（包括服务器 ip 地址，端口号，函数地址，函数参数表），最后客户手动输入所需参数，转化为序列化消息，即可开始调用：

```
else if(command=="call"){
    string sevicename;
    cout<<"请输入服务名：<<endl;
    cin>>sevicename;
    cout<<"是否手动指定服务器？(y/n)"<<endl;
    char c;
    cin>>c;
    string ip="";
    if(c=='y'){
        cout<<"请输入服务器ip地址：<<endl;
        cin>>ip;
    }
    handle hand = client.findServer(sevicename,ip);//ip为空则自动选择服务器
    if(hand.ser_ipnum=="fail"){
        cout<<"调用服务失败！<<endl;
        cout<<"错误信息：<<hand.ser_ipnum6<<endl;
        continue;
    }
    Rpc_data data=handtoRpc_Manual(hand);
    cout<<"服务器处理中"<<endl;
    Rpc_data result=client.callServer(hand,data);
    cout<<"调用结果：<<endl;
    result.show();
}
```

Client 的 findServer 函数将会返回一个 hand，其定义如下：

```
struct handle{
    string ser_ipnum;//服务端ipv4地址
    string ser_pornum;//服务端ipv4端口号
    string ser_ipnum6;//服务端ipv6地址
    string ser_pornum6;//服务端ipv6端口号
    vector<string> arglist;//参数列表
    u64 address;//指定函数的地址
    string funcinfo;//函数信息，用于向用户展示
    handle(string sip,string sip6,string spor,string spor6,u64 add,vector<string> alist=vector<string>(),string funinfo=""){
        ser_ipnum=sip;
        ser_ipnum6=sip6;
        ser_pornum=spor;
        ser_pornum6=spor6;
        address=add;
        arglist=alist;
        funcinfo=funinfo;
    }
};
```

如果在 findServer 时发生错误，ser_ipnum 将会被置为 fail，ser_portnum 将会存储错误信息，如未找到该服务，连接注册中心失败等。之后进入函数 handtoRpc_Manual，通过返回的函数参数表，提示用户输入每个参数，并返回可序列化消息 RPC_data。最后进行调用，返回 result，并展示结果（为实现协作通信部分的文件互传，源代码中客户端在这一部分也实现了接收文件，因较臃肿，故未展示在报告中）。

3、服务调用：

客户端调用服务器请求的消息格式为：**“函数编号” “序列化消息”**，服务器读函数编号以确定客户端所需服务，在 C++编程中可直接使用函数地址作为

函数编号。

定位服务后，为方便服务器处理数据，封装了类 RPC_data（于 RPC.h），拥有成员变量：strings、ints、doubles 和 chars，分别是存储字符串、整数、浮点数和字符的向量。

```
class Rpc_data{
public:
    Rpc_data();
    vector<string> strings;//string列表
    vector<int> ints;//int列表
    vector<double> doubles;//double列表
    vector<char> chars;//char列表
    SerializedData serialize();//序列化（SerializedData就是string）
    void deserialize(SerializedData data);//反序列化
    void show();//展示数据
};
```

服务器的所有函数都是形如 Rpc_data func(Rpc_data data)。通过对 Rpc_data 的处理，即可实现绝大部分函数的功能，简化了函数接口的设计。例如：

```
Rpc_data add(Rpc_data data){
    double a,b;
    a=data.doubles[0];
    b=data.doubles[1];
    Rpc_data result;
    result.doubles.push_back(a+b);
    return result;
}
```

Rpc_data 类中实现了序列化 serialize 和反序列化 deserialize 方法，能够将 Rpc_data 里的所有数据全部转化为上文提到的序列化消息。这种数据结构具有较高的可拓展性，通过在客户端服务器进行编码，可以实现许多功能，如传输文件，更复杂的函数处理等。

服务器处理请求细节：这里 info 代表客户端发送过来的完整消息。

```
stringstream ss(info);
string functionaddstr;
ss >> functionaddstr;
u64 functionadd=stoull(functionaddstr);
SerializedData serializedData(info.substr(functionaddstr.size()+1,info.size()-functionaddstr.size()-1));
Rpc_data data;
data.deserialize(serializedData);
Rpc_data (*func)(Rpc_data);
func = (Rpc_data (*)(Rpc_data))functionadd;
cout<<"对函数"<<(u64)func<<"进行调用"<<endl;
cout<<"请求处理中"<<endl;
Rpc_data result = func(data);
SerializedData resultstr = result.serialize();
// 发送结果给客户端
cout<<"请求处理完成"<<endl;
cout<<"正在将结果发送给客户端"<<endl;
ssize_t bytesSent = sendAll(clientSocket, resultstr.c_str(), resultstr.size());
```

服务器读取到函数地址后，再将后面的序列化消息转化成 Rpc_data，并直接通过函数地址调用目标函数。返回 Rpc_data 结果，再将其序列化发送回去。客户端收到序列化结果后进行反序列化，再调用 Rpc_data 里的成员函数 show，以展示可视化消息。

4、注册中心：

为方便进行服务器的管理，每个服务器需要将服务上传到注册中心，以便于客户端进行服务发现。服务器在与注册中心连接成功后，会将预编码的所有函数的信息发送给注册中心，单个服务并不是永久注册，在服务器上可以输入命令以注册或注销某个服务。服务器也不是永久注册在注册中心上，注册中心与服务器之间实现了心跳检测，注册中心接收不到某个服务器的心跳包超过一段时间后，注册中心将会把该服务器信息剔除。

心跳检测：

如果服务器注册成功，则将开启一个新线程，每隔 5s 进行一次心跳。

```
//开始心跳
if(id!=-1){
    thread heartbeatsThread(&Server::heartbeats,this);
    heartbeatsThread.detach();
    cout<<"服务器注册成功，注册于："<<reg_ipnum <<" id:"<<id<<endl;
}
```

服务器同时接收返回数据，如果超过三次未接收到注册中心的返回数据，则服务器将停止心跳，并要求用户重新注册。

```
if((response!="OKhb"||bytesReceived==-1)){
    if(sign==3){
        id=-1;
        cout<<"已经3次未收到心跳回复，服务器停止心跳"<<endl;
        cout<<"强烈建议重新注册"<<endl;
        return;
    }
```

注册中心启动成功后将会开启两个线程：

hearingheartbeat 监听所有心跳，并将监听到的心跳信息输入 handleheartbeat 进行处理，并得到需要返回的消息，如果接收到心跳信息则将服务器剩余时间重置为 20s。hearingheartbeat 单线程完全可以应付 20 个以上服务器的心跳，代码中使用 map 创建了服务器创建时间到 id 的映射，接收到心跳消息与返回心跳消息中间的代码时间复杂度只有 o(1)：

```
int idnum=dic[createtime];
string clientnum;
ss>>clientnum;
serverList[idnum].numofclients=stoi(clientnum);
resttime[idnum]=20;
return "OKhb";
```

如果接收到服务器下线信息也仅是将服务器剩余时间置为 0，具体计算量大的删除部分由另一个线程 resttimecount 处理。

resttimecount 对所有服务器进行倒计时，如果时间为 0 则剔除该服务器。虽然有两个线程在同时对 resttime 进行写操作，但并不致命。

```
while(true){
    //每隔二秒将resttime中的时间减二
    for(int i=0;i<resttime.size();i++){
        if(serverList[i].create_time==-1){
            continue;
        }
        resttime[i]-=2;
        //这里虽然会发生resttimecount与handleheartbeat同时修改resttime的问题，
        //但不致命，因为心跳包频率是5s一次，总会有心跳包与resttimecount不同时修改的情况
        //这样已经达到更新resttime的目的了
        if(resttime[i]<=0){
            unique_lock<shared_mutex> lock(registerLock);//加锁
            cout<<"服务器"<<" id:"<<i<<"已下线"<<endl;
            serverList[i]=Server_info();//将ServerList标志为死(Server_info中create_time为-1，表示死了)
            //删除字典中的信息
            for(auto it=dic.begin();it!=dic.end();it++){
                if(it->second==i){
                    dic.erase(it);
                    break;
                }
            }
        }
    }
    sleep(2);
}
```

注册中心处理服务器请求：

于 Register.h 的函数 void Register::handleclient(),此函数较为简单。包含对上文的消息请求的处理

5、并发处理：

服务器与注册中心均使用了多线程进行并发。

服务器：服务器启动服务后，会调用服务器类 readyToServe 成员函数，分别创建 IPv4 和 IPv6 的套接字，然后启动两个独立线程 acceptclient，和 acceptclient6 分别处理 IPv4 和 IPv6 的客户端连接，并将绑定好的欢迎套接字传递进去。

```
thread ipv4thread(&Server::acceptclient,this,serverSocket4,ok4);
thread ipv6thread(&Server::acceptclient6,this,serverSocket6,ok6);
ipv4thread.detach();
ipv6thread.detach();
```

acceptclient(6)进入一个无限循环，不断调用 accept 函数等待客户端连接，当有客户端连接时，立即将函数 handleclient 作为新线程调用，并将连接套接字传入，让新线程进行消息的接收，处理与返回。自身进入下一个循环，确保接受链接不会因为等待函数处理而阻塞。

```
if(!sign)return;
while (true) {
    sockaddr_in clientAddr;
    socklen_t clientAddrSize = sizeof(clientAddr);
    int clientSocket = accept(sock, (struct sockaddr*)&clientAddr, &clientAddrSize);
    if (clientSocket == -1) {
        cout << "接受连接失败" << endl;
        close(sock);
        return;
    }
    thread clientThread(&Server::handleclient,this, clientSocket);
    clientThread.detach();
}
close(sock);
```

注册中心：与服务器的方法基本一致，但是在 redayToServe 中还创建了线程 resttimecount 和 hearingheartbeat，一个用于处理心跳倒计时，另一个用于处理所有服务器的心跳信息。注册中心上涉及许多写操作，比如增删服务等，故额外使用了读写锁 shared_mutex 以保证并发的安全性：（以服务器改变函数可用性为例）

```
int createtime;
ss>>createtime;
int id=dic[createtime];
string functionname;
ss>>functionname;
unique_lock<shared_mutex> lock(registerLock);
for(int i=0;i<serverList[id].functionInfo.size();i++){
    if(serverList[id].functionInfo[i].functionname==functionname){
        if(serverList[id].functionInfo[i].available){
            serverList[id].functionInfo[i].available=0;
            cout<<"服务器"<<id<<"的函数"<<functionname<<"已下线"<<endl;
        }
        else{
            serverList[id].functionInfo[i].available=1;
            cout<<"服务器"<<id<<"的函数"<<functionname<<"已上线"<<endl;
        }
        break;
    }
}
reply="OK";
```

理论上注册中心可注册 10 个以上的服务器，每个服务器也可以同时处理 10 个以上的客户端请求。（服务器预编写了指定时间处理的服务供测试）。

6、超时与异常处理：

首先需要确保消息能够完整的被发送和接收，send 函数受到网络底层的限制，不能一次性发送过长的消息，所以自定义一个函数 sendAll：

```
ssize_t sendAll(int socket, const char* buffer, size_t length) {
    ssize_t n;
    long long total = 0;
    while (total < length) {// 循环发送数据
        n = send(socket, buffer + total, length - total, 0);
        if (n == -1) break; // 发送失败
        total += (long long)n;
    }
    // 发送结束标志
    if (n != -1) {
        n = send(socket, "\0", 1, 0); // 使用'\0'作为结束标志
    }
    return n==-1?-1:(ssize_t)total;
}
```

将一个长消息分多次发送，并且在最后发送‘\0’作为消息结束标志。

重写自带的函数 recv 为 recvWithTimeout（与原版 recv 相似，使用 string& 以动态接收数据，最后一个参数为超时时长）：

```cpp
int recvWithTimeout(int socket, string& buffer, size_t length, int timeoutSeconds=30) {
    ssize_t n;
    int total = 0;
    char tempbuf[1024];
    while (length==-1||total < length) {
        fd_set fds;
        FD_ZERO(&fds);
        FD_SET(socket, &fds);
        struct timeval timeout;
        timeout.tv_sec = timeoutSeconds;
        timeout.tv_usec = 0;
        int activity = select(socket + 1, &fds, NULL, NULL, &timeout);
        if (activity < 0)return -1;
        else if (activity == 0) return 0; // 超时，返回错误
        n = recv(socket, tempbuf, 1024, 0);
        if (n == -1) return -1; // 接收错误
        else if (n == 0) return -1; // 对端关闭连接
        total += n;
        if (tempbuf[n - 1] == '\0') {
            buffer.append(tempbuf, n - 1);
            break;
        }
        buffer.append(tempbuf, n);
    }
    return (int)total;
}
```

此函数会将‘\0’前的消息读进缓冲区，若迟迟未收到‘\0’则代表消息可能出错，函数返回 0 并在函数外做超时处理。使用 select 函数检测指定时间内套接字是否可读。

C++ 中的 connect 同样不存在连接超时选项，故自定义函数 connectWithTimeout，用以处理连接超时：

```cpp
int connectWithTimeout(int sock, const struct sockaddr *addr, socklen_t addrlen, int timeout_sec) {
    // 设置 socket 为非阻塞模式
    int flags = fcntl(sock, F_GETFL, 0);
    if (flags < 0) return -1;
    if (fcntl(sock,F_SETFL,flags|O_NONBLOCK)<0) return -1;
    int result = connect(sock, addr, addrlen);
    if (result < 0 && errno != EINPROGRESS) return -1;
    if (result < 0 && errno == EINPROGRESS) {
        fd_set set;
        FD_ZERO(&set);
        FD_SET(sock, &set);
        struct timeval tv;
        tv.tv_sec = timeout_sec;tv.tv_usec = 0;
        result = select(sock + 1, NULL, &set, NULL, &tv);
        if (result < 0) return -1;
        else if (result == 0) return -1; // 连接超时
        else {
            int error = 0;
            socklen_t len = sizeof(error);
            if (getsockopt(sock, SOL_SOCKET, SO_ERROR, &error, &len) < 0)return -1;
            if (error != 0)return -1;
        }
    }
    if (fcntl(sock, F_SETFL, flags) < 0)return -1;
    return 0;
}
```

类似的使用 select 进行超时处理，只不过需要临时将 socket 设置为非阻塞。

发送方异常/超时处理（以服务器发送注册信息为例）：

在项目中所有的通信都是发送一条消息，接受一条消息，且所有的通信操作都被封装在对应对象的某个方法，如果建立连接时产生异常或超时：

```
if (connectWithTimeout(clientSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr),10) == -1) {
    cout << "连接失败，请检查注册中心ip地址和端口号输入是否正确" << endl;
    close(clientSocket);
    return ;
}
```

对应的函数将提示错误信息并直接返回。

如果发送数据时产生异常：

```
ssize_t bytesSent = sendAll(clientSocket, info.c_str(), info.size());
if (bytesSent == -1) {
    cout << "发送注册信息失败" << endl;
    close(clientSocket);
    return ;
}
```

同样返回错误信息，并结束函数。

接收方异常/超时处理（以客户端调用服务后等待返回为例）：

如果接收不到响应，或是接收响应失败：

```
int bytesReceived = (int)recvWithTimeout(sock, buffer, 102400,40);
if (bytesReceived == 0) {
    reply.strings.push_back("数据接收超时");
    return reply;
}
else if (bytesReceived < 0) {
    reply.strings.push_back("数据接收失败");
    return reply;
}
```

类似地，立刻返回错误信息，这里的 reply 是 Rpc_data 的一个对象，将在函数外展示这个失败信息。

其余的消息发送与接收的异常处理方法都与上面的类似。(服务器预编写了指定处理时间代码供测试)

由于使用了自定义的消息发送和接受方法,确保了消息要么是完整的(以‘\0’结束），要么就直接返回错误标志，同时使用 TCP 协议进行传输，消息内容本身也不会出错，故不需要担心消息处理出现错误。

如果用户输入错误，当输入错误的 ip 地址和端口号时，通常会输出连接失败的错误,如果输入的数据不合法,则程序会提示重新输入,如端口号不合法时：

```
while(!isgoodport(regport)){
    cout<<"端口号不合法，请重新输入"<<endl;
    cin>>regport;
}
```

其中 isgoodport 是自定义函数，判断端口号是否合理。

```
bool isgoodport(string portnum){
    for(int i=0;i<portnum.size();i++){
        if(portnum[i]<'0'||portnum[i]>'9'){
            return false;
        }
    }
    int port=stoi(portnum);
    if(port>65535||port<0){
        return false;
    }
    return true;
}
```

7、负载均衡：

负载均衡部分由注册中心实现,服务器每次发送心跳包时将会更新在此服务器上运行的客户端的数量。当客户端想要调用服务时，若客户端没有指定需要调用的服务器，注册中心将返回客户端数量最少的服务器，若有多个最优服务器，则随机选取其中的一个进行调用，以防止总是调用同一个服务器。（于 Register.h 函数 string getServer()）：

```cpp
vector<Server_info> avaliableServer;
for(int i=0;i<serverList.size();i++){
    if(serverList[i].create_time==-1){
        continue;
    }
    for(int j=0;j<serverList[i].functionInfo.size();j++){
        if(serverList[i].functionInfo[j].available==0){
            continue;
        }
        if(serverList[i].functionInfo[j].functionname == functionname){
            avaliableServer.push_back(serverList[i]);
            break;
        }
    }
}
sort(avaliableServer.begin(),avaliableServer.end(),[](Server_info a,Server_info b){return a.numofclients<b.numofclients;});
if(avaliableServer.size()==0)return "Function not found";
int bestcount=0;
int thebest=avaliableServer[0].numofclients;
for(int i=0;i<avaliableServer.size();i++){
    if(avaliableServer[i].numofclients!=thebest)break;
    bestcount++;
}
srand(time(0));
int id=rand()%bestcount;
string reply=avaliableServer[id].selfipnum+" "+avaliableServer[id].selfipnum6 +" ";
reply+=avaliableServer[id].portnum + " " +avaliableServer[id].portnum6;
for(int j=0;j<avaliableServer[id].functionInfo.size();j++){
    if(avaliableServer[id].functionInfo[j].functionname == functionname){
        reply+=" "+to_string(avaliableServer[id].functionInfo[j].address);
        reply+=" "+to_string(avaliableServer[id].functionInfo[j].arglist.size());
        for(int k=0;k<avaliableServer[id].functionInfo[j].arglist.size();k++){
            reply+=" "+avaliableServer[id].functionInfo[j].arglist[k];
        }
        break;
    }
}
return reply;
```

8、网络协议的选择：

除了服务器发送心跳包选择了 UDP 协议外，其余所有通信均选择使用 TCP 协议，TCP 协议保障了消息的正确性。如果消息序列出错，将可能导致程序的崩溃。心跳包发送选择 UDP 是因为服务器需要频繁的发送心跳包，用 TCP 传输的话开销过大。且丢失一两个心跳包，或是心跳消息出错对程序影响不大。

9、协作通信：

与封鹏威 21310255 使用了相同的消息类型，并使用了同样的序列化消息（除了服务器与注册中心间的通信），实现了 C++与 python 语言之间的相互通信和调用，所有函数调用功能均能实现。此外，我们使用序列化消息实现了文件传输的功能，可以发挥 python 处理文件的优势。