Implementação de Sistemas Distribuidos Utilizando RMI

Aécio de O. Souza, Aurelicio Pereira , Gleidson Ramos, Guilherme França , Luiz Sacramento

[1]Sistemas de Informação – Universidade do Estado da Bahia (UNEB)
Caixa Postal 41150000 – Salvador – BA – Brazil

Abstract. *This article presents the implementation of a distributed system with its desired characteristics for this type of system, aiming at scalability, openness, transparency and resource sharing, as they bring with it a lot of significant benefit, such as the increase in performance provided by parallel computing and the ability to utilize resources distributed throughout the system. For this possibility of communication to occur between the computers of a system, we will use a mechanism called JAVA RMI, as it provides programmers with the necessary tools for distributed programming, thus offering an environment conducive to the efficient implementation of cryptographic algorithms in a scalable way.*

Keywords *– Parallel Computing, Java RMI, Encryption Algorithms, Distributed Systems.*

Resumo. *Este artigo apresenta a implementação de um sistema distribuído com suas características desejadas para esse tipo de sistema, visando a escalabilidade, abertura, transparência e compartilhamento de recursos, pois trazem consigo muito benefício de forma significativa, como o aumento do desempenho proporcionado pela computação paralela e a capacidade de utilizar recursos distribuídos em todo o sistema. Para que ocorra essa possibilidade de comunicação entre os computadores de um sistema utilizaremos um mecanismo chamado JAVA RMI pois proporciona aos programadores ferramentas necessárias para a programação distribuída oferecendo assim um ambiente propicio para a implementação eficiente de algoritmos de criptografia de forma escalável.*

Keywords *– Computação Paralela, Java RMI, Algoritmos de Criptografia, Sistemas Distribuídos*

1. Introdução

Sabemos que a cada dia mais os desafios na área da computação estão cada vez mais complexos e exigentes. Com o avanço tecnológico os computadores aumentaram a magnitude desses desafios trazendo soluções viáveis e interessantes. Dessa forma vemos de maneira bem otimista e promissora a utilização de sistemas distribuídos com processamento paralelo permitindo assim uma distribuição e divisão de processamento entre maquinas que compõe um sistema, possibilitando assim a redução de custos de hardware e proporcionando maior desempenho ao processamento.

Para a efetivação de um sistema distribuído, e imprescindível gerar uma forma de comunicação entre computadores que estão interconectados em uma rede. Existem vários métodos a serem utilizados na computação tais como SOCKETS e RPC mais nesse trabalho utilizamos um método exclusivo para a para a programação distribuída que utiliza a linguagem de programação Java que é o **RMI (Remote Method Invocation)**, pois disponibiliza as ferramentas fundamentais para implementar sistemas distribuídos.

Ao estudar sistemas distribuídos e computação paralela, com o suporte do Java RMI, entendemos que existem inúmeras possibilidades para lidar com desafios computacionais de forma mais eficiente e econômica, pois a capacidade de dividir tarefas e utilizar recursos distribuídos proporciona ganhos de forma significativa tanto em desempenho quanto em eficiência, permitindo assim a resolução de problemas de ordem complexa e avançada em diversos campos na área da computação.

2. Implementação de Sistemas Distribuídos

Definimos como Sistema distribuído um agrupamento de vários computadores que estão interligados por meio de uma rede de alta velocidade onde podem se comunicar. Aonde seu principal objetivo parte da necessidade que o usuário tem de compartilhar recursos. A Internet é um exemplo de sistemas distribuídos pois comunica e disponibiliza recursos para usuários do mundo todo. Neste momento identificamos algumas características presentes em um sistema distribuído, como: falhas de componentes individuais, concorrência e inexistência de um relógio global.Por ter estas características, existem alguns desafios que estão no caminho da implementação de um sistema distribuído.Neste momento definiremos a implementação levando em consideração as seguintes características desejadas:

2.1. Escalabilidade

Adotaremos uma abordagem escalável para permitir que o aumento do número de recursos e de usuários sem que ocorra a perda de desempenho do sistema. Utilizaremos algoritmos de balanceamento de carga, permitindo assim a distribuição da carga de trabalho de forma balanceada entre os servidores.

2.2. Abertura

O sistema será desenhado para ser aberto e extensível, possibilitando a integração com outros sistemas ou serviços externos. Utilizaremos APIs (Application Programming Interfaces) de forma bem definidas com interfaces flexíveis facilitando assim a integração e comunicação com outros sistemas.

2.3. Transparência

Característica fundamental em sistemas distribuídos a transparência leva ao usuário a percepção de está utilizando um sistema único de maneira coerente, por isso adotaremos mecanismos para ocultarmos a complexidade da distribuição possibilitando que os usuários interajam com o sistema como se fosse um sistema centralizado. Isso engloba transparência no acesso, replicação de recursos migração e localização.

2.4. Compartilhamento de Recursos

O sistema permitirá o compartilhamento de recursos entre os componentes distribuídos. Serão utilizados mecanismos de sincronização e controle de concorrência para garantir o acesso adequado aos recursos compartilhados, evitando conflitos e inconsistências.

A presença de computadores e tecnologias diversas em um sistema distribuído suscita a preocupação em lidar com as diferenças, a fim de promover a colaboração entre os componentes do sistema na troca de informações e, ao mesmo tempo, facilitar a programação para os desenvolvedores. Essa responsabilidade é atribuída a uma camada intermediária chamada middleware, posicionada entre a camada de aplicação e as camadas de hardware e sistema operacional. O middleware desempenha um papel crucial ao harmonizar as disparidades tecnológicas e garantir a interoperabilidade eficiente entre os componentes distribuídos do sistema. Além disso, ele oferece uma abstração das complexidades subjacentes.

Figura 1. Camada de middleware em um sistema distribuído

Em resumo, o middleware desempenha um papel fundamental na implementação de sistemas distribuídos, permitindo a colaboração entre componentes, abstraindo diferenças tecnológicas, fornecendo escalabilidade, transparência e facilitando o compartilhamento de recursos.

Para construir um sistema distribuído precisamos superar desafios únicos para garantir sua efetividade. É também de suma importância que um sistema distribuído tenha uma finalidade bem definida. Entre as finalidades comuns de um sistema distribuído estão sistemas para fins computacionais, sistemas para disponibilizar informação e sistemas embutidos. Não obstante da finalidade, a construção de um sistema distribuído requer considerações cuidadosas de design e arquitetura para garantir o cumprimento dos requisitos de escalabilidade, desempenho, confiabilidade e segurança.

3. Java RMI

O **RMI** possibilita ao programador se concentrar especificamente nas funcionalidades do sistema, de forma que não é necessário se preocupar com a diferença entre tecnologias de hardware e redes. No RMI a realização da comunicação é feita através da invocação de métodos remotos, aonde o cliente faz a utilização dos serviços que estão disponíveis em um servidor. Em uma rede com vários computadores é muito interessante a utilização do Java RMI, pois após o processamento é possível realizar em dois ambientes de execução diferentes uma análise comparativa, sendo um utilizando o RMI

como método de comunicação e o outro utilizando outro método de comunicação. Esta análise torna possível identificar diferenças de desempenho determinado assim quando a utilização do Java RMI é mais vantajosa.

Na implementação de um sistema distribuído utilizando a linguagem Java, é adotado o conceito de orientação a objetos. Nesse contexto, uma classe é responsável por conter todos os serviços disponibilizados pelo sistema. Esses serviços estarão acessíveis para invocações por parte dos clientes, que podem utilizar objetos distribuídos para interagir com o sistema. Dessa forma, a arquitetura orientada a objetos facilita a organização e o acesso aos serviços do sistema distribuído.

Ou seja quando um cliente se vincula a um objeto distribuído acontece a criação de duas entidades. No cliente sera criado um proxy denominado stub, situado no espaço de endereçamento do cliente. Sendo o stub um representante local realiza a implementação na mesma interface que o objeto que está representando, e sua responsabilidade é fazer a serialização dos parâmetros de uma invocação e fazer o envio ao servidor.No lado do servidor um objeto denominado skeleton faz a comunicação direta com o cliente por meio do stub, o skeleton tem a responsabilidade de decodificar os parâmetros recebidos na invocação e repassar ao servidor para que o método mais apropriado para aquela invocação seja executado, e mais tarde, serializar a resposta que será encaminhada ao stub.

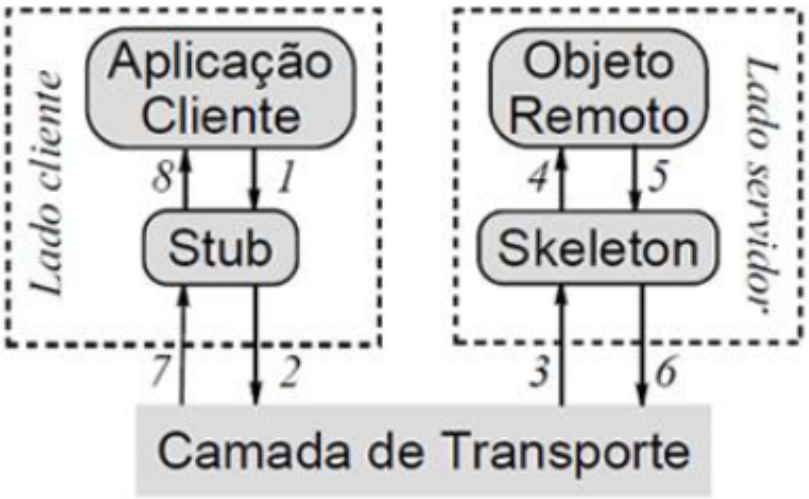

Fig. Estrutura de um cliente e um servidor utilizando Java RMI

4. Implementação prática: Construindo um sistema distribuído com Java RMI

Neste momento após analisarmos e discutir alguns dos conceitos e fundamentos do Java RMI (Remote Method Invocation), é chegada a hora de explorarmos a implementação prática desse middleware em um exemplo de código. Iremos abordar nesta seção a construção simples de um sistema distribuído fazendo a utilização do Java RMI como tecnologia de comunicação entre os componentes.

Para facilitar a compreensão, vamos considerar um cenário em que temos um sistema de gerenciamento de um carrinho em uma loja online. O sistema é composto por um servidor central que armazena as informações e permite que os clientes se conectem ao servidor, obtenham referências ao objeto remoto e interajam com o carrinho de compras remoto, chamando os métodos definidos na interface.

Neste exemplo o código envolverá a criação de quatro componentes: a RemoteCart, RemoteCartImpl além da implementação do servidor e a implementação do cliente. Com essa transição para a implementação do código, estaremos explorando na prática os conceitos discutidos anteriormente. O exemplo fornecerá uma base sólida para entendermos como o Java RMI permite a criação de sistemas distribuídos robustos e eficientes. Vamos agora mergulhar nos detalhes da implementação do código, explorando cada componente e suas funcionalidades específicas.

5. Classe Server

Começaremos com a classe Server que é um exemplo de implementação de um servidor **RMI (Remote Method Invocation).** Ele cria um servidor que instancia e disponibiliza um objeto remoto do tipo **RemoteCartImpl**, representando um carrinho de compras remoto.

```java
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;

public class RMIServer {
    public static void main(String[] args) {
        try {
            // Cria uma instância do carrinho de compras remoto
            RemoteCartImpl remoteCart = new RemoteCartImpl();

            // Cria o registro RMI na porta 1099
            Registry registry = LocateRegistry.createRegistry(1099);

            // Registra o carrinho de compras remoto no registro RMI
            registry.bind("RemoteCart", remoteCart);

            System.out.println("Servidor RMI iniciado.");

        } catch (Exception e) {
            System.err.println("Erro no servidor RMI: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

1. **public static void main(String[] args):** Este é o método principal do programa do servidor RMI.
2. **RemoteCartImpl remoteCart = new RemoteCartImpl():** Esta linha cria uma instância do objeto remoto RemoteCartImpl, que representa o carrinho de compras remoto.
3. **Registry registry = LocateRegistry.createRegistry(1099):** Esta linha cria um registro RMI na porta 1099. O método **LocateRegistry.createRegistry()** cria um registro RMI local que pode ser usado para registrar e armazenar objetos remotos.
4. **registry.bind(”RemoteCart”, remoteCart):** Esta linha registra o objeto remoto remoteCart no registro RMI, associando-o ao nome ”RemoteCart”. O método **registry.bind()** vincula o objeto remoto ao nome fornecido no registro RMI.
5. **System.out.println(”Servidor RMI iniciado.”):** Esta linha imprime uma mensagem indicando que o servidor RMI foi iniciado com sucesso.
6. O bloco **catch** captura e trata qualquer exceção que possa ocorrer durante a execução do servidor RMI, imprimindo informações de erro no console.

6. Classe Client

Ilustra como um cliente pode se comunicar com um objeto remoto do tipo **RemoteCart** para adicionar, remover e obter itens de um carrinho de compras remoto.

```java
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.util.List;

public class RMIClient {
    public static void main(String[] args) {
        try {
            // Localiza o registro RMI na porta 1099
            Registry registry = LocateRegistry.getRegistry("localhost", 1099);

            // Obtém a referência ao carrinho de compras remoto do registro RMI
            RemoteCart remoteCart = (RemoteCart) registry.lookup("RemoteCart");

            // Adiciona itens ao carrinho de compras
            remoteCart.addItem("Item 1");
            remoteCart.addItem("Item 2");

            // Obtém a lista de itens do carrinho de compras
            List<String> items = remoteCart.getItems();
            System.out.println("Itens no carrinho de compras: " + items);

            // Remove um item do carrinho de compras
            remoteCart.removeItem("Item 1");

            // Obtém a lista de itens atualizada do carrinho de compras
            items = remoteCart.getItems();
            System.out.println("Itens no carrinho de compras: " + items);

        } catch (Exception e) {
            System.err.println("Erro no cliente RMI: " + e.toString());
            e.printStackTrace();
        }
    }
```

1. **public static void main(String[] args):** Este é o método principal do programa cliente RMI.
2. **Registry registry = LocateRegistry.getRegistry(”localhost”, 1099):** Essa linha localiza o registro RMI na porta 1099 no localhost. O método **LocateRegistry.getRegistry()** retorna uma referência ao registro RMI que pode ser usado para procurar e obter referências a objetos remotos registrados.
3. **RemoteCart remoteCart = (RemoteCart) registry.lookup(”RemoteCart”):** Essa linha obtém uma referência ao objeto remoto do tipo **RemoteCart** registrado no registro RMI. O método **registry.lookup()** recebe uma string que corresponde ao nome sob o qual o objeto remoto foi registrado no registro e retorna uma referência a ele. É necessário fazer um cast para o tipo **RemoteCart** para poder invocar os métodos definidos nessa interface.
4. **remoteCart.addItem(”Item 1”) e remoteCart.addItem(”Item 2”):** Essas linhas chamam o método **addItem** no objeto remoto **remoteCart**, passando os itens ”Item 1”e ”Item 2”para serem adicionados ao carrinho de compras remoto.
5. **List¡String¿ items = remoteCart.getItems():** Essa linha chama o método **getItems** no objeto remoto **remoteCart** para obter a lista de itens atual no carrinho de compras remoto. O resultado é armazenado na variável **items**, que é do tipo **List¡String¿.**
6. **System.out.println(”Itens no carrinho de compras: ”+ items):** Essa linha imprime a lista de itens do carrinho de compras no console do cliente.
7. **remoteCart.removeItem(”Item 1”)**: Essa linha chama o método **removeItem** no objeto remoto **remoteCart** para remover o item ”Item 1”do carrinho de compras remoto.
8. **items = remoteCart.getItems()**: Essa linha chama novamente o método **getItems** no objeto remoto **remoteCart** para obter a lista de itens atualizada no carrinho de compras remoto.
9. **System.out.println(”Itens no carrinho de compras: ”+ items):** Essa linha imprime a lista de itens atualizada do carrinho de compras no console do cliente.
O código lida com exceções que podem ocorrer durante a comunicação RMI e imprime informações de erro se ocorrer algum problema.

7. Classe RemoteCart

O código define a interface de chamada RemoteCart aqui está a descrição do código:

```java
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

public interface RemoteCart extends Remote {
    void addItem(String item) throws RemoteException;
    void removeItem(String item) throws RemoteException;
    List<String> getItems() throws RemoteException;
}
```

1. **public interface RemoteCart extends Remote:** Esta linha declara a interface **RemoteCart** que estende a interface **Remote**. A interface **Remote** é uma interface marcadora

especial usada no RMI para indicar que uma interface estende **Remote** e seus métodos podem ser chamados remotamente.
2. **void addItem(String item) throws RemoteException:** Este método, sem retorno **(void),** chamado **addItem,** aceita um parâmetro do tipo **String** chamado **item.** Ele declara que pode lançar uma exceção **RemoteException.** Esse método é responsável por adicionar um item ao carrinho de compras remoto.
3. **void removeItem(String item) throws RemoteException:** Este método, sem retorno **(void),** chamado **removeItem,** aceita um parâmetro do tipo **String** chamado **item.** Ele declara que pode lançar uma exceção **RemoteException.** Esse método é responsável por remover um item do carrinho de compras remoto.
4. List¡String¿ **getItems() throws RemoteException:** Este método retorna um objeto do tipo List¡String¿, chamado **getItems,** e declara que pode lançar uma exceção **RemoteException.** Esse método é responsável por retornar a lista de itens presentes no carrinho de compras remoto. Em resumo, essa interface **RemoteCart** define três métodos que podem ser chamados remotamente em um objeto do tipo **RemoteCart.** Os métodos permitem adicionar itens ao carrinho de compras remoto, remover itens e obter a lista atualizada de itens presentes no carrinho. Todos os métodos podem lançar uma exceção **RemoteException,** que é uma exceção comum usada no RMI para lidar com erros de comunicação e chamadas remotas.

8. Classe RemoteCartImpl

Veremos agora a implementação da interface **RemoteCart na classe RemoteCartImpl:**

```java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.ArrayList;
import java.util.List;

public class RemoteCartImpl extends UnicastRemoteObject implements RemoteCart {
    private List<String> items;

    public RemoteCartImpl() throws RemoteException {
        items = new ArrayList<>();
    }

    public void addItem(String item) throws RemoteException {
        items.add(item);
        System.out.println("Item adicionado: " + item);
    }

    public void removeItem(String item) throws RemoteException {
        items.remove(item);
        System.out.println("Item removido: " + item);
    }

    public List<String> getItems() throws RemoteException {
        return items;
    }
}
```

1. **public class RemoteCartImpl extends UnicastRemoteObject implements RemoteCart:** Esta linha declara a classe **RemoteCartImpl,** que implementa a interface **RemoteCart** e estende a classe **UnicastRemoteObject**. A classe **UnicastRemoteObject**

é uma classe fornecida pelo RMI que facilita a exportação de objetos remotos.
2. **private** List¡String¿ **items**: Esta linha declara uma variável privada do tipo List¡String¿ chamada **items**, que será usada para armazenar os itens do carrinho de compras.
3. **public RemoteCartImpl() throws RemoteException:** Este é o construtor da classe **RemoteCartImpl**. Ele declara que pode lançar uma exceção **RemoteException.** O construtor é responsável por inicializar a lista de itens do carrinho de compras como um novo **ArrayList.**
4. **public void addItem(String item) throws RemoteException:** Este método, sem retorno **(void), chamado addItem**, aceita um parâmetro do tipo **String** chamado **item.** Ele declara que pode lançar uma exceção **RemoteException.** Esse método adiciona o item fornecido à lista de itens do carrinho de compras e imprime uma mensagem no console informando que o item foi adicionado.
5. **public void removeItem(String item) throws RemoteException:** Este método, sem retorno (void), chamado **removeItem,** aceita um parâmetro do tipo **String** chamado **item**. Ele declara que pode lançar uma exceção **RemoteException.** Esse método remove o item fornecido da lista de itens do carrinho de compras e imprime uma mensagem no console informando que o item foi removido.
6. **public** List¡String¿ **getItems() throws RemoteException:** Este método retorna um objeto do tipo List¡String¿, chamado **getItems**, e declara que pode lançar uma exceção **RemoteException.** Esse método retorna a lista de itens do carrinho de compras.
Em resumo, a classe **RemoteCartImpl** implementa a interface **RemoteCart**, definindo os métodos necessários para adicionar, remover e obter itens do carrinho de compras remoto. Ela mantém a lista de itens em uma variável privada e realiza operações nessa lista conforme solicitado pelos métodos definidos na interface. Além disso, o código inclui a lógica para imprimir mensagens informativas no console quando os itens são adicionados ou removidos.

9. Conclusões parciais

Com base nos estudos realizados sobre as referências bibliográficas concluímos ate o presente momento que o desafio principal neste projeto é paralelizar o algoritmo criptográfico para que se possa obter o melhor desempenho. Nosso artigo traz a utilização do Java RMI (Remote Method Invocation) para comunicação em sistemas distribuídos. Pois o RMI torna fácil a programação, pela sua semelhança à programação sequencial. Todavia, a tarefa de distribuir as partes do algoritmo entre as máquinas que compõem o sistema pode afetar o desempenho.Objetivamos assim analisar e executar um algoritmo de forma paralela usando RMI determinando se e vantajoso o ganho ou perda de desempenho em sua execução. Esperamos concluir se a abordagem é viável e apresentar os resultados para ajudar projetos futuros que utilizem algoritmos paralelos com RMI, fornecendo informações relevantes para trabalhos relacionados.

Referências

G. Coulouris, J. Dollimore andT. Kindberg, “Sistemas Distribuídos”, 4 ed., São Paulo: Artmed, 2007.

A. S. Tanenbaum, M. V. Steen, “Sistemas Distribuidos”,2 ed., Rio de Janeiro: Prentice-Hall,2007.

H. V. Bacellar, “Cluster: Computação de Alto Desempenho”,Campinas: Instituto de Computação,2010.