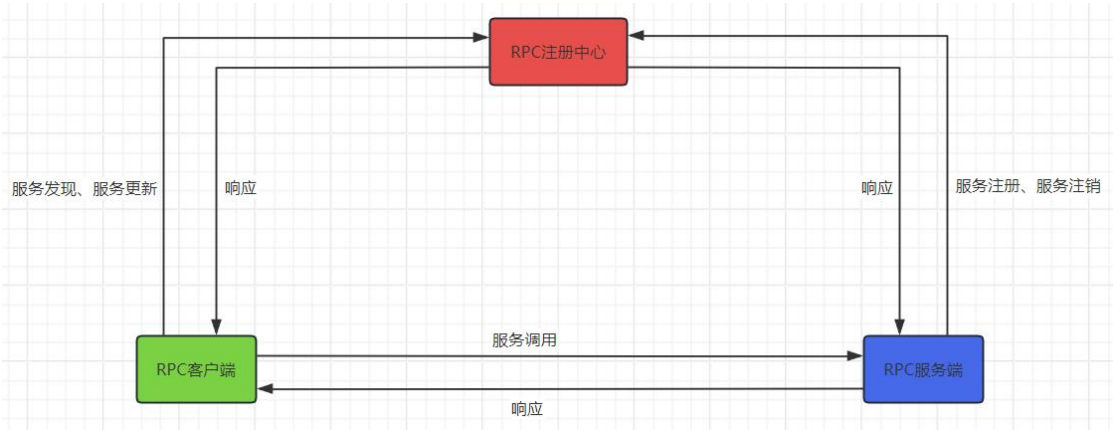


# MingleRPC 设计文档

余XX 2131XXXX

## 一、RPC 整体架构图



## 二、开发环境

- C++语言及其多线程、原子变量等库
- Linux 操作系统及其网络编程库
- 除上述提到的内容外，RPC的实现不再涉及任何第三方库和框架

## 三、RPC 功能需求分析

根据实验要求，整个 RPC 框架需要实现的功能有：①消息格式定义，消息序列化和反序列化②服务注册③服务发现④服务调用⑤**服务注册中心**（可选）⑥支持并发⑦异常处理及超时处理⑧**负载均衡**（可选）⑨**协作开发**（可选）。在本RPC实现中，除了协作开发，其余功能均得到了实现。

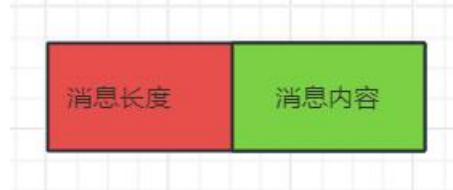
根据功能分析结果，功能被分配给了RPC框架的三端来协同实现，分配情况如下：

- 注册中心：①②③⑥⑦，其中②③⑥⑦是为了辅助客户端的服务发现和服务端的服务注册
- 客户端：①③④⑦⑧
- 服务端：①②⑥

## 四、注册中心

### 1. 消息格式定义，消息序列化和反序列化

实现思路：在 RPC 三端之间的所有交互消息都是通过 TCP 进行传输的。消息格式定义：所有的消息都是由长度和消息内容组成，并且消息内容用字符串存储；消息序列化：将消息长度和消息内容打包成整体发送；消息反序列化：先读取消息长度，再根据长度读取消息内容。消息结构如图所示：



关键代码：为了方便实现三端之间交互信息的传输，在三端中都封装了两个函数来辅助消息通信，函数 `makePacket` 用于将要发送的消息打包，函数 `extractMessage` 从接收到的数据包中提取消息，具体实现如下：

```
145 std::vector<char> RegistrationCenter::makePacket(std::string message) {
146     int msgLen = message.size();
147
148     std::vector<char> packet(sizeof(int) + msgLen);
149     std::memcpy(packet.data(), &msgLen, sizeof(int));
150     std::memcpy(packet.data() + sizeof(int), message.data(), msgLen);
151
152     return packet;
153 }

82 bool RegistrationCenter::extractMessage(int sockfd, std::string &message) {
83     // 读取消息头，消息头是 4 字节，表示消息体的长度
84     int msgLen;
85     int n = read(sockfd, &msgLen, sizeof(msgLen));
86     if (n < 0) {
87         std::cout << "ERROR reading message length" << std::endl;
88         close(sockfd);
89         return false;
90     }
91     // msgLen = ntohs(msgLen); // 将消息长度从网络字节序转换为主机字节序 no need
92
93     // 根据消息长度读取消息体
94     std::vector<char> buffer(msgLen);
95     n = read(sockfd, buffer.data(), msgLen);
96     if (n < 0) {
97         std::cout << "ERROR reading message body" << std::endl;
98         close(sockfd);
99         return false;
100     }
101
102     message = std::string(buffer.begin(), buffer.end());
103     return true;
104 }
```

客户端和服务端的实现与上述代码相同。

## 2. 服务注册

实现思路：在注册中心有一个专门用于处理服务端请求的函数 `handleServerRequest`，如果服务端的请求是“ServiceRegistration”，注册中心将进行服务注册操作，如果服务端的请求是“ServiceCancellation”，注册中心将进行服务注销操作，其余请求都是无效请求。其中服务信息是用 `unordered_map<std::string, Server>` 数据结构进行存储的，键是服务器 IP 地址，值是存储服务器信息的结构体或类。

关键代码如下：

```
106 void RegistrationCenter::handleServerRequest(int sockfd) {
107     std::string message;
108     if (extractMessage(sockfd, message) == false) {
109         std::cout << "Failed to read a server request" << std::endl;
110         close(sockfd);
111         return;
112     }
113
114     int idx = message.find(' ');
115     std::string request = message.substr(0, idx);
116     message = message.substr(idx + 1);
117     idx = message.find(' ');
118     std::string ip = message.substr(0, idx);
119     message = message.substr(idx + 1);
120     int port = std::stoi(message);
121     // std::cout << request << std::endl; // debug
122
123     mtx.lock();
124     if (request == std::string("ServiceRegistration")) {
125         servers[ip] = Server(ip, port);
126         std::vector<char> packet = makePacket("OK");
127         int n = write(sockfd, packet.data(), packet.size());
128         if (n < 0) {
129             std::cout << "Failed to send response to server" << std::endl;
130             close(sockfd);
131             return;
132         }
133         // std::cout << servers.size() << std::endl; // debug
134     } else if (request == std::string("ServiceCancellation") && servers.find(ip) != servers.end()) {
135         std::cout << "ServiceCancellation" << std::endl;
136         servers.erase(ip);
137     } else {
138         std::cout << "An illegal request from the server" << std::endl;
139     }
140     mtx.unlock();
141
142     close(sockfd);
143 }
```

## 3. 服务发现

实现思路：在注册中心有一个专门用于处理客户端请求的函数 `handleClientRequest`，如果客户端的请求是“ServiceDiscovery”，注册中心将所有服务端的具体信息打包成响应消息发送回客户端，其余客户端请求都是无效请求。

关键代码如下：

```
155 void RegistrationCenter::handleClientRequest(int sockfd) {
156     std::string message;
157     if (extractMessage(sockfd, message) == false) {
158         std::cout << "Failed to read a client request" << std::endl;
159         close(sockfd);
160         return;
161     }
162
163     if (message != std::string("ServiceDiscovery")) {
164         std::cout << message.size() << std::endl;
165         std::cout << std::string("ServiceDiscovery").size() << std::endl;
166         std::cout << "An illegal request from the client" << std::endl;
167         close(sockfd);
168         return;
169     }
170
171     std::string response;
172     for (auto& element : servers) {
173         response += element.second.toString();
174     }
175
176     if (response.size() == 0) {
177         response = "0";
178     }
179
180     std::vector<char> packet = makePacket(response);
181     int n = write(sockfd, packet.data(), packet.size());
182     if (n < 0) {
183         std::cout << "Failed to send response to client" << std::endl;
184         close(sockfd);
185         return;
186     }
187
188     close(sockfd);
189 }
190 }
```

#### 4. 支持并发

实现思路：注册中心每次启动后，用函数 `handleRequest` 创建两个线程，一个线程在固定端口 10000 监听客户端请求，一个线程在固定端口 10001 监听服务端请求。

关键代码如下：函数 `run` 用于创建并启动线程，函数 `handleRequest` 封装了具体处理逻辑。

```
27 void RegistrationCenter::run() {
28     std::cout << "Registration center start..." << std::endl;
29     std::thread serverThread([this, port = sport] { handleRequest(port, "Server Request"); });
30     std::thread clientThread([this, port = cport] { handleRequest(port, "Client Request"); });
31     serverThread.join();
32     clientThread.join();
33 }
```

```

35 void RegistrationCenter::handleRequest(int port, std::string requestType) {
36     int serverSockfd, clientSockfd;
37     socklen_t clientlen;
38     struct sockaddr_in serverAddr, clientAddr;
39
40     serverSockfd = socket(AF_INET, SOCK_STREAM, 0);
41     if (serverSockfd < 0) {
42         std::cout << requestType << ": Socket creation failed" << std::endl;
43         close(serverSockfd);
44         exit(1);
45     }
46
47     memset((char *) &serverAddr, 0, sizeof(serverAddr));
48     serverAddr.sin_family = AF_INET;
49     serverAddr.sin_addr.s_addr = INADDR_ANY;
50     serverAddr.sin_port = htons(port);
51
52     if (bind(serverSockfd, (struct sockaddr *) &serverAddr, sizeof(serverAddr)) < 0) {
53         std::cout << requestType << ": Failed to bind to port " << port << std::endl;
54         close(serverSockfd);
55         exit(1);
56     }
57
58     listen(serverSockfd, 5);
59     clientlen = sizeof(clientAddr);
60
61     while (true) {
62         clientSockfd = accept(serverSockfd, (struct sockaddr *) &clientAddr, &clientlen);
63         if (clientSockfd < 0) {
64             close(clientSockfd);
65             continue;
66         }
67
68         if (requestType == std::string("Server Request")) {
69             handleServerRequest(clientSockfd);
70             // std::cout << servers.size() << std::endl; // debug
71             // for (auto &element: servers) {
72             //     std::cout << element.second.toString();
73             // }
74         } else {
75             handleClientRequest(clientSockfd);
76         }
77     }
78
79     close(serverSockfd);
80 }

```

## 5. 异常处理及超时处理

实现思路：没有单独封装处理异常和超时的逻辑，而是将异常和超时处理分散在了各个函数实现中，一遇到异常或超时错误，立即进行处理。

## 五、客户端

### 1. 消息格式定义，消息序列化和反序列化

已在注册中心部分进行说明，不再赘述

### 2. 服务发现

实现思路：由于实现具体服务（即函数）的发现和注册对于个人来说开发难度大，因此我假设了一个场景，即客户端知道服务端提供了哪些服务（就像我们可以通过文档查看C++STL 的用法那样，不用再与C++STL 库交互来获得使用方法），于是注册中心只保存提供这些服务的服务器的地址和端口，因此客户端服务发现就是获取这些服务器 的 IP 地址和对应 的端 口 。我在客户端 中封装 了 函数 discovery用于实现服务发现，同时这个函数也用于服务更新。

关键代码如下：



```

22 bool RPCClient::discovery() {
23     bool res = establishConnection(dsockfd, rIPAddr, rport, dport);
24     if (res == false) {
25         std::cout << "Failed to connect to the registry" << std::endl;
26         close(dsockfd);
27         return false;
28     } else {
29         std::cout << "Connected to the registry successfully\n";
30         // send request
31         std::vector<char> request = makePacket("ServiceDiscovery");
32         int n = write(dsockfd, request.data(), request.size());
33         if (n < 0) {
34             std::cout << "Failed to send response to client" << std::endl;
35             close(dsockfd);
36             return false;
37         }
38         // receive response
39         std::string response;
40         if (extractMessage(dsockfd, response) == false) {
41             close(dsockfd);
42             return false;
43         }
44         if (response == std::string("0")) {
45             std::cout << "No registered service" << std::endl;
46             close(dsockfd);
47             return false;
48         }
49         processRResponse(response);
50         close(dsockfd);
51
52         return true;
53     }
54 }

```

### 3. 服务调用

实现思路：客户端等待用户的输入，将用户输入打包成请求发送给经负载均衡函数选定的服务端，接收到服务端的响应后，直接将响应结果输出到终端。如果服务调用连续失败 3 次，客户端将进行服务更新，如果服务更新连续失败两次，客户端程序将强制退出。

关键代码：函数 call 封装了服务调用逻辑

```

56 void RPCClient::call() {
57     int counter1 = 0;
58     int counter2 = 0;
59     while (true) {
60         // Load balancing, server selection
61         int idx = balance();
62         if (idx == -1) {
63             std::cout << "Serverless service provision" << std::endl;
64             exit(1);
65         }
66         // Establish a connection with the server
67         std::string serverIP = servers[idx].IPAddress;
68         int sport = servers[idx].port;
69         bool res = establishConnection(csockfd, (const char [33])"Failed to connect to the server ");
70         if (res == false) {
71             std::cout << "Failed to connect to the server " << serverIP << std::endl;
72             close(csockfd);
73             ++counter1;
74             if (counter1 >= 3) {
75                 counter1 = 0;
76                 ++counter2;
77                 if (counter2 >= 2) {
78                     std::cout << "The remote procedure call failed 6 times in a row, exiting" << std::endl;
79                     exit(1);
80                 }
81                 std::cout << "The connection establishment failed 3 times in a row. Update the service" << std::endl;
82                 if (discovery() == false) {
83                     std::cout << "Service update failed" << std::endl;
84                     exit(1);
85                 }
86             }
87             continue;
88         }
89         counter1 = 0;
90         counter2 = 0;

```

```

91 // Waiting for customer input
92 std::cout << "Call the remote procedure like this: funcName parameter1 parameter2..." << std::endl;
93 std::string request;
94 getline(std::cin, request);
95 if (request == std::string("quit")) {
96     std::vector<char> packet = makePacket(request);
97     int n = write(csockfd, packet.data(), packet.size());
98     close(csockfd);
99     close(dsockfd);
100     exit(0);
101 }
102 // send request
103 std::vector<char> packet = makePacket(request);
104 int n = write(csockfd, packet.data(), packet.size());
105 if (n < 0) {
106     std::cout << "Failed to send remote procedure call request" << std::endl;
107     close(csockfd);
108     continue;
109 }
110 // Receive Response
111 std::string response;
112 if (extractMessage(csockfd, response) == false) {
113     close(csockfd);
114     continue;
115 }
116 // std::cout << response << std::endl; // debug
117 close(csockfd);
118 processSResponse(response);
119 }
120 }

```

#### 4. 异常处理及超时处理

实现思路：没有单独封装处理异常和超时的逻辑，而是将异常和超时处理分散在了各个函数实现中，一遇到异常或超时错误，立即进行处理。已经处理的异常和超时情况如下（包括但不限于）：

- 与服务端建立连接时产生的异常/超时
- 发送请求到服务端，写数据时出现的异常/超时
- 等待服务端处理时，等待处理导致的异常/超时（比如服务端已挂死，迟迟不响应）
- 从服务端接收响应时，读数据导致的异常/超时

#### 5. 负载均衡

实现思路：客户端的负载均衡用轮询算法实现。

关键代码：

```

217 int RPCClient::balance() {
218     int idx = -1;
219     if (servers.size() == 0) {
220         return idx;
221     }
222     idx = selection;
223     selection = (selection + 1) % servers.size();
224     return idx;
225 }

```

## 六、服务端

### 1. 消息格式定义，消息序列化和反序列化

已在注册中心部分进行说明，不在赘述。

### 2. 服务注册

实现思路：服务端通过辅助函数 getEns33 获取本地 IP 地址，将 IP 地址和用于客户端服务调用的固定端口 20001 打包，在固定端口 20000 发送给注册中心进行服务注册。

为了降低开发难度，服务端不支持心跳，而是用辅助函数 handleSignal 接收 Ctrl + C 或 Ctrl + Z 信号，当接收到信号时，服务端向注册中心发送“ServiceCancellation”请求来注销服务。

关键代码如下：

```

48 std::string RPCServer::getEns33() {
49     int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
50     struct ifreq ifr;
51     std::memset(&ifr, 0, sizeof(ifr));
52     std::strncpy(ifr.ifr_name, "ens33", IFNAMSIZ - 1); // 指定网卡接口名
53     std::string ip;
54
55     if (ioctl(sockfd, SIOCGIFADDR, &ifr) == 0) {
56         struct sockaddr_in *addr = (struct sockaddr_in *)&ifr.ifr_addr;
57         char ipstr[INET_ADDRSTRLEN];
58         inet_ntop(AF_INET, &addr->sin_addr, ipstr, sizeof(ipstr));
59         ip = std::string(ipstr);
60     }
61
62     close(sockfd);
63     return ip;
64 }

```



```

91 bool RPCServer::registe() {
92     bool res = establishConnection(rsockfd, rIPAddr, rport, gport);
93     if (res == false) {
94         std::cout << "Failed to connect to the registry" << std::endl;
95         close(rsockfd);
96         return false;
97     } else {
98         std::cout << "Connected to the registry successfully" << std::endl;
99         // send request
100         std::string request = std::string("ServiceRegistration ") + IPAddr + " " + std::to_string(cport);
101         std::vector<char> packet = makePacket(request);
102         int n = write(rsockfd, packet.data(), packet.size());
103         if (n < 0) {
104             std::cout << "Failed to send service registration request" << std::endl;
105             close(rsockfd);
106             return false;
107         }
108         // receive response
109         std::string response;
110         if (extractMessage(rsockfd, response) == false) {
111             close(rsockfd);
112             return false;
113         }
114         return processRResponse(response);
115     }
116 }

66 void RPCServer::handleSignal(int signal) {
67     bool res = establishConnection(rsockfd, rIPAddr, rport, gport);
68     if (res == false) {
69         std::cout << "Failed to connect to the registry" << std::endl;
70         close(rsockfd);
71     } else {
72         std::cout << "Connected to the registry successfully" << std::endl;
73         // send request
74         std::string request = std::string("ServiceCancellation ") + IPAddr + " " + std::to_string(cport);
75         std::vector<char> packet = makePacket(request);
76         int n = write(rsockfd, packet.data(), packet.size());
77         if (n < 0) {
78             std::cout << "Failed to send service cancellation request" << std::endl;
79             close(rsockfd);
80         }
81         // receive response
82         std::string response;
83         if (extractMessage(rsockfd, response) == false) {
84             close(rsockfd);
85         }
86     }
87     close(csockfd);
88     exit(0);
89 }

```

### 3. 支持并发

实现思路：服务端在固定端口 20001 监听客户端请求，每接收一个请求，就创建并运行一个线程来处理客户端请求。

关键代码：

```

23 void RPCServer::run() {
24     // Service Registration
25     if (registe() == false) {
26         std::cout << "Service registration failed" << std::endl;
27         close(rsockfd);
28         exit(1);
29     }
30     close(rsockfd);
31     // Waiting for client request
32     if (acceptConnection() == false) {
33         close(csockfd);
34         exit(1);
35     }
36
37     while (true) {
38         int clientSockfd = accept(csockfd, nullptr, nullptr);
39         if (clientSockfd < 0) {
40             std::cout << "Accept failed" << std::endl;
41             continue;
42         }
43         std::thread clientThread([this, sockfd = clientSockfd] { handleRequest(sockfd); });
44         clientThread.detach();
45     }
46 }

```

```

194 bool RPCServer::acceptConnection() {
195     csockfd = socket(AF_INET, SOCK_STREAM, 0);
196     if (csockfd < 0) {
197         std::cout << "Socket creation error" << std::endl;
198         return false;
199     }
200
201     struct sockaddr_in server_addr;
202     server_addr.sin_family = AF_INET;
203     server_addr.sin_addr.s_addr = INADDR_ANY;
204     server_addr.sin_port = htons(cport);
205
206     if (bind(csockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
207         std::cout << "Bind failed" << std::endl;
208         close(csockfd);
209         return false;
210     }
211
212     if (listen(csockfd, 10) < 0) {
213         std::cout << "Listen failed" << std::endl;
214         close(csockfd);
215         return false;
216     }
217
218     std::cout << "Server is listening on port " << cport << std::endl;
219     return true;
220 }

```

```

222 void RPCServer::handleRequest(int clientSockfd) {
223     // std::cout << "handleRequest1" << std::endl; // debug
224     std::string request;
225     std::string result;
226     std::string funcName;
227     std::string arguments;
228     if (extractMessage(clientSockfd, request) == false) {
229         close(clientSockfd);
230         return;
231     }
232     // std::cout << request << std::endl; // debug
233     // std::cout << "handleRequest2" << std::endl; // debug
234     int idx = request.find(' ');
235     if (idx != std::string::npos) {
236         funcName = request.substr(0, idx);
237         arguments = request.substr(idx + 1);
238     }
239
240     // std::cout << funcName << " " << arguments << std::endl; // debug
241     if (funcName == std::string("add")) {
242         Add add(arguments);
243         result = add();
244     } else if (funcName == std::string("sub")) {
245         Sub sub(arguments);
246         result = sub();
247     } else if (funcName == std::string("mul")) {
248         Mul mul(arguments);
249         result = mul();
250     } else if (funcName == std::string("div")) {
251         Div div(arguments);
252         result = div();
253     } else if (funcName == std::string("sqr")) {
254         Sqr sqr(arguments);

```

```

255         result = sqr();
256     } else if (funcName == std::string("sqrt")) {
257         Sqrt sqrt(arguments);
258         result = sqrt();
259     } else if (funcName == std::string("cube")) {
260         Cube cube(arguments);
261         result = cube();
262     } else if (funcName == std::string("echo")) {
263         Echo echo(arguments);
264         result = echo();
265     } else if (funcName == std::string("reverse")) {
266         Reverse reverse(arguments);
267         result = reverse();
268     } else if (funcName == std::string("fact")) {
269         Fact fact(arguments);
270         result = fact();
271     } else if (funcName == std::string("quit")) {
272         close(clientSockfd);
273         return;
274     } else {
275         result = "No such function or invalid request";
276     }
277     // std::cout << "handleRequest3" << std::endl; // debug
278     std::vector<char> packet = makePacket(result);
279     int n = write(clientSockfd, packet.data(), packet.size());
280     if (n < 0) {
281         std::cout << "Failed to send rpc result" << std::endl;
282         close(clientSockfd);
283     }
284     // std::cout << "handleRequest4" << std::endl; // debug
285 }

```

如代码所示，服务端总共有 10 个可调用服务：add（加法）、sub（减法）、mul（乘法）、div（除法）、sqr（平方）、sqrt（开方）、cube（立方）、echo（回显）、reverse（字符串反转）、fact（阶乘）。其中所有的运算都是针对整数，每一个服务都被封装在一个类中，每个类自行检查参数的合法性。如果需要添加服务，在 Function.hpp 文件中编辑即可。

#### 4. 异常处理及超时处理

实现思路：没有单独封装处理异常和超时的逻辑，而是将异常和超时处理分散在了各个函数实现中，一遇到异常或超时错误，立即进行处理。已经处理的异常和超时情况如下（包括但不限于）：

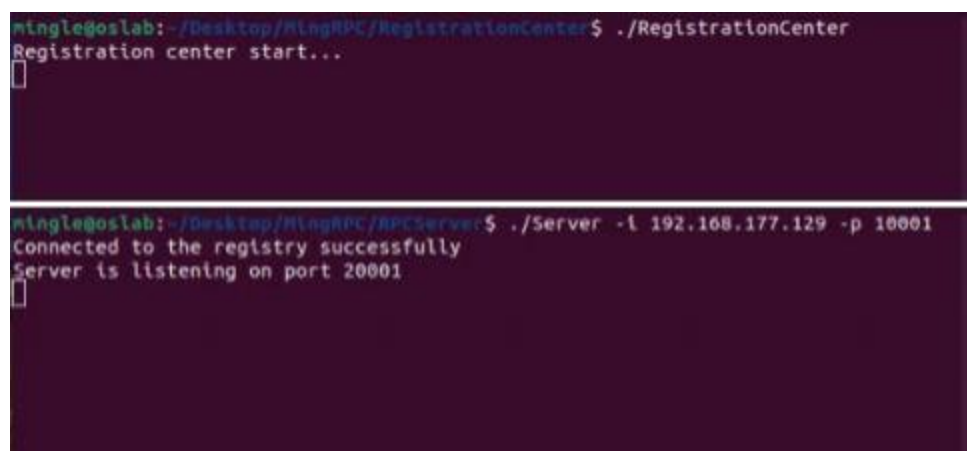
- 读取客户端请求数据时，读数据导致的异常/超时
- 发送响应数据时，写数据导致的异常/超时
- 调用映射服务的方法时，处理数据导致的异常/超时

## 七、测试

1. 测试环境：本地虚拟机 A 192.168.177.129（Ubuntu22.04）、本地虚拟机 B 192.168.177.130（Ubuntu20.04）。

2. 测试过程：

（1）在虚拟机 A 中先后启动一个注册中心和一个服务端



```
minglegoslab:~/Desktop/MinigRPC/RegistrationCenter$ ./RegistrationCenter
Registration center start...
□

minglegoslab:~/Desktop/MinigRPC/RPCServer$ ./Server -l 192.168.177.129 -p 10001
Connected to the registry successfully
Server is listening on port 20001
□
```

（2）在虚拟机 B 中启动一个服务端

```

mingle@ubuntu:~/Desktop/MinglerRPC/RPCServer$ ./Server -i 192.168.177.129 -p 10001
Connected to the registry successfully
Server is listening on port 20001

```

(3) 在虚拟机 A 中启动一个客户端，并进行服务调用测试

```

mingle@oslab:~/Desktop/MinglerRPC/RPCClient$ make
g++ -g -std=c++14 -o Client main.cpp CheckArguments.cpp RPCClient.cpp
mingle@oslab:~/Desktop/MinglerRPC/RPCClient$ ./Client -i 192.168.177.129 -p 10000
Connected to the registry successfully
Call the remote procedure like this: funcName parameter1 parameter2...
add 123 321
Result: 444
Call the remote procedure like this: funcName parameter1 parameter2...
reverse Hello, world!!!
Result: !!!dlrow ,olleH
Call the remote procedure like this: funcName parameter1 parameter2...
fact 3
Result: 6
Call the remote procedure like this: funcName parameter1 parameter2...

```

(4) 在虚拟机 B 中启动一个客户端，并进行服务调用测试

```

mingle@ubuntu:~/Desktop/MinglerRPC/RPCClient$ make
g++ -g -std=c++14 -o Client main.cpp CheckArguments.cpp RPCClient.cpp
mingle@ubuntu:~/Desktop/MinglerRPC/RPCClient$ ./Client -i 192.168.177.129 -p 10000
Connected to the registry successfully
Call the remote procedure like this: funcName parameter1 parameter2...
add 345 543
Result: 888
Call the remote procedure like this: funcName parameter1 parameter2...
cube 4
Result: 64
Call the remote procedure like this: funcName parameter1 parameter2...
fact 6
Result: 720
Call the remote procedure like this: funcName parameter1 parameter2...
div 9 3
Result: 3
Call the remote procedure like this: funcName parameter1 parameter2...

```

## 八、运行教程

### 1. 注册中心

(1) 在 RegistrationCenter 目录下运行 make 命令，得到 RegistrationCenter 可执行程序；

(2) 运行 ./RegistrationCenter 命令启动注册中心；

(3) 键入 Ctrl+C 或 Ctrl+Z 终止程序。

## 2. 客户端

(1) 在 RPCClient 目录下运行 make 命令，得到 RPCClient 可执行程序；

(2) 运行 ./Client -h 命令获取帮助；

(3) 运行 ./Client -i ip -p port 命令启动客户端，其中 ip 指的是注册中心的 ip 地址，port 指的是注册中心的端口号，端口号固定为 10000；

(4) 键入 Ctrl+C 或 Ctrl+Z 终止程序。

## 3. 服务端

(1) 在 RPCServer 目录下运行 make 命令，得到 RPCServer 可执行程序；

(2) 运行 ./Server -h 命令获取帮助；

(3) 运行 ./Server -i ip -p port 命令启动服务端，其中 ip 指的是注册中心的 ip 地址，port 指的是注册中心的端口号，端口号固定为 10001；

(4) 键入 Ctrl+C 或 Ctrl+Z 终止程序。

4. 运行过程：应当先启动注册中心，再启动服务端，最后启动客户端