Report of RMI Facility

Project 1 of 15-640

Kailiang Chen(kailianc) & Yang Pan(yangpan)

# Design

In this project, our goal is to design and implement a **RMI(Remote Method Invocation)** facility for Java, which provide a mechanism by which objects within one Java Virtual Machine (JVM) can invoke methods on objects within another JVM, even if the target object resides within a JVM hosted by a different, but network accessible, machine.

As we know, **remote procedure call (RPC)** approach extends the common programming abstraction of the procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it is local. **Remote method invocation (RMI)** is similar to RPC but for distributed objects, with added benefits in terms of using object-oriented programming concepts in distributed systems and also extending the concept of an object reference to the global distributed environments, and allowing the use of object references as parameters in remote invocations. As in Figure 1 and Figure 2, RMI is a middleware layer framework, which is based on Request-Reply protocol.

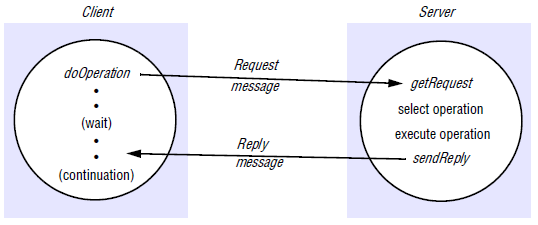
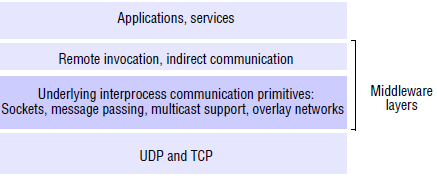


Figure 1 Middleware Layer Figure 2 Request-Reply Protocol

Standard Java RMI framework, which realizes a **Distributed Object Model** using Java Socket, Reflection and Serialization mechanisms. It is made up of a communication module, a remote reference module and some other parts like proxy, dispatcher and skeleton, as Figure 3 and 4 shown.

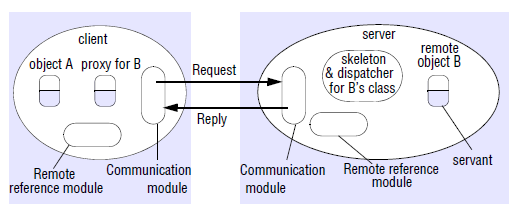
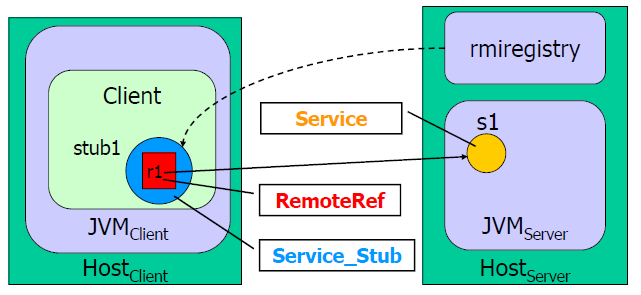
 

Figure 3 Role of proxy and skeleton Figure 4 Stub and Remote Reference

The key to **RMI** is **proxy** pattern, which is based on an idea that interface defines behavior and class defines implementations (Figure 5). It is built from three abstraction layers, Stub and Skeleton layer, Remote Reference layer and Transport layer (Figure 6).

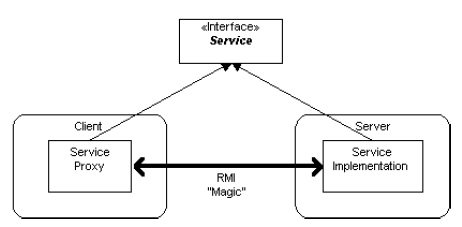
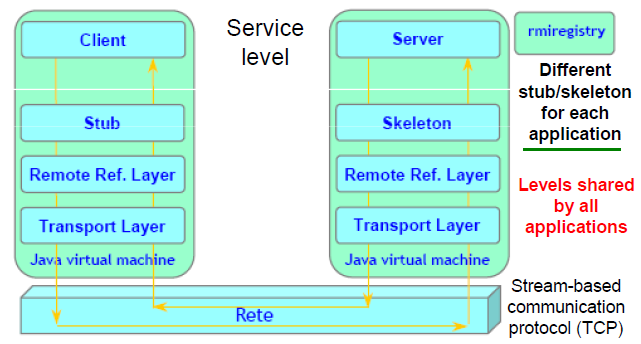
 

Figure 5 Proxy Pattern Figure 6 RMI Architecture

The whole RMI work flow is:

(1) A server object is registered with the RMI registry; it runs on server that hosts remote service and accepts queries for services, by default on port 1099.

(2) A client looks through the RMI registry for the remote object by lookup();

(3) Once the remote object is located, its stub is returned in the client;

(4) The remote object can be used in the same way as a local object. The communication between the client and the server is handled through the stub and skeleton.

Our RMI facility mimics the fundamental functions of standard Java RMI. It has a communication module, a remote reference module, a stub interface for different kinds of service to implement and a dispatcher. More specifically, our RMI has Remote Registry, Remote Object Reference, Remote Interface, Remote Exception and Actions in distributed system environment. However, we have no consideration on distributed garbage collection, the .class files download system and rmic compiler system.

Moreover, the basic design decision might be on the Security and Scalability of RMI system, which means Server has right to authorize Client to invoke some actions and multiple Clients can manipulate the same remote object at the same time. In standard java, we can find a SecurityManager class to provide security protection mechanism. Here, we have not implemented it as we figure the purpose of this project is to mimic basic remote invocation function in RMI. And if we want to consider Scalability problem, synchronization control might be needed.

# UML

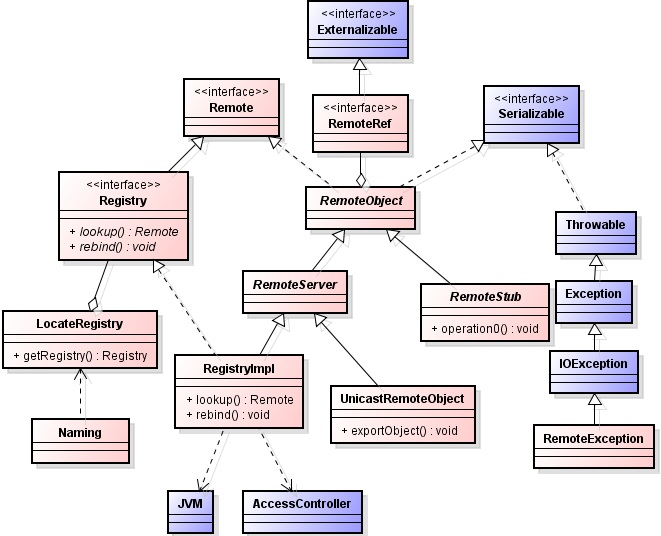


Figure 7 Java RMI Diagram

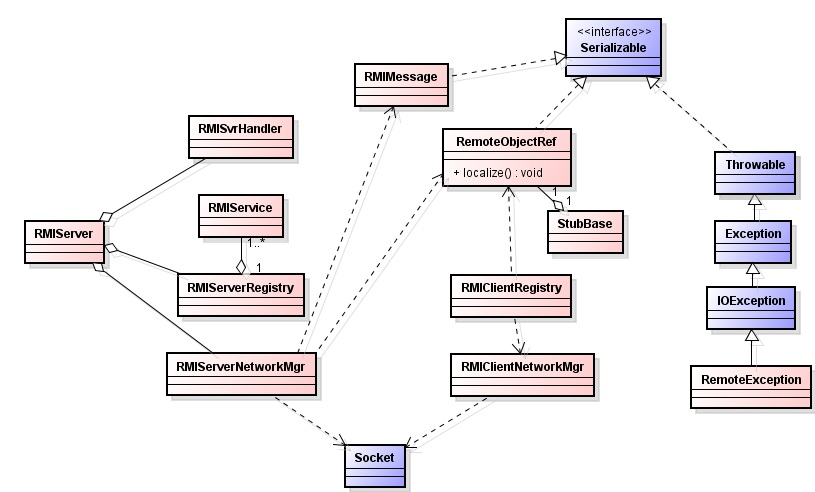


Figure 8 Our RMI Class Diagram

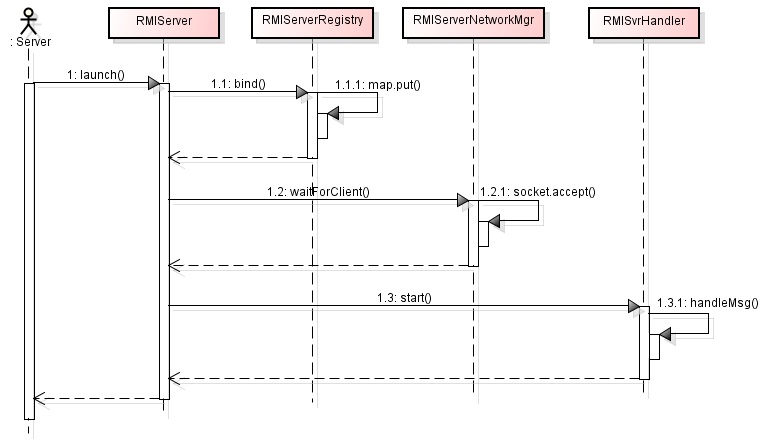


Figure 10 Server Sequence Diagram

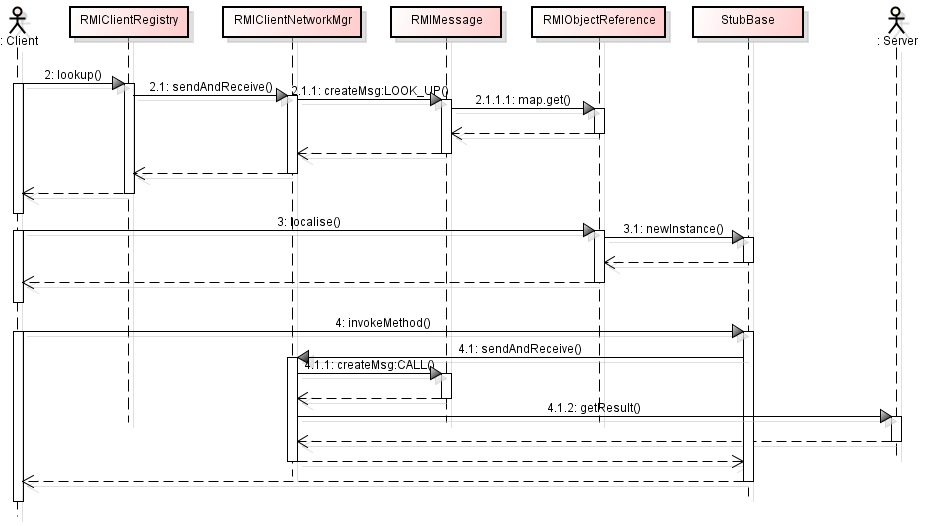


Figure 11 Server Sequence Diagram

## Special features

1. Support RMI(Remote Method Invocation) between Server-Multiple Clients
2. Support object, primitive type as parameter and return value
3. Support easy-to-use facility interface as simple as standard java RMI
4. Support easy-to-extend and loosely coupled communication module and remote reference module for Server-Client

# Implementation

## Class Implementation

According to the UML class diagram, our system consists of:

1. RMIMessage: a concrete class defines the message transmitted between Server and Client, which is used during service lookup, method invocation and result return. Additional message type and content can be added if necessary.
2. RemoteObjectReference: a concrete class defines the service instance reference Server provides for Client, its information is managed by Server.
3. RMIService: an abstract class defines interface for services and serves as an element of Server’s information table. It supports pass by reference and pass by value depending on whether Client service extends it or not.
4. StubBase: an abstract class defines interface for services and wraps RemoteObjectReference for Server/Client communication.
5. RemoteException: a common exception class defines the specific exception for RMI facility, it can be extended by adding some message display or handling method in the future.
6. RMIServerRegistry/RMIClientRegistry: registry classes define the basic service lookup and bind functions for Server and Client. A common base class or interface can be added to improve the extensibility, however, we remain it unimplemented as it is not necessary in our project.
7. RMIServerNetworkMgr/ RMIClientNetworkMgr: communication classes define the basic message transmission functions between Server and Client. Similarly, we divide it into Server/Client module as we do for registry classes for readability.
8. RMISvrHandler: Server monitor thread for accepting messages from a client. It accepts message and passes it to RMIServerNetworkMgr to process.
9. RMIServer: Server main thread to bind a new service and launch monitor thread.
10. RMIClient: Client main thread to lookup, localize and invoke a new service.

## Development environment

This project is developed with Eclipse IDE for Java Developers, Luna Release (4.4.0), JDK 8u20. If you want to write a test class inheriting MigratableProcess, you should work in the same environment.

# Test with our examples

We provide two service classes to test the TransactionIO classes and other migration features. They are NonIOProcess and IOProcess.

NonIOProcess is a simple class with only a counter tick-tocking every 0.1s. After migration, the counter starts from the stage right before migration.

IOProcess mainly test the TransactionIO classes, TransactionalFileInputStream and TransactionalFileOutputStream. This class keeps reading in a series of shuffled characters from a file, sorting them and writing them out to a file.

## Test environment

A server at unix.andrew.cmu.edu, unix machines connected to AFS

## Deploy and run

1. Open three Terminals and connect them to a server connected to AFS using ssh. Make sure they have the same login user (saving the troubles brought by file permission).
2. Change directory to ./src/. (for convenience, we denote ProcessMigration/ as ./) in three Terminals.
3. Type “make” and then “make run” in the three Terminals.
4. In one Terminal, type “server” to become a server. And you will get an echo showing the IP and port (say, 192.168.0.1:6777). In the other two, type “client 192.168.0.1 6777) and they will connect to that server.
5. In Client A, type “create NonIOProcess” to create a migratable process without IO operations.
6. In Client B, type “create IOProcess input.txt output.txt” to create a migratable process with IO operations. Note: input.txt and output.txt should have absolute path.
7. In server, type “ps” to show all the running processes on all clients. In the two clients, type “ps” to show all the local running processes.
8. In server, type “migrate 1 0 0” to migrate the IO process on Client B (cid 1) to Client A (cid 0).
9. In server, type “migrate 0 0 1” to migrate the non-IO process on Client A to Client B.
10. In server, type “ps” to show all the running process on all clients after migration. In the two clients, type “ps” to show all the local running processes.
11. Wait for IOProcess to finish on Client A. When it shows “JOB COMPLETED”, the IOProcess finishes all the IO operations.
12. In server, type “exit” to exit server. All other clients connected to this server will exit automatically.

## Understand the test result

The result consist of three parts:

1. Migration status: In step 7, you can see the NonIOProcess running on Client A and IOProcess on Client B. In step 10, you can see NonIOProcess running on Client B and IOProcess on Client A. This means that the two processes have been migrated to each other client.
2. Process status: In step 9, we can see an echo from NonIOProcess saying its counter, like “NonIOProcess : suspend(), cnt = xxx” on Client A. After migration, we can see an echo like “NonIOProcess : run() begin, cnt = yyy”, where yyy = xxx + 1. This means the suspended work before migration is resumed after migration.
3. IO status: In IOProcess, we read a shuffled alphabet one character by one character from input.txt, sort them and write them to output.txt, and repeat this procedure 8 times. So, open output.txt, and you can see 8 set of ordered alphabets in it. This means the IO operations works fine during migration.

# Writing customized test class

By implementing the MigratableProcess class, you can write your own process class to test TransactionInputFileStream, TransactionOutputFileStream and other migration features.

## Developer environment

You can write the class anywhere you want, as long as you inherit from MigratableProcess and implement the three abstract methods. However, it’s strongly recommended to develop in the same environment as ours.

## Understand our interfaces and framework

To run your test class under our framework, you must inherit your test class from MigratableProcess. In MigratableProcess, there are three abstract classes that you have to override in your own class:

1. suspend(): This method will be called before the object is serialized. It affords an opportunity for the process to enter a known safe state.
2. resume():This method will be called after migration. Resume all the work that was suspended.
3. toString():This method is used for debugging. It can print the class name of the process as well as the original set of arguments with which it was called.

Besides, here is some other things should be noted: (assume your class file is GregProcess.java)

1. The constructor of your test class should have exactly one argument with type String[], no matter you need it or not.
2. You can design some test methods to be called after GregProcess has been instantiated. This method should have exactly one argument with type String[].
3. At the beginning of GregProcess.java, you should declare the package by typing “package edu.cmu.andrew.ds.ps;”.
4. In the suspend() method

## Deploy and run

Once you finish your work, copy your test class file (GregProcess.java) to the directory “./src/edu/cmu/andrew/ds/ps/”. And then, edit ./src/Makefile by adding “./edu/cmu/andrew/ds/ps/GregProcess.java” to “CLASSES”.

Now, change directory to ./src/ and type the following commands to run the whole program:

> make

> make run

In the client, after you connect to a server, you can type the following command to create a new instance of your process:

create GregProcess [ANY OPTIONAL ARGUMENT TO CONSTRUCTOR]

You can also call any method in GregProcess to test by typing:

call PID METHOD\_NAME [ANY OPTIONAL ARGUMENT]

where PID is the pid of that instance.

Use all these interfaces to test our framework.