15640 Distributed System

Project 2 Report

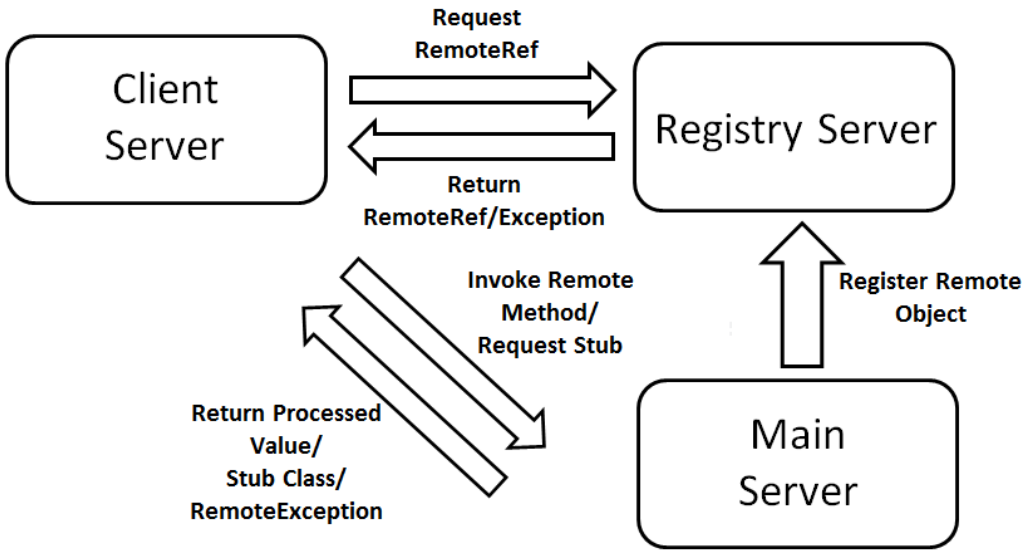
Menglong He (menglonh), Sidi Lin (sidil)

6/23/2014

Table of Contents

System Overview

Overview



Our solution consists of three modules: registry server, main server and client server.

Registry server

Registry server is part of the RMI framework that we designed. On real world deployment it should be included in the jar that is to setup RMI service. Registry server module is used to give client remote object reference that was registered by main server module. RemoteRef is identified by object id and service name. Whenever client send a request for remote object reference, Register server will look up the service name from register list and give back a RemoteRef which consist of server IP, port, object id and interface implementation class name. If no matching service is found, registry server will return a remote exception that includes the specific cause of exception.

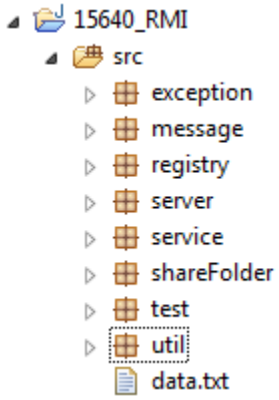
Main server

In our design, main server is setup by those who owns the service that will be providing to others for remote invocation. Main server is responsible for creating and registering services and it should own all the implementation classes of services. It is capable to generate stub classes using interface implementation class as input. Also, it handles all the method invocations by generalizing and stored them. Whenever there is an invocation that requires stub/interfaces as return value, which means, the method invocation is going to change the state of specific instance, we will create another instance based on the previous one (the InstanceID will automatically plus one). If nothing will be changed in that invocation, it will return ReturnMessage object that includes the result of computation.

Client server

Client server invocates the method that the RMI service provided. They can use the RegistryClient in framework to connect to registry server and fetch coordinated RemoteRef object. After that it can establish connection to main server and invoke remote method on server. If stub class does not exist on client side, this framework will download it automatically.

Package architecture



Framework packages

Framework packages include exception, message, registry, server and util. They will be packed into a jar for server or client side for import and setup purpose.

Client packages

Client side should possess the following packages to make RMI calls: framework jar, test class which uses RMI, shareFolder for interfaces, an empty service package for storing stub class.

Server packages

Server side should possess framework jar, shareFolder for interfaces and service stubs and implementation of interfaces.

Key features

Auto stub download

In our framework it can automatically download the stub needed for the remote method invocation. Whenever there is a ClassNotFoundException when localizing the stub, the client server will send a GetStubMessage to server and server will send back the corresponding stub class that is compiled before started. Then the client server will use java classLoader to load that class at runtime and instantiate it.

RMIC

In our framework it is capable to generate stub using the following command:

```
java server.RMIC service.Hello_Interface_Impl
```

It takes implementation class as input and generates the stub used by client. We did this by dynamically generalizing stub source code and compile it into the .class file. We can use the StringBuffer to dynamically append the contents. The total contents include the imported packages, the initialization, communicating between server and client (sending and receiving messages). We use the java reflection to get the methods dynamically and generate the contents.

Hashcode

Hashcode function is used to create identification for different methods. We cannot identify overload methods by number of arguments or types of arguments because they will be unbox as objects. Also, the built-in hashcode function is not stable for objects (wrapper) class like Integer or Double. However it is stable on String object, which we suspected it is because of the string pool in JVM. So on server side we rewrote a hashcode function that generates unique method identification using the following method: We put return type, method name and parameters in to a string and invoke hashCode(), which works pretty well in our solution.

Keep the User's state

To identify which method is being invocated, we created a table to store the hashcode and method object. For those methods that will change the state of object registered, server will create new instance after method invocation and create, return a stub that contains new RemoteRef in it. This helps us to manage different client's instances.

Socket Cache

We abstracted the common functions of communications between the different servers and built a utility class. This class is put in the util package and named CommunicationUtil. This class maintains a HashMap. The key is "ip+port" and the value is "socket" object. It caches the socket connections and save lot of system resources.

Pre-requisition & limitation

There are some pre-requisition and limitation on this framework:

1. Ant (we use the 1.9.4 version) will be used to compile class files.
2. The registry server must run before than the main server running.
3. Instances should be registered from server to registry server before being invocated.
4. Package name should be included when registering classes to registry server.
5. Due to slash problem (Windows OS, it uses the "\"), this framework should only be running on unix or linux platform.
6. Client side should at least possess test, shareFolder and the framework packages.
7. Client side will need empty "service" package in order to store stub classes download in "bin" folder.
8. Each client will have its own instance after initial invocation of method in registered instance.
9. Framework packages should be included in both server and client side.
10. All classes should be compiled using ant before running this system.
11. Stubs should be generated before the system running.
12. The Registry IP, port and main server's port are needed to be set before compiling.

System design

Registry

Registry client

Registry client is basically implementing the same function as Naming in Java RMI. It is used to build connection to registry server by client side. It provides lookup method that returns RemoteRef object that gives client connection information about server and remote objects' ID. The rebind method is used to bind the service name and the specified Remote Object Reference. It also provides list method that is not used in this test project.

Registry server

It basically setup a socket server opens to outer connection requests. It also maintains a HashMap that was used to store the mapping between the service name and Remote Object Reference. It was used to handle the rebind operation and providing the lookup function.

Registry service

Registry service is the main service thread of registry server. It plays as a listener. When the listener received the messages from the others server, it will parse the message and send the message to the Registry Server to get the return value.

Server

Main server

Main server is the main service class of server. It registers object onto registry server and handle messages sent from client side such as GetStubMessage or MethodInvocationMessage. It also provides a command line for users to register service or check status of services. ListenerService class handles all socket requests.

RemoteRef

RemoteRef provide reference to main server and remote object. Information it contains includes: main server IP, port, instance id, and interface implementation class name for stub reference.

RemoteStub

RemoteStub is an interface that designed for those methods that needs to change the state of registered instances and with interface return type.

RMIC

RMIC is used to generate stub class in the server side with implementation class as input. It will generate class file with "[class_name]_Stub.class".

Test

This package consists of four test client classes: Hello_Client, NameServerClient, ZipCodeClient, ZipCodeRListClient. The first test case is used for RMI basic functionality test and prints out a string.

Message

There are a couple of message type will be used in this system: ExceptionMessage, GetStubMessage, MethodInvocationMessage, RegistryMessage and ReturnMessage. They are all implemented RMIMessage and Remote interface.

Deployment

Deploying server

The deployment of server side of this system will include following 4 steps:

1) Setup the Ant Environment:
   For Windows:
   Assume Ant is installed in c:\ant\. The following sets up the environment:
   set ANT_HOME=C:\apache-ant-1.9.4
   set JAVA_HOME=C:\Program Files\Java\jdk1.7.0_60
   set PATH=%PATH%;%ANT_HOME%\bin
   For Mac:
   export ANT_HOME=/Users/menglonghe/apache-ant-1.9.4
   export PATH=${PATH}:${ANT_HOME}/bin

   Reference: http://ant.apache.org/manual/install.html#optionalTasks

2) Set the Constants in src/util/Constants
   The default value is localhost:1099 and the MainServer's port is 15640

3) Build the project:
   Enter into the project dir and type the command
   ant -file build.xml

4) Build stub using RMIC
   The example commands: (before doing this, you must enter into the project/bin dir. The following steps must do like this!)
   java server.RMIC service.Hello_Interface_Impl
   java server.RMIC service.ZipCodeRListImpl

Deploying client

Deploying client side of this system includes x steps:

1) Setup the Ant Environment:
   For Windows:
   Assume Ant is installed in c:\ant\. The following sets up the environment:
   set ANT_HOME=C:\ant
   set JAVA_HOME=C:\Program Files\Java\jdk1.7.0_60
   set PATH=%PATH%;%ANT_HOME%\bin
   For Mac:

export ANT_HOME=/Users/menglonghe/apache-ant-1.9.4
export PATH=${PATH}:${ANT_HOME}/bin

Reference: http://ant.apache.org/manual/install.html#optionalTasks

2) Set the Constants in src/util/Constants
   The default value is localhost:1099 and the MainServer's port is 15640

3) Build the project:
   Enter into the project dir and type the command
   ant -file build.xml

4) Clean up "service" folder **(DO NOT DELETE THE FOLDER ITSELF)** in \bin to test the stub downloading function.

Testing

To start the testing, we need to run all the servers in this project:

1. **Start registry server.**
2. **Start main server.**
3. **Registry service through main server's command line.**
4. **Run the test client class.**

Detailed commands

1) Start the RegistryServer

java registry.RegistryServer

2) Start the MainServer

java server.MainServer

3) Register the service to the Registry Server using command line provided:

For HelloClient:

register service.Hello_Interface_Impl sayHello

For ZipCodeClient:

register service. ZipCodeServerImpl zipCode1

For ZipCodeRListClient:

register service.ZipCodeRListImpl zipCode2

For NameServiceClient:

register service.NameServerImpl nameServer

4) **Using the client to get the result**

For HelloClient:

java test.Hello_Client

For ZipCodeClient:

java test.ZipCodeClient localhost 1099 zipCode1 data.txt

For ZipCodeRListClient:

java test.ZipCodeRListClient localhost 1099 zipCode2 data.txt

For NameServiceClient:

java test.NameServiceClient localhost 1099 nameServer

Future works

Future works could be done to make this project more easy to use.

Garbage collection

For those methods that need to change the inner state of a registered remote object reference, we created a new instance after method is invocated. This may cause out of memory problem since those instances that is out of date are not collected and also, reclaiming the available number of object ids will eliminate the risk of integer overflow. Also, for those objects that will not be used remotely, they should be collected too.

Unify message format

We used a couple different message classes in this solution, which we thought could be integrated in future version.

Reference to:

1) Java source code;
2) https://github.com/richardzhangrui/RMI