DISTRIBUTED SYSTEMS

ASSIGNMENT REPORT

ASSIGNMENT ID: No.2 RMI

Student Name: 李田

Student ID: 12112628

DESIGN

The RMI interface of this assignment is implemented by Kotlin. While taking advantage of its language flexibility and convenience, it also provides full capabilities with Java language.

The Java structure provided is not used. Instead, I studied the JRMI structure under Java 17 and build my RMI based on its structure heavily. So do not be surprised at how similar my structure is to JRMI! However, it turns out that the structure is quite different from the provided one. The underlying technics are identical, though.

Below shows the class Diagram of my implementation. (The diagram is also provided as a single file in the submission directory since it is too large to be put inside a report file)

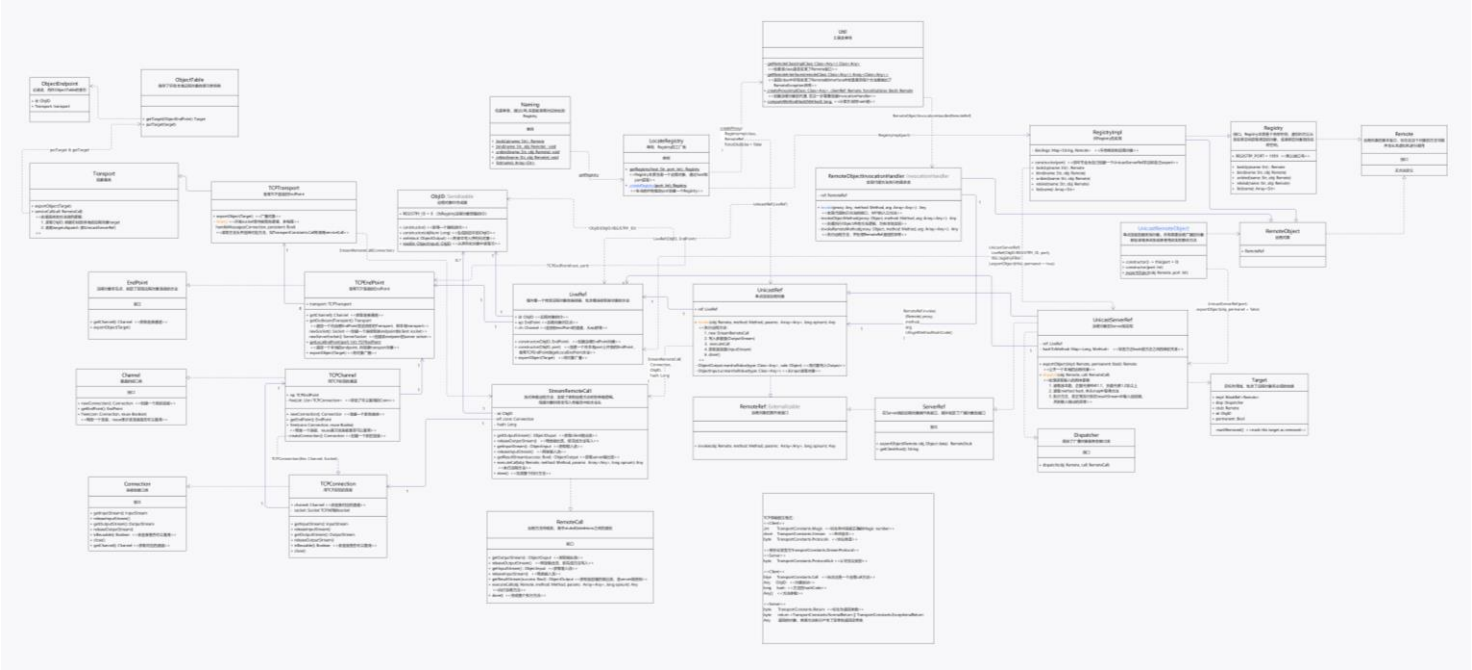


Figure 1. Class Diagram of my RMI

Notably, the ***Skeleton*** and ***RemoteStub*** classes have been marked deprecated in Java 17. Instead, the class ***UnicastServerRef*** and ***UnicastRef*** play similar roles as skeleton and stub, while the transport details are put inside the *transport* package. Additionally, JRMI uses static stubs and skeletons for ***RegistryImpl*** (i.e., using real

classes ***RegistryImpl_Stub*** and ***Registry_Skel***), but in my observation, it is completely fine to use a dynamic implementation. Hence all skeletons and stubs in my implementation are dynamic.

The *transport* layer of JRMI is very well designed, which I borrow a lot in my design. There are four major components in this part: Endpoint, Transport, Channel, and Connection. ***Endpoint*** is barely a class containing a host and a port, it can either represent local or remote endpoints, and new sockets are derived from this class. ***Channel*** is derived from a corresponding Endpoint, and it is responsible for managing the connections connected to the endpoint. ***Connection*** servers as an abstract level of communication, providing input stream and output stream to the endpoint. Finally, the class ***Transport***, only binding to a corresponding local endpoint, manages the server socket and incoming remote invocations. Both server sockets and their accepted sockets will be run on individual threads, managed by the Transport class. The singleton class ***ObjTable*** is used in the process to trace all the objects that have been exported.

On the top of the transport layer, we have the class ***LiveRef***, which acts as a bridge between the transport layer and our skeleton and stub classes: UnicastServerRef and UnicastRef. ***UnicastRef*** can also be referred to as a remote reference, meaning that the client can use it just like a local class reference to invoke methods. This is accomplished by its ***LiveRef*** field, which consists of ***ObjID***, the unique id for identifying the unique object, and the Endpoint of the remote host. ***UnicastServerRef*** inherits from UnicastRef, and it also implements the ***Dispatcher*** interface, which is invoked by Transport on receiving remote method calls, allowing the skeleton to deal with the actual invocation data after the decapsulation from the transport layer.

The following two figures represent the invocation processes:

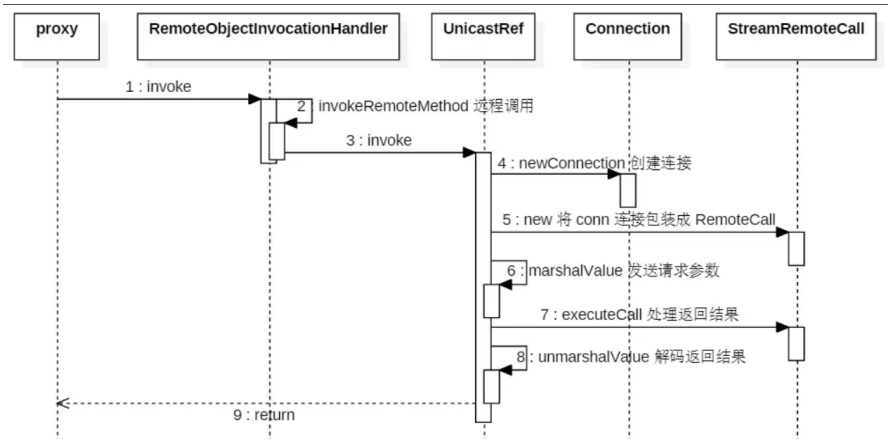


Figure 2. JRMI Invoke process of client
[Source: https://www.jianshu.com/p/7ef5ffa38dfa]

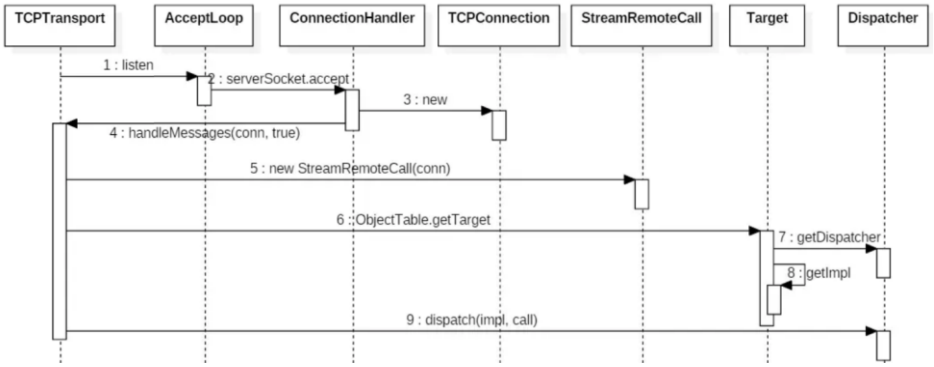


Figure 3. JRMI Invoke process of server
[Source: https://www.jianshu.com/p/7ef5ffa38dfa]

The ***Registry*** class is essentially a remote class that will be automatically exported. Remote hosts can invoke methods on the local registry, and remote references will be used in transport for locally exported objects. This means a local exported object can be bound to a remote registry.

The usage of my RMI is identical to JRMI. Basically, the server first uses ***LocateRegistry.createRegistry*** method to create a registry locally, then implements a remote interface, either letting it inherit ***UnicastRemoteObject*** or using the static method ***UnicastRemoteObject.exportObject*** to export it, then bind it to the registry using ***Naming.bind***. Afterward, the client can get the remote object reference by invoking ***Namine.lookup***. Finally, the server can close the export of objects by invoking ***UnicastRemoteObject.closeExport***.

In summary, what has been implemented in my RMI:

- Invocation of remote methods based on remote references and proxies.
- Locally resolve invocation of object methods (***equals***, ***hashcode,*** and ***toString***) for remote references.
- Serialization of an exported local object as a remote reference, instead of the object itself.
- Client will receive exceptions that happened in the server during invocation.
- Export local objects using ***UnicastServerRef*** (Skeleton), which will listen on an individual thread and manage a thread pool for incoming method invocations.
- ***Registry*** for binding remote objects and looking for remote objects.
- ***Naming*** singleton for accessing remote objects via URL.

What is **not implemented** in my RMI:

- Distributed Garbage Collection (DGC)
- Security check (e.g. *RegistyIml.checkAccess*)
- Other JRMI functions

RUNNING RESULT

The basic functionality of my RMI is tested for both Java and Kotlin usage. Test scripts can be found under the *src/test* folder.

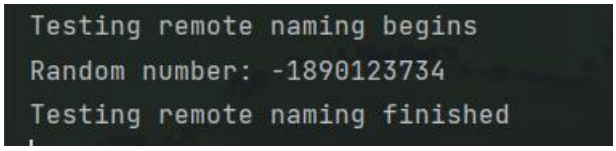
```
public class NamingServerTest {
    little_etx
    public static void main(String[] args) {
        LocateRegistry.createRegistry( port: 8080);
        TestRemoteInterface impl = new TestRemoteImpl();
        UnicastRemoteObject.exportObject(impl);
        Naming.rebind( url: "rmi://localhost:8080/test", impl);
    }
}
```

Figure 4. RMI test script (Server)

```
public class NamingClientTest {
    little_etx
    public static void main(String[] args) {
        TestRemoteInterface test = (TestRemoteInterface) Naming.lookup( url: "rmi://localhost:8080/test");
        test.print("Testing remote naming begins");
        println("increase server field to " + test.increase());
        println("square of 5 is " + test.square( x: 5));
        println("add 5 to 6 is " + test.add( x: 5,  y: 6));
        println("value from server is " + test.getValue());
        try {
            test.testExp();
        } catch (Exception e) {
            println("Exception caught from server: " + e.getMessage());
        }
        println("test remote toString: " + test);
        TestRemoteInterface test2 = (TestRemoteInterface) Naming.lookup( url: "rmi://localhost:8080/test");
        println("test remote equals: " + test.equals(test2));
        test.print("Testing remote naming finished");
    }

    7 usages  little_etx
    private static void println(String string) { System.out.println(string); }
}
```

Figure 5. RMI test script (Client)



```
Testing remote naming begins
Random number: -1890123734
Testing remote naming finished
```

Figure 6. RMI test result (Server)

```
increase server field to 1
square of 5 is 25
add 5 to 6 is 11
value from server is -1890123734
Exception caught from server: Test exception thrown from the serve
test remote toString: Proxy[TestRemoteInterface, RemoteObjectInvocationHandler[UnicastRef[endpoint:[localhost:62240](local), objID:[193237794389472386]]]]
test remote equals: true
```

Figure 7. RMI test result (Client)

PROBLEMS

The first problem I encountered, is the difficulty to understand the source codes of JRMI. Since I have made up my mind to build a structure based on JRMI on my own at the very beginning, I must go through almost all JRMI source codes and find their inner relationship. It is fair to say drawing the **class diagram** (see Figure. 1) really saves me, by which I can clearly see how the classes work together and neglect those classes not involved in the core functionality. In my final implementation, there

are 30 non-exception classes. Without the class diagram, it is impossible to handle their relationships.

The second problem happens during the debugging process. Since I use thread pools to handle remote invocation on the server side, it is frustrating to track system states using IDEA debug tool by choosing the focus thread each time. It will be easier to simply print out each step in the console, but it will break the usability of my RMI and it is not a good practice in software engineering. To solve this problem, I utilize Java Logger to log and trace processes. It is my first time to use Java Logger and it takes me a while to understand its inheritance mechanism. Whatever, the result is pretty good, as can be seen in the test scripts. The increment process model I applied in my development process, that is, writing a part and then testing a part, also helps to reduce possible bugs.

Another huge issue that happens during development is Serialization. It is easy to tell Java VM to serialize objects by adding ***@Serilizable*** tags. However, some fields in the class, such as loggers, should not be serialized. So, we can add ***@Transient*** in Kotlin to tell Java VM not to serialize that field, which will become null after deserialization. However, ***NullPointerException*** will be thrown when again trying to use the field. It is possible to add a custom ***readObject*** method to reassign the field during deserialization, but it cannot be done on *val* fields (similar to *final* fields in Java, whose values cannot be reassigned). I finally solve the problem by a val getter (which will not be considered as a field) accomplished with a transient var field (see *TCPEndpoint.kt, line 26-33*). When the getter found the var field is null, it will reassign a value to that field.

Serialization of remote objects is also an issue. When invoking ***Registry.lookup*** on a remote registry, the serialization of the return object should not be the object itself, but a remote reference that points to the object (otherwise a copy of that object will be generated). At first, I didn't realize the problem and simply serialize the

object, and surprised to find that although the client can correctly receive the object, all its invocations are taken locally and do nothing with the remote object that should be invoked. Then I trace back JRMI source codes and found that JRMI uses a custom output writer, ***MarshalOutputStream***, rather than ***ObjectOutputStream*** to write objects. When writing ***Remote*** obejcts, *MarshalOutputStream* will check whether the object is exported (i.e., whether be put in the ***ObjTable***). If so, it will instead write its stub, i.e., the proxy object with our custom ***RemoteObjectInvocationHandler*** with the ***UnicastRef*** pointing to the object. When the client receives the object, it can invoke it as if invoking a remote object.