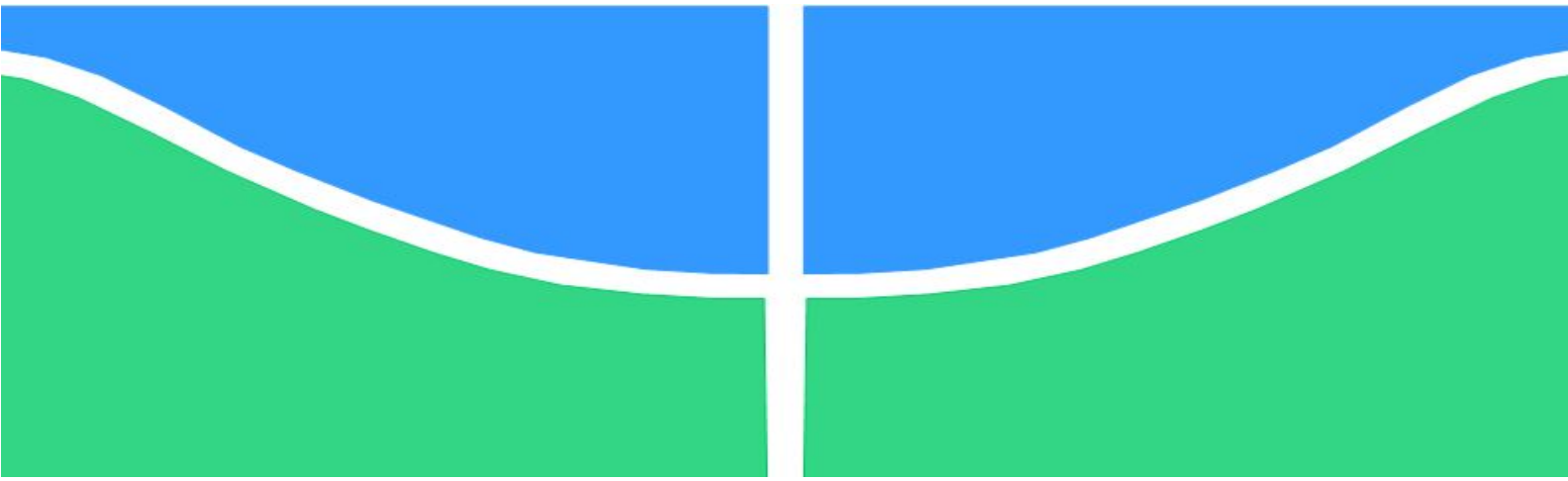

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Fundamentos de Sistemas Distribuídos

Relatório de Experimento com RMI

Phelipe Wener (12/0132893)

Professor:
Fernando W. Cruz

Brasília, DF
19 de Abril de 2017

1. Introdução

Dentro de comunicação interprocesso, tópico essencial para sistemas distribuídos, existem diversas formas de se solucionar os problemas inerentes a tal configuração. O RPC (Remote Procedure Call) é uma maneira de se implementar algumas soluções.

Analogamente o RMI(Remote Method Invocation), surge de uma demanda dentro de implementações java, segundo a própria definição da api: É um mecanismo que habilita um objeto sobre uma maquina virtual Java para invocar métodos em um outro objeto em uma outra maquina virtual [1]. Outras linguagens com paradigma de orientação a objetos usam tecnologias próprias para tal suporte, por exemplo, o Ruby tem o DRb[2].

2. Objetivo

O objetivo desse experimento é entender a construção de aplicações distribuidas com linguagens orientadas a objetos, nesse caso especificamente Java.

3. Ambiente e configuração

Todos os experimentos abaixo foram executados numa distribuição linux chamada kubuntu, que difere fundamentalmente em termos de interface com o ubuntu. Portanto, deve ser possivel executar os mesmos passos em qualquer versão do Ubuntu 14.04, bem como em um Debian 7.

4. Desenvolvimento

Antes de iniciar a implementação java usando RMI, foi feito um pequeno estudo para entender como a API funciona. Além do material do professor, indexado junto ao arquivo de entrega do experimento, foi utilizado um tutorial do oracle sobre RMI[3].

4.1 Implementando a solução

- No roteiro é requisitado uma aplicação servidor que responde a um cliente que passa determinadas operações matemáticas.
- O primeiro a se fazer é definir a interface que irá herdar `java.rmi.Remote` . Foi criado o arquivo Calculator com as operações de soma, subtração, multiplicação e divisão, segue abaixo a assinatura da classe:

```
public interface Calculator extends Remote {
        /**
 * Realizes sum operation
 *
 * @param firstNumber
 * @param secondNumber
 * @return
 */
Float add(Float firstNumber, Float secondNumber) throws RemoteException;

// As outras assinaturas de método eram parecidas.
// Observação: Todo método que herda Remote tem que tratar uma RemoteException
// Caso contrário uma exceção será lançada ao iniciar o servidor
}
```

- Logo após foi implementado o servidor que realiza tal operação:

```
public class Server implements Calculator {...}
```

- Apenas algumas linhas da implementação dessa classe merecem destaque, primeiramente a classe Calculator, que é utilizada como stub:

```
        Calculator stub = (Calculator) UnicastRemoteObject.exportObject(obj, 0);
```

- Segundo o javadoc de UnicastRemoteObject, essa classe é usada para exportação de objeto remoto com JRMP e obtenção de um stub que comunica ao objeto remoto. Para uso dos stubs é preciso utilizar o aplicativo `rmic` , conforme será explicado adiante.
- Ainda na classe Server:

```
Registry registry = LocateRegistry.getRegistry();
registry.bind("Calculator", stub);
```

- Essa linha ainda é parte do registro de stub, é usada pelo `rmiregistry` que também será explicado adiante, basicamente o bind deve usar o design pattern Reflection para executar os métodos implementados de Calculator, por isso ele requer o nome do Objeto que implementa Remote e a instância de um stub, que em tempo de execução irá chamar o devido método.
- Para mostrar que o servidor está ativo temos ao final do método main a seguinte impressão:

```
System.out.println("Server ready");
```

- Já na classe `Client` , tratamos inicialmente os argumentos na implementação do método main, que lida com os seguintes dados:
  - host (IP)
  - primeiro operando
  - operador
  - segundo operando
- Algumas linhas importantes em `Client` :

```
Registry registry = LocateRegistry.getRegistry(host);
Calculator objectStub = (Calculator) registry.lookup("Calculator");
// ...
System.out.println("Result is " + result);
```

- As primeiras linhas registram um objeto stub que será utilizado para fazer a chamada de método remoto, essa chamada irá invocar o servidor que retornará o result impresso na última linha.

4.2 Passos de execução:

- Na pasta do código é necessário compilar todos arquivos de extensão .java:

```
$ javac *.java
```

- Segundo o comando `man rmic` : rmic - Generates stub, skeleton, and tie classes for remote objects that use the Java Remote Method Protocol (JRMP) or Internet Inter-Orb protocol (IIOP). Portanto é uma ferramenta utilizada para geração dos stubs explicados acima.

```
$ rmic Server
```

- Segundo o comando `man rmiregistry` : rmiregistry - Starts a remote object registry on the specified port on the current host. Portanto foi executado:

```
$ rmiregistry
```

- O comando abaixo executa o servidor. Deve ser executado em outro bash uma vez que o comando acima fica executando em paralelo.

```
$ java Server
```

- Os comandos abaixo são do cliente, também devem ser executados em outro bash, pois teremos 2 processos citados acima ativos.

```
$ java Client 192.168.0.105 2 '*' 2
Writes the operation:
Result is 4.0
$ java Client 192.168.0.105 2 / 2
Writes the operation:
Result is 1.0
$ java Client 192.168.0.105 2 + 2
Writes the operation:
Result is 4.0
$ java Client 192.168.0.105 2 - 2
Writes the operation:
Result is 0
```

5.1 Problemas encontrados

Durante o experimento o método `registry.lookup("Calculator")` estava escrito de forma errada, uma exceção bem explicativa foi lançada e foi possível corrigir o problema.

5.2 Limitações de código

A operação de multiplicação deve ser passado entre aspas simples, uma vez que o bash reconhece o caractere * como um regex.

6. Referências

[1] https://docs.oracle.com/javase/7/docs/api/java/rmi/package-summary.html

[2] https://ruby-doc.org/stdlib-1.9.3/libdoc/drb/rdoc/DRb.html

[3] https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html