UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

PROGRAMAÇÃO DISTRIBUÍDA COM JAVA RMI

Servidor distribuído de arquivos

Breno Ferreira Leonez Leite

Natal - RN

ABRIL/2022

Sumário

1. Introdução

Java RMI (Remote Method Invocation) é uma interface que executa chamadas remotas estilo RPC e permite a invocação de métodos que estejam localizados em máquinas diferentes. Em resumo, RMI é um mecanismo de chamada de procedimentos remotos orientado a objetos.

Como exemplo de caso de uso da ferramenta, foi implementado um sistema de arquivos e diretórios distribuído, no modelo cliente-servidor. O intuito do projeto é permitir que os clientes conectados possam criar pastas e arquivos, bem como listar e abrir seu conteúdo. Além disso, é possível realizar operações como copiar, apagar e mover um arquivo ou pasta.

A seguir, serão abordados detalhes sobre o desenvolvimento do projeto (arquitetura, implementação e execução) e as percepções e considerações finais sobre ele.

2. Arquitetura, Implementação e Execução

2.1. Arquitetura

O projeto foi desenvolvido no modelo de arquitetura cliente-servidor, no qual o cliente realiza conexão com o servidor e solicita um determinado serviço, enquanto o servidor processa essa requisição através de tarefas. Com relação ao tipo de comunicação, foi escolhido o tipo síncrono, dada a necessidade de obter-se uma resposta em tempo real da operação solicitada pelo cliente.

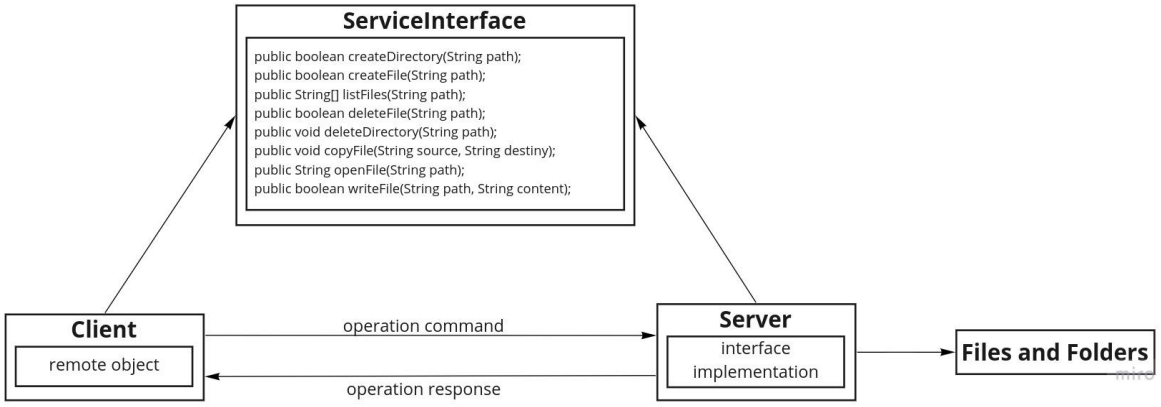

Figura 1 - Diagrama da arquitetura do projeto

O fluxo de funcionamento do sistema e a sua arquitetura podem ser descritos pela Figura 1. Inicialmente, o cliente se conecta ao servidor para ter acesso a um terminal. Através dele, é possível enviar comandos correspondentes a operações com os arquivos e diretórios presentes no servidor, os quais o cliente é responsável por interpretar e fazer as chamadas dos métodos remotos presentes no servidor. Todos os métodos são implementados baseados em uma interface chamada Service. Após realizar as devidas manipulações, o servidor retorna as respostas das operações ao cliente, que por sua vez, imprime uma mensagem no terminal indicando se ela foi bem-sucedida ou não.

2.2. Implementação

Inicialmente, temos a interface remota Service (Figura 2) contendo a declaração dos métodos a serem implementados pelo servidor e invocados pelo cliente. Por padrão, uma interface remota deve estender a interface Remote do pacote java.rmi. Além disso, cada método

declarado deve lançar a exceção RemoteException, que é uma classe de exceções que o RMI lança quando ocorre erro na invocação dos métodos remotos. Na Figura 2, pode-se observar que a exceção IOException também está inclusa, a qual serve para sinalizar que um erro de entrada/saída de algum tipo ocorreu. No total, são 8 métodos declarados no código da interface e suas atribuições, parâmetros e retornos podem ser resumidas da seguinte forma:

```
public interface Service extends Remote {
    public boolean createDirectory(String path) throws RemoteException, IOException;
    public boolean createFile(String path) throws RemoteException, IOException;
    public String[] listFiles(String path) throws RemoteException, IOException;
    public boolean deleteFile(String path) throws RemoteException, IOException;
    public void deleteDirectory(String path) throws RemoteException, IOException;
    public void copyFile(String source, String destiny) throws RemoteException, IOException;
    public String openFile(String path) throws RemoteException, IOException;
    public boolean writeFile(String path, String content) throws RemoteException, IOException;
}
```

Figura 2 - Código da interface Service contendo os métodos do servidor

- createDirectory: Recebe uma string path contendo o caminho de um diretório a ser criado. Retorna verdadeiro caso a operação seja bem-sucedida, falso caso contrário.
- createFile: Recebe uma string path contendo o caminho de um arquivo a ser criado. Retorna verdadeiro em caso de sucesso, falso caso contrário.
- listFiles: Recebe uma string path contendo o caminho de um diretório cujo conteúdo será listado. Retorna um array de strings contendo os nomes dos diretórios e arquivos dentro da pasta passada por parâmetro.
- deleteFile: Recebe uma string path representando o caminho de um arquivo a ser apagado. Retorna verdadeiro caso o arquivo seja removido com sucesso, falso caso contrário.
- deleteDirectory: Recebe uma string path com o caminho de um diretório a ser apagado.
- copyFile: Recebe uma string source com o caminho de origem de um arquivo e uma string destiny contendo um caminho de destino para onde o arquivo deverá ser copiado.
- openFile: Recebe uma string path contendo o caminho de um arquivo e retorna uma string com o conteúdo desse arquivo.
- writeFile: Recebe uma string path com o caminho de um arquivo e uma string content com um conteúdo que deverá ser armazenado nesse arquivo. Retorna verdadeiro caso a escrita ocorra com sucesso, falso caso contrário.

A classe Server é a classe que implementa o servidor do projeto, cujo código pode ser

observado na Figura 3. Ela possui três exceções a serem lançadas na sua cláusula throws: RemoteException, explicada anteriormente, MalformedURLException, que indica que houve a ocorrência de uma URL mal formatada, e NotBoundException, a qual é lançada caso não seja encontrada uma associação a um nome de registro pesquisado. Primeiramente, definimos o hostname do servidor do RMI como 127.0.0.1 (localhost). Em seguida, instanciamos a nossa interface remota através da classe Task, responsável por implementar os métodos declarados em Service, e criamos um registro RMI na porta 1098. Após isso, definimos uma URL para registrar o nosso serviço e imprimimos no terminal que o servidor foi iniciado.

```java
public class Server {
    public static void main(String[] args) throws MalformedURLException, RemoteException, NotBoundException {
        System.setProperty("java.rmi.server.hostname","127.0.0.1");
        Service service = new Task();
        LocateRegistry.createRegistry(1098);
        Naming.rebind("rmi://127.0.0.1:1098/FileService", service);
        System.out.println("Server started.");
    }
}
```

Figura 3 - Código da classe Server, que implementa o servidor do projeto

Já o código da classe Client (Figura 4) pode ser dividido em duas partes. No método main, o cliente pesquisa pelo serviço que foi registrado anteriormente pelo servidor e imprime que a conexão foi realizada. Em seguida, criamos um scanner para receber os comandos digitados pelo usuário no terminal e criamos um loop infinito para capturar esses comandos. Dentro do loop, a função runCommand é chamada, passando o serviço pesquisado e umas lista de strings contendo o que foi digitado pelo usuário.

```java
public class Client {
    public static void main(String[] args) {
        try {
            Service service = (Service) Naming.lookup("rmi://127.0.0.1:1098/FileService");
            System.out.println("Client connected.");
            Scanner sc = new Scanner(System.in);
            while (true) {
                System.out.print("\u001B[32m" + "$ ");
                String line = sc.nextLine();
                System.out.print("\u001B[0m");
                runCommand(service, line.split(" "));
            }
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (NotBoundException e) {
            e.printStackTrace();
        }
    }
```

Figura 4 - Código da classe Cliente, que implementa o cliente do projeto

A função runCommand é responsável por interpretar os comandos digitados pelo usuário e

invocar os métodos remotos do servidor corretamente. Em termos de código, ela é composta por um switch case (Figura 5), que verifica o primeiro índice da lista de strings args passada como argumento e faz a chamada do método correto, além de imprimir no terminal as mensagens necessárias.

```java
private static void runCommand(Service service, String[] args) {
    String command = args[0];

    try {
        switch (command) {
            case "ls":
                String[] fileList = service.listFiles(args[1]);
                System.out.println("List of files and directories:");
                for (int i = 0; i < fileList.length; i++) {
                    System.out.println(fileList[i]);
                }
                break;
            case "mkfile":
                boolean isFileCreated = service.createFile(args[1]);
                if (isFileCreated) {
                    System.out.println("Successful file creation.");
                } else {
                    System.out.println("File creation failed.");
                }
                break;
```

Figura 5 - Trecho de código da função runCommand

Na classe Task, a classe File do pacote java.io foi utilizada em boa parte das implementações para acessar os arquivos e diretório e utilizar alguns métodos úteis ao projeto.

2.3. Execução

Ao executar o cliente, é impressa uma mensagem de conexão e iniciado um terminal com o qual o usuário pode enviar os comandos das operações e por padrão, receber uma mensagem de sucesso ou erro (Figura 6). Os comandos e suas respectivas operações são:

- ls <path>: listar conteúdo de um diretório.
- mkfile <path>: criar arquivo.
- mkdir <path> criar diretório.
- rmfile <path>: remover arquivo.
- rmdir <path>: remover diretório.
- cp <source> <destination>: copiar arquivo.
- open <path>: abrir arquivo.
- write <path>: escrever arquivo.

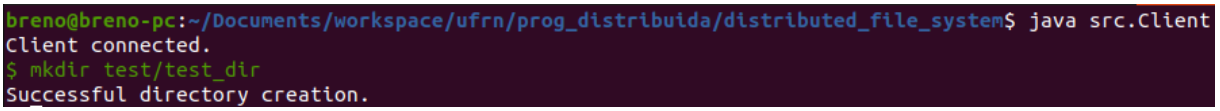

Figura 6 - Execução do cliente e exemplo de comando

3. Conclusão

Como produto final do trabalho, obteve-se um sistema de arquivos distribuído construído com Java RMI, no qual o cliente conectado ao servidor pode criar arquivos e diretórios e gerenciá-los utilizando comandos em um terminal.

Trata-se de uma solução robusta, com arquitetura e implementações simples, que se aproveitam de soluções nativas do Java. Durante o seu desenvolvimento, foram encontradas dificuldades ao implementar um sistema de navegação mais complexo, semelhante ao shell do Linux, por exemplo, de forma que o usuário possa entrar dentro das pastas e digitar comandos mais simples. Um ponto de melhoria seria a criação de logs do servidor, de forma a evidenciar possíveis erros na sua execução.

4. Referências

RMI (Remote Method Invocation), Understanding requirements for the distributed applications,Java RMI Example,RMI Example, - W3cschoool.COM. Disponível em: <https://w3cschoool.com/java-rmi>. Acesso em: 3 maio. 2022.

RMI. Disponível em: <https://pt.wikipedia.org/wiki/RMI>. Acesso em: 3 maio. 2022.

Uma introdução ao RMI em Java. Disponível em: <https://www.devmedia.com.br/uma-introducao-ao-rmi-em-java/28681>. Acesso em: 3 maio. 2022