CS328 · Distributed and Cloud Computing

Assignment 2

刘乐奇 / 12011327

April 9, 2023

1 Introduction

RPC (remote procedure call) is a technology that allows a client application to make requests to a server application running on a different machine over a network. The client and server communicate through a communication protocol, such as HTTP or TCP/IP, and the server processes the request and returns a response to the client. RPC allows applications to be distributed across multiple machines and enables developers to build distributed systems that are more scalable and flexible.
RMI (remote method invocation) is a Java-specific technology that allows Java objects to be invoked remotely, as if they were local objects. RMI works by exposing Java objects as remote services, which can be invoked by remote clients using a standard Java interface. RMI provides a seamless integration with Java and allows Java objects to be shared and used across different machines, making it a powerful tool for building distributed Java applications.

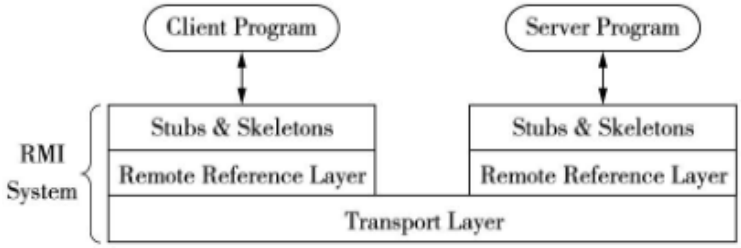

Figure 1: Remote Method Invocation (RMI)

2 Prerequisite

2.1 Proxy, Stub and Skeleton

Proxy design pattern, which allows an object to act as a stand-in for another object, is a significant thought. In the context of Java RMI, the proxy design pattern is used to allow a client to interact with a remote object as if it were a local object.
In Java RMI, a *stub* is a client-side proxy for a remote object, and a *skeleton* is a server-side proxy for a remote object. The stub and skeleton work together to enable communication between the client and server.
When a client invokes a method on a stub, the stub marshals the arguments and sends them over the network to the server. The server receives the message, unmarshals the arguments, and invokes the method on the skeleton. The skeleton marshals the result and sends it back over the network to the client, which unmarshals the result and returns it to the caller.
The stub and skeleton work together to provide a transparent mechanism for remote method invocation in Java RMI, hiding the complexities of network communication and serialization from the programmer.

2.2 Serialization and Deserialization

Serialization and deserialization are often used in distributed systems, where data needs to be transmitted over a network between different nodes in a system. By serializing an object and transmitting it as a stream

of bytes, the object can be reconstructed on another node and used in a program running on that node. There are different serialization and deserialization formats and techniques, such as Java serialization, JSON serialization, and Protobuf serialization. In this assignment, I choose Protobuf serialization since it has lightweight and efficient format and is interoperable with different programming languages and platforms.

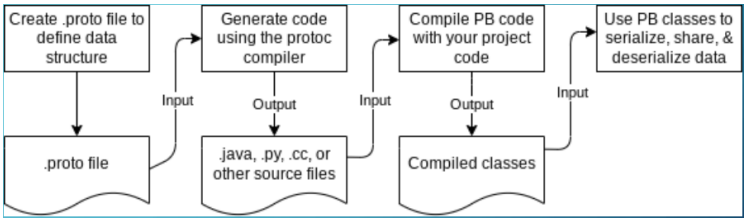

Figure 2: Protobuf workflow

3 Design

I have designed 2 versions, among which one is a basic implementation and the other utilizes *ZooKeeper* ensuring load balancing and high availability.

3.1 Basic Design

3.1.1 Description of packages and files

This version is in the package $myrmi$. There are 4 packages and a $.java$ file in it:

- $exception$: Some self-defined exceptions that may be thrown in the invocation of methods.
- $info$: The formats of request and result messages that will be transmitted during the communication between client and server (more specifically, stub and skeleton).
- $registry$: The implementation of registry and the handler for stub to invoke remote method.
- $server$: For the creation of skeleton, stub, and the handlers.
- $Remote.java$: A java *interface*. Every remote object should implement it.

3.1.2 How to use

1. First, we should define a *class* or *interface* which implements or extends the interface *Remote*. If an interface is defined here, then we should create a concrete *class* that implements it and extends $UnicastRemoteObject$ in $server$ package with some concrete methods that will be invoked in remote.
   However, if the concrete *class* has extended another *class* or if we do not want the concrete *class* to extend $UnicastRemoteObject$ in case of the extension in the future, we can export the remote object by using the static method $UnicastRemoteObject.exportObject()$.
   Here is an example [Figure 3] (in package *tests/publicFile*).
2. Then on the server side, it will create a registry and bind the remote object on the registry. Since I had wrapped registry and its operations in a new *class* named $Naming$, we can use another equivalent way (notice the format and the port number). Correspondingly, on the client side, it will look up for the service and the invoke method transparently. Similarly, we can also use *Naming*. [Figure 4] (in package *tests/testMyRMI*)

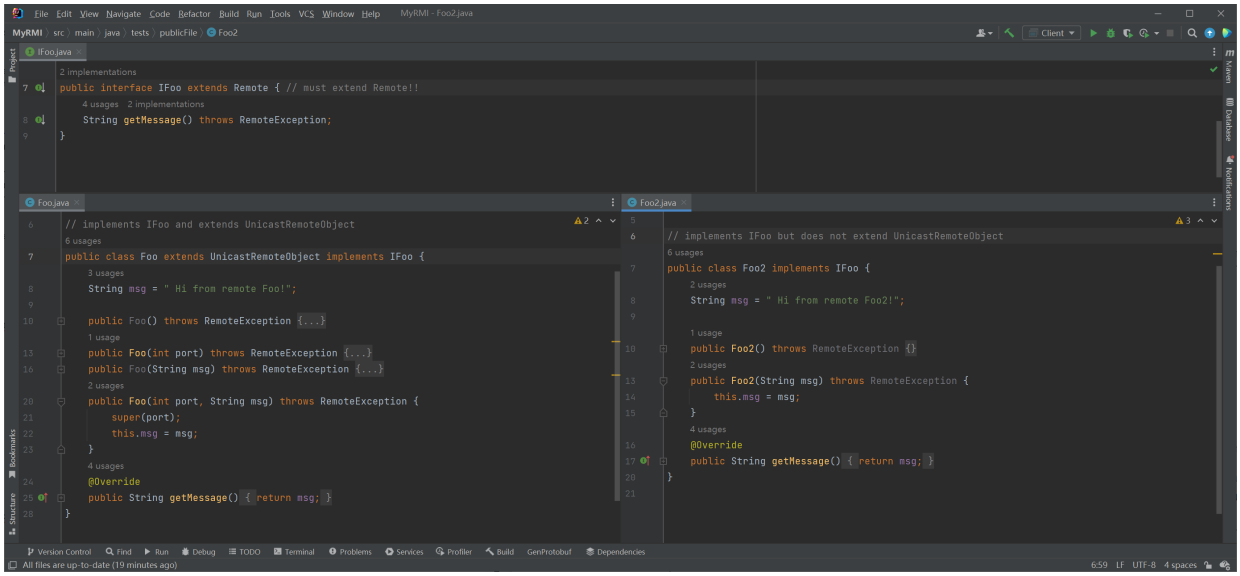

Figure 3: Remote Objects

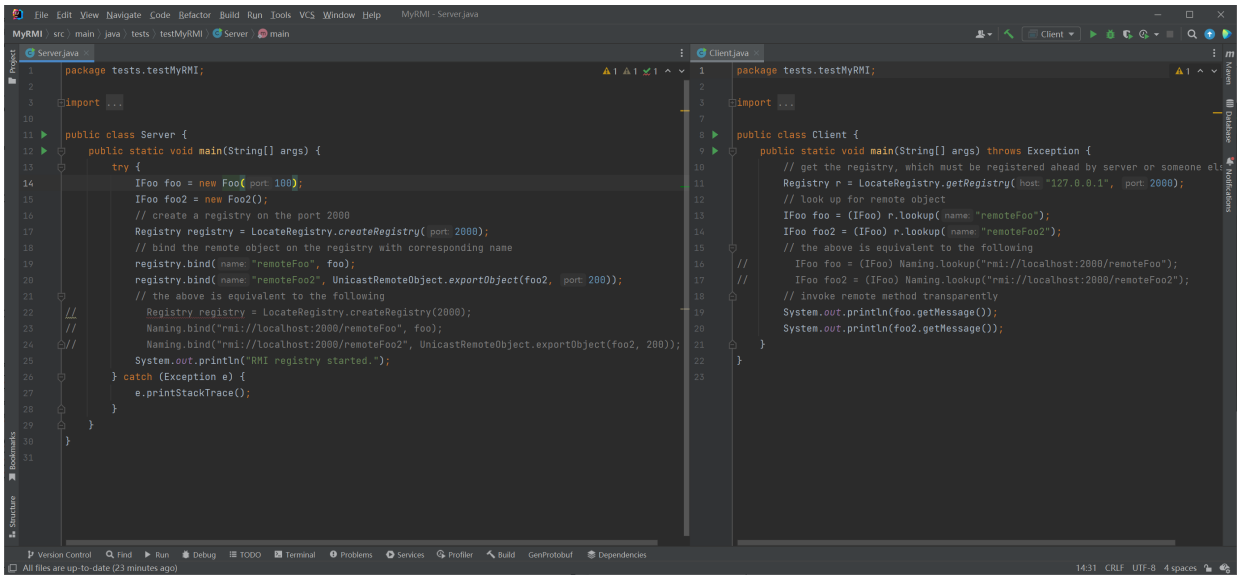

Figure 4: Server and Client

3. Notice that we should start server first then client, or the registry and stub will not be created so that client cannot get the stub from registry.

3.1.3 Significant issues

There are some noticeable design details.

- When server creates a registry, a skeleton, which extends *Thread*, will run. This skeleton will start a server socket for any incoming connection, and will create a thread pool for request handler. As soon as the incoming connection is accepted, a thread will be found in the thread pool or created for the request handler.

```
// ...
// Thread pool is created
ExecutorService executorService = new ThreadPoolExecutor(10, 10, 60L, TimeUnit.
    SECONDS, new ArrayBlockingQueue<>(10));
ServerSocket serverSocket = null;
```

```
try {
    InetAddress address = InetAddress.getByName(host);
    serverSocket = new ServerSocket(port, BACKLOG, address);
    this.port = serverSocket.getLocalPort();
    while (true) {
        // blocked until accepting the incoming connection,
        Socket client = serverSocket.accept();
        System.out.printf("[R_S]>> Request from port: %d\n", client.getPort());
        // submits a Runnable task (request handler) for execution
        Runnable clientRequest = new RequestProtoHandler(client, remoteObj, objectKey);
        executorService.submit(clientRequest);
    }
}
// ...
```

- There are some deprecated *classes* in *myrmi*. It is because that I used Java serialization for request and result information at the first time. No sooner I turned to *Protobuf* instead. It was running well in my local test. Here is the *.proto* files (under the *resources* directory in *proto* package). It needs to be noticed that all the *args* should implements the interface *java.io.Serializable*.

```
// RequestInfo.proto
syntax = "proto3";

option java_package = "myrmi.Info";
option java_outer_classname = "RequestInfoProto";

package myrmi.Info;
message RequestInfo {
  string method_name = 1;
  repeated bytes args = 2;
  repeated string arg_types = 3;
  int32 object_key = 4;
}
```

```
// ResultInfo.proto
syntax = "proto3";

option java_package = "myrmi.Info";
option java_outer_classname = "ResultInfoProto";

package myrmi.Info;
message ResultInfo {
  bytes result = 1;
  bytes Exception = 2;
  int32 status = 3;
  int32 object_key = 4;
}
```

3.2 ZooKeeper for Load Balancing and High Availability

3.2.1 Introduction

Although the basic design tests well, there are still some drawbacks.

1. Basic design has a poor encapsulation.
2. Clients have no idea about the server. If server's load is too heavy, it may crash.
3. The RMI server might inevitably fail during running. If the RMI service is no longer connected, the client will be unable to respond.

In case of the drawbacks, we need to equip the system with load balancing and high availability. So I integrated ZooKeeper into MyRMI. Server side will publish services and client side will consume the published services.

3.2.2 Description of packages and files

This version is in the package *RMIZK*. There are 3 *.java* file in it:

- *Constant.java*: A java *interface*, containing some public constant values.
- *ServiceProvider.java*: Publish RMI service, and then register the service address into ZooKeeper. Servers are listened by ZooKeeper so that when server crashes, the server's services will be remove from ZooKeeper.
- *ServiceConsumer.java*: Listen to ZNodes in ZooKeeper and renew the service list. Consumer can get the wanted service.

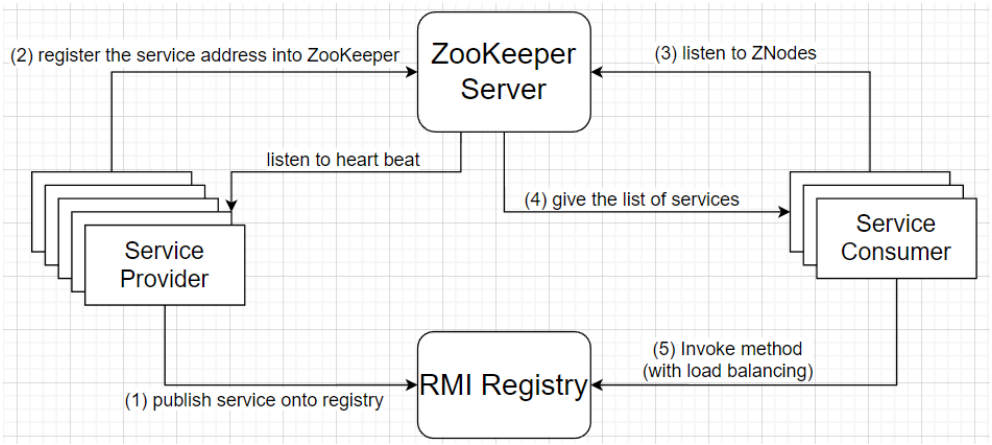

Figure 5: Provider-Consumer workflow

3.2.3 How to use

1. The first step is the same as basic design: some *class* or *interface* must be defined. Here I still used the *.java* files written in basic design.
2. Do not forget to start ZooKeeper server first! Or the clients cannot get the address of services.
3. On the server side, it needs to publish some services. On the client side, it will look up for the service and the invoke method transparently. We can choose the look up mode, either random or least connection. The details of create registry, bind services and look up for services are concealed. It is a better encapsulation. [Figure 6 & 7] (in package *tests/testZK*)

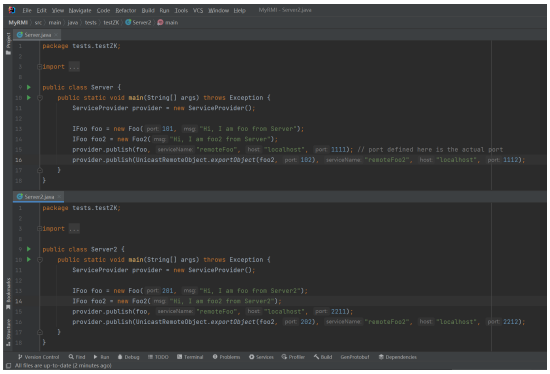

Figure 6: Server side

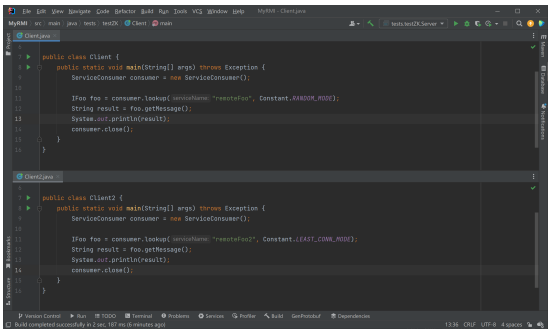

Figure 7: Client side

3.2.4 Significant issues

There are some noticeable design details.

- For simplification, I did not provide any method or interface for server (service provider) to cancel its service (unpublish). Only when server crashes or close normally, can the services provided by this server are all deregistered from ZooKeeper at a time.
- I designed 2 policies to ensure load balancing, both implemented in *ServiceConsumer.java*. One is to randomly allocate consume to services, the other is to allocate consume to the server with least connections. Both can be seen in *RMIZK/ServiceConsumer.java*.
  In *ServiceConsumer*, we can see there are 3 fields and a nested class, which are used to handle getting service. *serverMap* and *serviceMap* are static since they are the global variables to handle all the behaviours of consumers. *serverMap* is the map from *serverHost* (the host of server that provides the service) to *serverService* (a self-defined class), indicating what services the server can provide (*ServerService::srevices*), and how many connections on the server (*busyNum*). *serviceMap* is the map from *service* (service address) to *serverHost* (the host of server), which can be used to quickly find which server provides the service. *conn* indicates what servers are connecting with this client, which is important in least connection policy.

```
public class ServiceConsumer {
    /* ... */
    private static final Map<String, ServerService> serverMap = new HashMap<>();
    private static final Map<String, String> serviceMap = new HashMap<>();
    private final List<String> conn = new ArrayList<>();
    /* ... */
    private static class ServerService {
        String serverHost;
        List<String> services;
        int busyNum;
        /* ... */
    }
}
```

If the consumer does not specify the service name, the service will randomly chosen from all published services. If one service is chosen, then its *busyNum* is increased by 1 and the *serverHost* is added into the current consumer's connection.

```
public <T extends Remote> T lookup()
        throws NotBoundException, ExecutionException, InterruptedException {
    T service = null;
    int size = serverMap.size();
    if (size > 0) {
```

```
            // randomly choose service
            List<String> serviceList = new ArrayList<>(serviceMap.keySet());
            int idx = ThreadLocalRandom.current().nextInt(serviceList.size());
            String url = serviceList.get(idx);

            System.out.printf("[_C]>> using url: %s\n", url);
            service = lookupService(url);

            if (service != null) {
                String serverHost = serviceMap.get(url);
                serverMap.get(serverHost).busyNum++;
                conn.add(serverHost);
            }
        }
        // 1. no server
        // 2. has server but did not find any service
        if (service == null)
            throw new NotBoundException();
        return service;
    }
```

If the consumer specifies the service name, he needs to specify the load balancing mode too. There are only 2 modes now: RANDOM_MODE = 0, LEAST_CONN_MODE = 1. Previous one is random policy, former one is least connection policy.

```
public <T extends Remote> T lookup(String serviceName, int mode)
        throws NotBoundException, ExecutionException, InterruptedException {
    return (mode == Constant.RANDOM_MODE) ? lookupRandom(serviceName) :
        lookupLeast(serviceName);
}
```

- Random policy is much analogous with the do-not-specify-service-name-method above. The difference is that it must first filter out the services with corresponding service name.
- Least connection policy relies on *ServerService::busyNum*. After filter out the services with corresponding service name, it will find out the server with the least connections, i.e., the least *ServerService::busyNum*.

To ensure the least connection policy working well, client must invoke $consumer.close()$ to awoke that the connection is over before its end, and then decrease the *busyNum* of the corresponding server.

```
public void close(){
    for (String serverHost : conn) {
        serverMap.get(serverHost).busyNum--;
    }
}
```

4 Running Result

All the test files can be found in the package *tests*.

4.1 Basic Design

test files' package: *tests/testMyRMI*
Server bound 2 remote object onto the registry. Client lookup for remote object and invoke methods. We can see that the client successfully invoke the methods and print out the messages. [Figure 8]

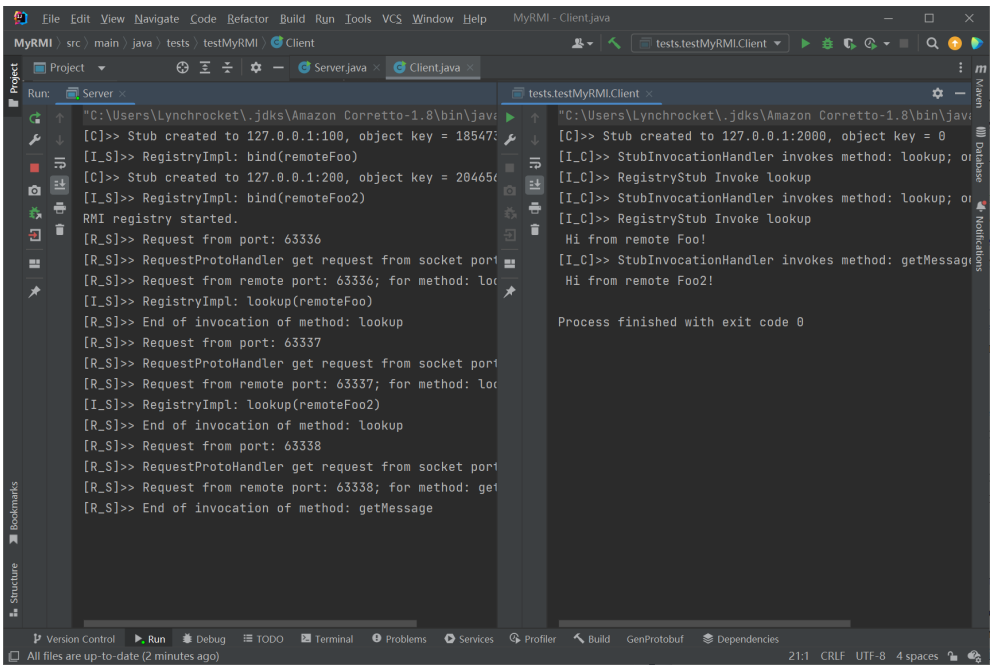

Figure 8: Result of Basic Design

4.2 ZooKeeper

test files' package: *tests/testZK*
Servers publish some services. We can see from the left of figure 9 that the addresses of services are successfully registered into ZooKeeper (this is a visual tool called ZooInspector). Red warning is because I had not configured the logger, which does not matter our goal.

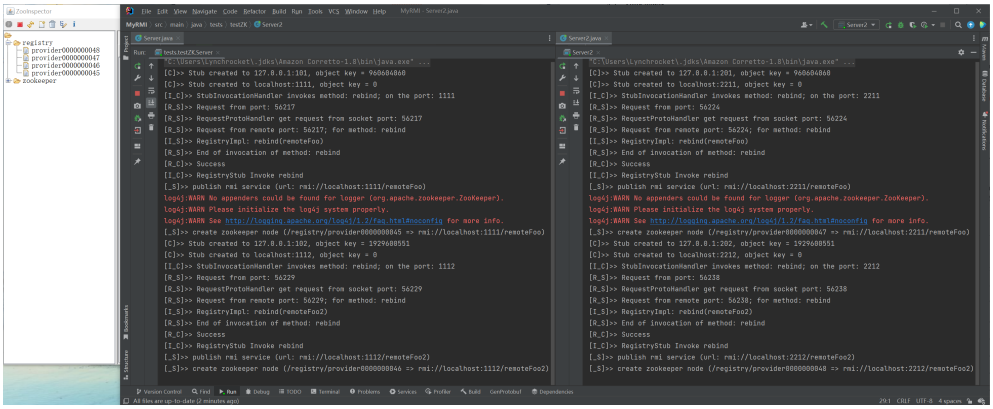

Figure 9: Successfully Publish Services

Clients consume the services successfully (both print out the messages). [Figure 10]

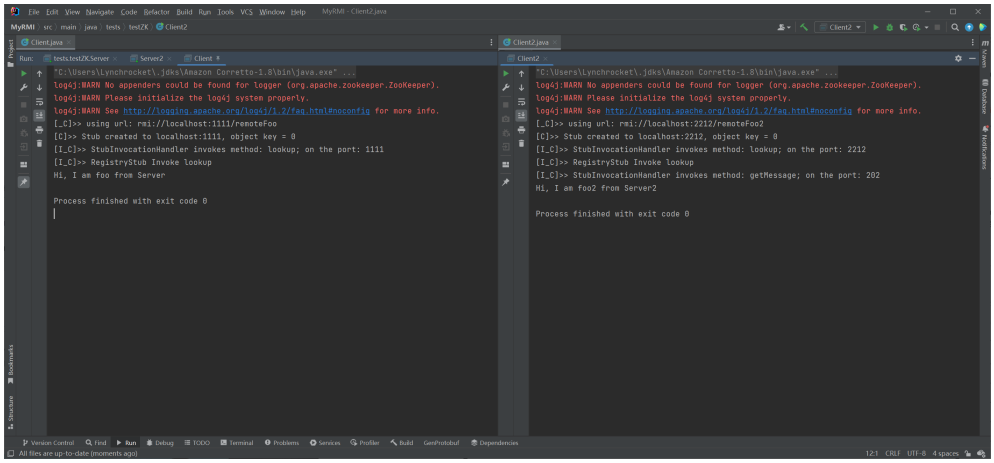

Figure 10: Successfully Consume Services

What if one of the servers crashes? (here I manually interrupted *Server2*) We can see from ZooInspector that 2 services vanished [Figure 11], which were originally the services provided by *Server2*. From figure 12 we can see that the services on *Server2* are on longer invoked.

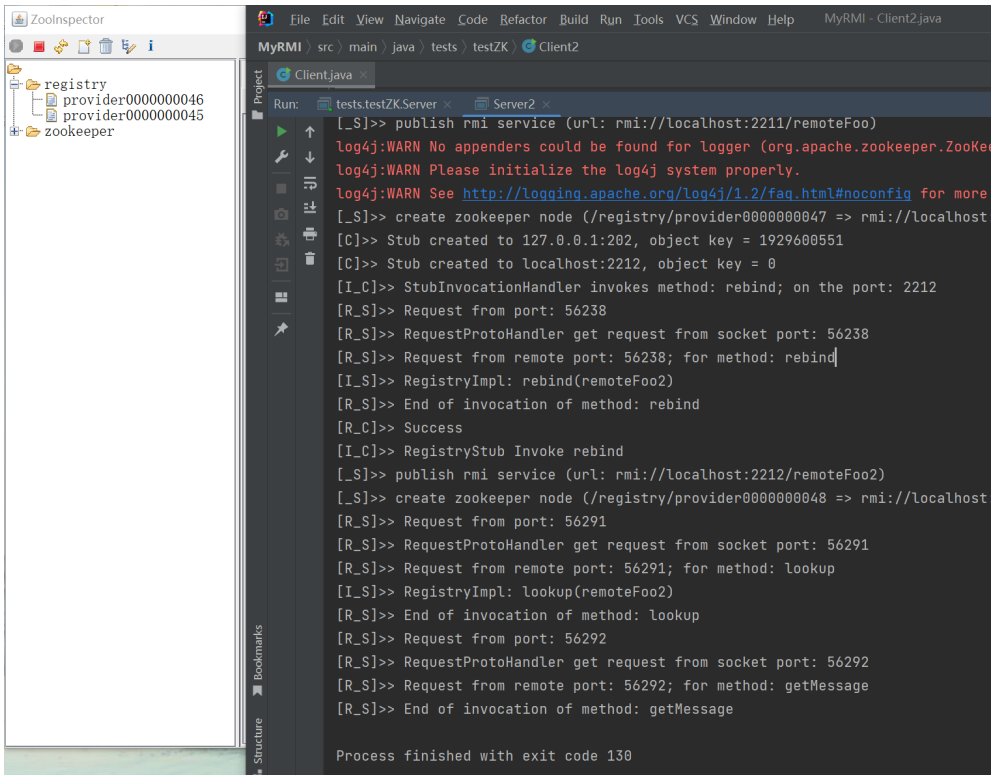

Figure 11: One server crashes

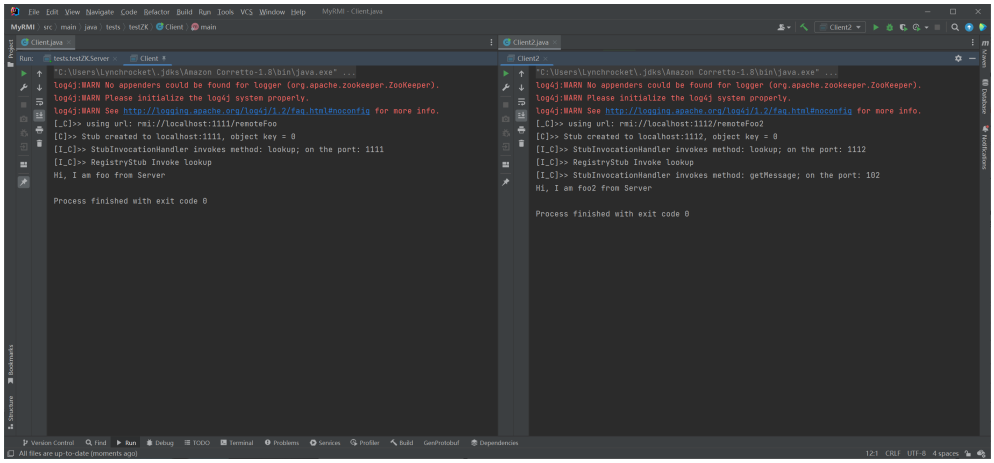

Figure 12: Consume services when one server crashes

5 Problems

5.1 Java I/O

There are plenty of choices in Java I/O. Stream is one of the most practical ones. However, there exists some confusing points when using it.

- Block stream: $socket.getInputStream()$ will be blocked until receive the message. Besides, in case that the stream is truncated, we can first send an integer indicating the stream length. And then the receiver side can receive the whole stream properly.

```
DataInputStream inputStream = new DataInputStream(client.getInputStream());
int length = inputStream.readInt();
byte[] buffer = new byte[length];
int read = 0;
while (read < length) {
    int n = inputStream.read(buffer, read, length - read);
    if (n < 0) {
        throw new EOFException("Unexpected end of input stream");
    }
    read += n;
}
ResultInfoProto.ResultInfo resultInfo = ResultInfoProto.ResultInfo.parseFrom(buffer);
```

- Flush stream: After the bytes are written into output stream, we should flush this output stream ($outputStream.flush()$) to force any buffered output bytes to be written out to the stream.

```
DataOutputStream outputStream = new DataOutputStream(client.getOutputStream());
outputStream.writeInt(requestInfo.getSerializedSize());
outputStream.write(requestInfo.toByteArray());
outputStream.flush();
```

5.2 Serialization and Deserialization with Protobuf

Protobuf (Protocol Buffers) is a language- and platform-neutral data serialization format developed by Google. It is efficient and powerful. However, it is not easy to serialize and deserialize the *Object* and *Class* type in Java. For example, if I need to serialize the class as below

```
public class RequestInfo{
```

```
    private String methodName;
    private Object[] args;
    private Class<?>[] argTypes;
    private int objectKey;
}
```

the corresponding *.proto* file is

```
syntax = "proto3";

option java_package = "myrmi.Info";
option java_outer_classname = "RequestInfoProto";

package myrmi.Info;
message RequestInfo {
  string method_name = 1;
  repeated bytes args = 2;
  repeated string arg_types = 3;
  int32 object_key = 4;
}
```

which means I need to manipulate the parameter $args$ and arg_types. More concretely, we should turn arguments from *Object* to *com.google.protobuf.ByteString* and turn arguments' type from *Class<?>* to *String* before transmit request, and get arguments from *com.google.protobuf.ByteString* and arguments' type from *String* after receiving the request. (the code fragments can be seen in *myrmi/server/RequestProtoHandler.java* and *myrmi/server/StubInvocationProtoHandler.java*)

5.3 Method invocation using reflection

This problem was found during test. The following example is in *tests/test/MIReflect.java*. Main part of the code fragments are from *myrmi/server/RequestProtoHandler.java*.
Assume there exists

```
public class Hello {
    public void invokeMe(String id, IFoo ifoo){/*...*/}
    public void invokeMe(String fake){/*...*/}
    public void invokeThou(String id, IFoo ifoo){/*...*/}
}
```

where

```
// People.java
public interface IFoo{/*...*/}
```

```
// Student.java
public class Foo implements IFoo{/*...*/}
```

Now we have got

```
String methodName = "invokeMe";
Object[] arg = {/*...*/};
Class<?>[] argTypes = {String.class, Foo.class};
```

and want to invoke the method $invokeMe(String\ id, IFoo\ ifoo)$ but neither $invokeMe(String\ fake)$ nor $invokeThou(String\ id, IFoo\ ifoo)$ by reflection. We cannot directly invoke like

```
// obj is an instance of class Hello
Method method = obj.getClass().getMethod(methodName, argTypes);
method.invoke(obj, arg);
```

since it will throw *NoSuchMethodException*. Instead, we should check that the corresponding parameter type in the method if assignable from argument type.

```
Method[] methods = obj.getClass().getMethods();
boolean foundMethod = false;
for (Method m : methods) { // iterate all public methods
    if (m.getName().equals(methodName)) {
        Class<?>[] parameterTypes = m.getParameterTypes();
        int n = parameterTypes.length;
        if (n != argTypes.length) continue;
        boolean flag = false;
        for (int i = 0; i < n; i++) {
            if (!parameterTypes[i].isAssignableFrom(argTypes[i])) { // check if assignable
                flag = true;
                break;
            }
        }
    if (flag) continue;                 // not assignable then next
    result = m.invoke(obj, arg);    // else invoke
    foundMethod = true;
    }
}
```

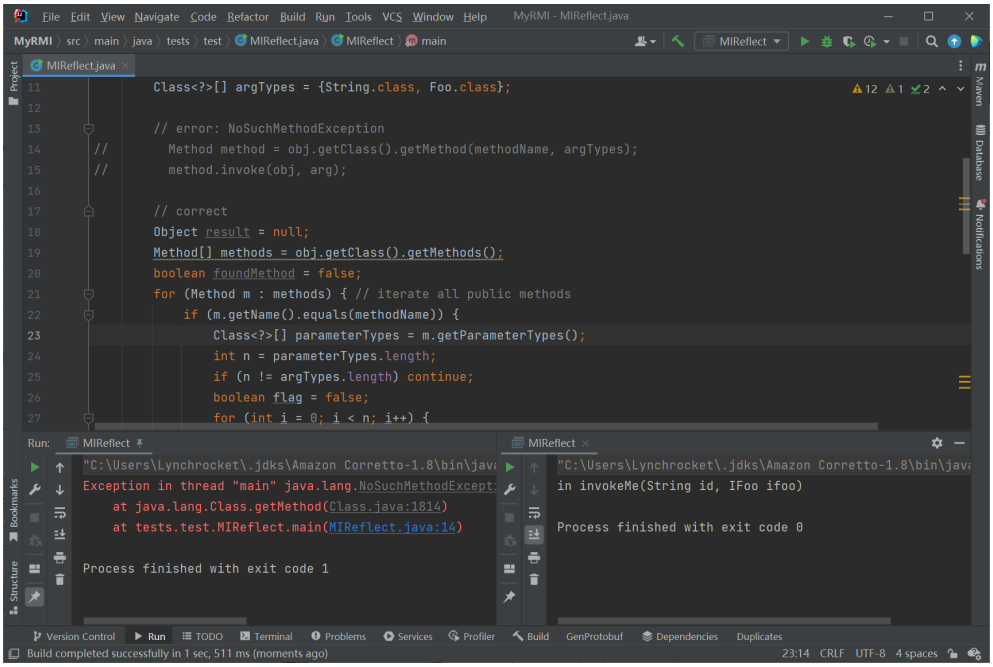

Figure 13: Method invocation using reflection

5.4 Void Parameter and Void Result

Sometimes when we want to invoke the method with no parameter and no return value like

```
public void invokeMe(){/*...*/}
```

we should carefully manage the arguments and result to transmit between local invoker and remote method. More concretely, since the parameter is void, the argument list to transmit may be *null* or a list of 0 length. For unification, I chose to reassign the argument list with an empty list, i.e., a list of 0 length. (the code fragments are from *myrmi/server/RequestProtoHandler.java* and *myrmi/server/StubInvocationProtoHandler.java*)

- Request to send:

```
// arguments are originally as Object, here we need to turn it to com.google.protobuf.ByteString
// be careful with void parameter
List<ByteString> argsList =
        (args == null || args.length == 0) ? new ArrayList<>()
                : Arrays.stream(args)
                .map(x -> ByteString.copyFrom(SerializationUtils.serialize((Serializable) x)))
                .collect(Collectors.toList());
// types of arguments are originally as Class<?>, here we need to turn it to the String name of the Class<?>
List<String> argTypesList =
        (args == null || args.length == 0) ? new ArrayList<>()
                : Arrays.stream(args)
                .map(Object::getClass)
                .map(Class::getName)
                .collect(Collectors.toList());
```

- Receive request:

```
// from com.google.protobuf.ByteString to Object
List<ByteString> argsList = requestInfo.getArgsList();
args = (argsList.size() == 0) ? new Object[0]
        : argsList.stream().map(x -> SerializationUtils.deserialize(x.toByteArray())).toArray();
// find Class<?> from class name using reflection
List<String> argTypesList = requestInfo.getArgTypesList();
argTypes = (argTypesList.size() == 0) ? new Class[0]
        : argTypesList.stream().map(x -> {
    try {
        return Class.forName(x);
    } catch (ClassNotFoundException e) {
        resultInfoBuilder.setException(ByteString.copyFrom(SerializationUtils.serialize(e)));
        resultInfoBuilder.setStatus(0);
        throw new RuntimeException(e);
    }
}).toArray(Class[]::new);
```

However, it may find that I have not do the same process on the *resultInfo*. It is because that the fields of *resultInfo* are neither list nor array (though deprecated, the fields are the same with the final version). So it dose not matter.
From the test (in *tests/test/MiscTest.java*) we can see that the *null* can be normally serialized and then deserialized.

Figure 14: ResultInfo and RequestInfo

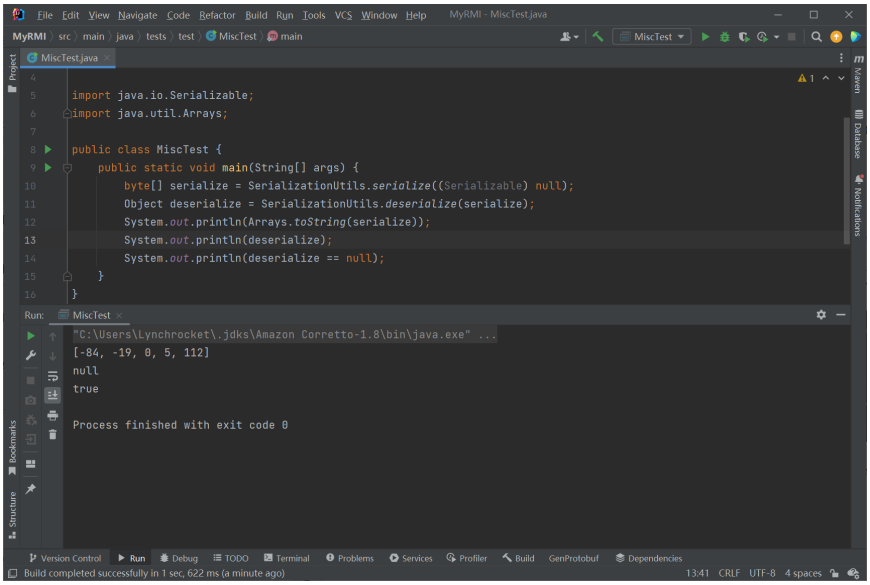

Figure 15: Test of Serialization on null

5.5 Service Monitor

You may notice that in package *RMIZK*, there is a deprecated *class*: *MonitorService*. Recall that in least connection policy, client must invoke *consumer.close()*. This design is not a good option for a language with garbage collection. So the deprecated *MonitorService* actually utilized garbage collection.
You can see in the method *ServiceConsumer::lookup()*, I created a monitor (a weak reference) on the service, and continuing monitoring it in a new thread to see if the service is going to be *gc*. As soon as the service is *gc*, then the *busyNum* of server, which provided the service, is decreased by 1.

```
// MonitorService::call()
@Override
public String call() throws Exception {
    while (!isMonitoredServiceGarbageCollected());
    return this.serverHost;
}
```

```
// ServiceConsumer::lookup()
// monitor the service
MonitorService<Remote> monitor = new MonitorService<>(service, serverHost);
Future<String> futureResult = executorService.submit(monitor);
if (futureResult.isDone()) {
    serverHost = futureResult.get();
    serverMap.get(serverHost).busyNum--;
}
```

However, it did not work as I expected. It is because that the garbage collection does not happen as soon as the object's life time is over, and it is hard to handle the life time of object in Java. Therefore, this plan was deprecated finally.

6 Future Work

Being inspired by the project of reference 8, I expected to implement a simple Google file system using Java RMI at the beginning. It is a pity that due to the limit of time and my knowledge, the expectation was just a blue print. In the future, this work might come true.

7 Reference

[1] Protocol Buffers Documentation
[2] Protobuf in Java: basics, just one!
[3] Protobuf in Java: go deeper, just one!
[4] Basic knowledge of Java IO
[5] Module java.rmi
[6] [leak] - the way to attack Java RMI
[7] The function and usage scenario of WeakReference
[8] Using Java RMI to implement a simple GFS (Google file system)