Relatório do trabalho prático: Invocação de Métodos Remotos (RMI)

Artur Curinga
Victor Agnez

Agosto 2018

1 Introdução

O presente relatório visa descrever e detalhar o trabalho prático desenvolvido para o tema:

Problema 15 - Algoritmos de Ordenação *: Implementar um servidor que realiza a ordenação de um conjunto de valores inteiros e o tempo despendido para a realização dessa operação. O servidor deverá prover a implementação de diferentes algoritmos de ordenação e os disponibilizar aos clientes, tais como os apresentados em Wikipedia . Em sua requisição, o cliente deverá especificar o conjunto de valores a ordenar e o tipo de algoritmo que deverá ser utilizado para realizar a ordenação.*

A atividade é parte da disciplina de DIM0614 - Programação Distribuída ministrada pelo professor Dr. Everton Ranielly de Sousa Cavalcante. O código dessa atividade pode ser encontrado no repositório: `https://github.com/victoragnez/programacao-distribuida/tree/master/projeto1-RMI`

2 Implementação do RMI

Remote Method Invocation (RMI) é uma tecnologia de comunicação entre objetos que permite a um objeto cliente invocar um método de um objeto remoto.

O projeto foi desenvolvido em Java, linguagem que possui módulo RMI no pacote *java.rmi*.

Para separar melhor as classes pertencentes ao servidor e cliente, dividimos o projeto em três pacotes: *server*, para o servidor e suas classes; *client*, para o cliente; e *shared*, para a interface remota, acessível tanto pelo servidor quanto pelo cliente. Existe também o pacote *tests*, destinado criação, execução e verificação de 555 testes para o projeto. A interface remota *shared.Compute* é implementada pelo servidor e é visível ao cliente, de modo que a referência do objeto registrado no servidor pode ser utilizado pelo cliente.

O *client.Client* representa o cliente que realiza requisições ao servidor por meio do método *Naming.lookup* para procurar a referência registrada pelo servidor no RMI Registry no endereço definido, e assim poder invocar o método estabelecido na interface *shared.Compute*.

A classe abstrata *server.Sort* define os métodos que cada classe com um algoritmo de ordenação deve implementar. Foram escolhidos 11 algoritmos, tendo então 11 classes que herdam dessa classe.

O *server.Server* implementa a interface *shared.Compute*. Já a classe *server.Main*, cria uma nova instância dele, registra-a no RMI Registry (módulo de referência remota) e executa o *Naming.rebind* para o endereço especificado. No método *sortArray* da classe *server.Server*, é instanciada uma classe *server.Sort*, de acordo com o tipo de ordenação especificada, e é executado o algoritmo, medindo-se o seu tempo. O retorno desse método é um par com o vetor ordenado e o tempo medido.

2.1 Como executar

Executar primeiro o método *server.Main* para registrar a classe *server.Server* no RMI Registry no endereço previamente estabelecido. Depois executar a classe *client.Client*, que utiliza o método *Naming.lookup* para adquirir a referência registrada pelo servidor. Nela, serão especificados o vetor de inteiros a ser ordenado, algoritmo desejado e o tipo de ordenação (crescente, decrescente).

3 Ordenação

Foram implementados ao todo 11 algoritmos de ordenação: BinaryTree Sort, Bubble Sort, Couting Sort, Heap Sort, Insertion Sort, Merge Sort, Quick

Sort, LSD Radix Sort, Random Sort, Selection Sort e Stooge Sort. A tabela abaixo mostra a complexidade assintótica dos algoritmos de ordenação implementados.

Nome	Melhor caso	Caso Médio	Pior caso
BinaryTree Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Couting Sort	$O(n + r)$	$O(n + r)$	$O(n + r)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
LSD Radix Sort	$O((n + k)\frac{\log n}{\log k})$	$O((n + k)\frac{\log n}{\log k})$	$O((n + k)\frac{\log n}{\log k})$
Random Sort	$O(n)$	$O((n + 1)!)$	Unbounded
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Stooge Sort	$O(n^{\frac{\log 3}{\log 1.5}})$	$O(n^{\frac{\log 3}{\log 1.5}})$	$O(n^{\frac{\log 3}{\log 1.5}})$

4 Testes

Os algoritmos de ordenação foram devidamente testados através de testes unitários com o JUnit 5 utilizando uma classe parametrizada *SortTest.java*. Foram divididos em 5 conjuntos de testes: *smallTests*, *bigElementsTests*, *bigArrayTests*, *bigTests gigantic Tests*. A divisão ocorreu da seguinte forma:

Bloco de teste	Número de testes	Tamanho do vetor	Faixa de valores
smallTest	15	$0 \leq n \leq 6$	$\|v_i\| < 100$
bigElements	9	$1 \leq n \leq 6$	Sem restrição
bigArray	15	$100 \leq n < 300$	$\|v_i\| \leq 2 \cdot n$
bigTest	15	$100 \leq n < 300$	Sem restrição
giantTest	3	$n = 200000$	Sem restrição

Os conjuntos de testes foram direcionados aos algoritmos de acordo com seu desempenho. A tabela a seguir lista essa divisão:

Nome	smallTest	bigElements	bigArray	bigTest	giantTest
BinaryTree Sort	Sim	Sim	Sim	Sim	Sim
Bubble Sort	Sim	Sim	Sim	Sim	Não
Couting Sort	Sim	Não	Sim	Não	Não
Heap Sort	Sim	Sim	Sim	Sim	Sim
Insertion Sort	Sim	Sim	Sim	Sim	Não
Merge Sort	Sim	Sim	Sim	Sim	Sim
Quick Sort	Sim	Sim	Sim	Sim	Sim
LSD Radix Sort	Sim	Sim	Sim	Sim	Sim
Random Sort	Sim	Sim	Não	Não	Não
Selection Sort	Sim	Sim	Sim	Sim	Não
Stooge Sort	Sim	Sim	Sim	Sim	Não

Com isso, executamos os 555 testes primeiramente sem o uso de RMI, no sistema operacional Ubuntu 16.04 LTS. Assim, pudemos verificar a corretude na implementação dos algoritmos escolhidos. Em seguida, executamos os testes fazendo a comunicação com o servidor, via RMI, simulando uma conexão do cliente. Isso nos permitiu verificar o funcionamento da comunicação via invocação de métodos remotos, além de analisar o impacto no tempo dos testes por causa desse overhead. Por fim, executamos novamente esses testes com a comunicação, porém dessa vez a partir de outro computador, com outro sistema operacional, Windows 10, conectado através da rede local, via conexão wireless. Esse computador não possuía a implementação do pacote *server*, tendo disponível apenas a interface remota, e mostrou na prática o funcionamento do RMI para comunicação remota através da rede, além de mostrar a queda no desempenho graças ao gargalo da rede. A tabela a seguir mostra o tempo total de cada uma dessas execuções:

Comunicação	Tempo total
Sem comunicação - invocação direta do método de ordenação	4.089s
Comunicação com o servidor via RMI na mesma máquina	10.475s
Comunicação com o servidor através da rede	52.973s

Através desses testes fica nítido a diferença no desempenho devido à comunicação via RMI e, principalmente, à rede, reforçando que ela será frequentemente um obstáculo durante o desenvolvimento de sistemas distribuídos.

5 Conclusão

O RMI é um facilitador na comunicação de softwares distribuídos e mostrou-se prático, visto que a comunicação do servidor e do cliente foi facilmente utilizada. O projeto desenvolvido foi testado tanto localmente quanto por duas máquinas com sistemas operacionais diferentes (Windowns 10 e Ubuntu 16.04 LTS), via conexão wireless, sem grandes problemas, porém mostrando os desafios que surgem para se manter o desempenho durante o desenvolvimento de sistemas distribuídos, causados principalmente pela rede.