Exemplos práticos do uso de RMI em sistemas distribuídos

Elder de Macedo Rodrigues, Guilherme Montez Guindani, Leonardo Albernaz Amaral[1]
Fábio Delamare[2]

Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação
Sistemas Distribuídos
Porto Alegre/RS – Brasil – CEP 96616-900

Resumo:

O objetivo deste artigo é explorar exemplos de programas que utilizem a estrutura RMI (Remote Method Invocation). Para isso, serão abordados dois exemplos práticos distribuídos: um chat e uma calculadora, traçando-se um paralelo entre suas implementações sockets e RMI na linguagem Java.

Palavras-chave: *Java, Sockets, RMI.*

1. Introdução

Sistemas distribuídos descritos em linguagem Java, podem ser implementados utilizando-se diversas estruturas de comunicação entre processos, no entanto, as estruturas mais usadas são sockets e o RMI [1]. Um socket, por definição, é um canal de comunicação entre processos que estabelece uma conexão entre eles na forma de cliente-servidor. Por meio de sockets, os computadores podem trocar informações através de uma rede. Para se criar uma conexão entre dois computadores, um socket deve ser definido, basicamente, através das seguintes informações: endereço IP do servidor, porta onde se encontra o serviço solicitado no servidor, endereço IP do cliente, porta através da qual o cliente solicita o serviço [2].

Já o RMI (*Remote Method Invocation*), tem como objetivo, permitir aos programadores o desenvolvimento de aplicações distribuídas em Java com a mesma sintática e semântica usada em programas não distribuídos. Para isso, é necessário fazer com que os programas Java que rodam em uma JVM (máquina virtual) tenham acesso a programas em máquinas virtuais distribuídas, que no caso do RMI é conhecido como "invocação de métodos remotos". A arquitetura RMI define como os objetos se comportam, como e quando exceções podem ocorrer, como a memória é manipulada e como os parâmetros são passados e retornados em métodos remotos [1, 3].

A partir destas definições foram implementados dois exemplos de aplicações distribuídas: um **chat** e uma **calculadora**, ambos baseados nas duas estruturas de comunicação: **sockets** e **RMI**. O objetivo é traçar um paralelo entre estas implementações na linguagem Java e tentar identificar algumas vantagens e desvantagens do uso de RMI em aplicações distribuídas. No Capítulo 2 será abordado a aplicação de chat. No Capítulo 3 será abordado a calculadora. E por fim, no Capítulo 4 têm-se as conclusões do artigo.

[1] elder.macedo, guilherme.guindani, lamaral @inf.pucrs.br
[2] fldelama@gmail.com

2. Aplicação do Chat

A aplicação do Chat trata-se de um programa cliente-servidor o qual permite que diversos usuários realizem um bate-papo em tempo real. A comunicação (troca de mensagens) deve ser centralizada no servidor, ou seja, todas as mensagens enviadas pelos usuários do bate papo devem ser enviadas primeiro para o servidor, o qual deve repassá-las para os usuários envolvidos no chat. Basicamente, toda a abstração e controle dessa comunicação devem ser de responsabilidade do servidor.

2.1 Chat usando Sockets

Neste exemplo será abordado a implementação do aplicativo de chat usando sockets como meio de comunicação. O aplicativo e baseado em uma arquitetura do tipo cliente-servidor, onde o servidor recebe as mensagens de um cliente e as repassa para os demais clientes conectados a ele. A Figura 1 mostra o diagrama de classes da aplicação em sockets.

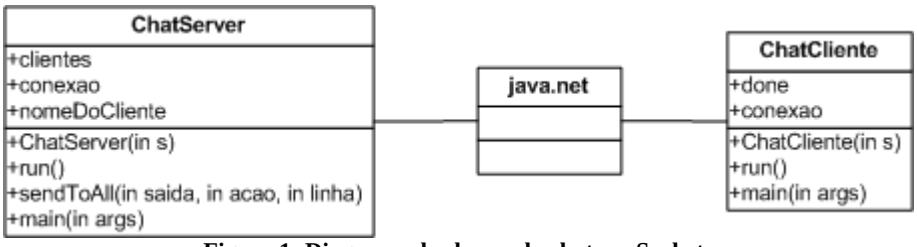


Figura 1: Diagrama de classes do chat em Sockets

2.1.1 Servidor:

Servidor implementado em Multithreads em que cada novo cliente conectado a ele dispara uma thread do servidor para tratar as mensagens enviadas por este. O servidor armazena cada canal de saída (socket do cliente conectado a ele) em um vetor de sockets.

A execução do servidor (***main*** do servidor na figura 1) ocorre da seguinte forma:

- O servidor cria uma socket na porta 5000;
- Aguarda uma conexão de algum cliente (**conexão**);
- Ao receber um pedido de conexão, deve aceita-la;
- Dispara uma nova thread (***run***) para o tratamento de mensagens do cliente;
- Esta thread, primeiramente, recebe o nome (*nickname*) do cliente. Este nome não pode ser NULL, já que um nome vazio indica ao servidor que o cliente desconectou;
- Em seguida armazena o socket do cliente para o repasse de mensagens no seu vetor de clientes ativos (**clientes**);
- A seguir lê as mensagens enviadas pelo cliente e as repassa aos demais clientes conectados a ele através do método ***sendToAll***;
- Ao detectar uma mensagem nula (NULL) o servidor informa aos demais clientes que este desconectou, removendo-o do vetor de clientes ativos.

2.1.2 Cliente:

Cliente implementado em Multithreads em que existe uma thread para ler a mensagem do teclado e outra que imprime as mensagens recebidas do servidor na tela.

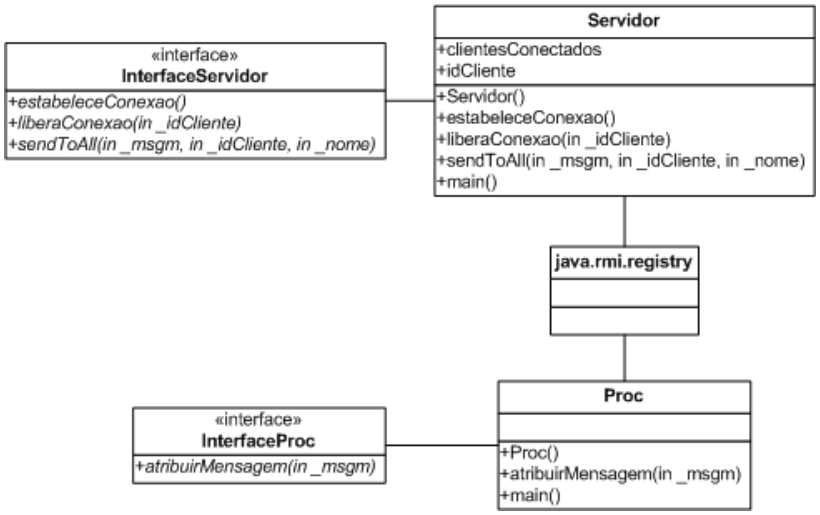
A execução do cliente (***main*** do cliente na figura 1) ocorre da seguinte forma:

- O cliente cria uma conexão com o servidor (socket) na porta 5000;
- Pede que o usuário informe o nome (*nickname*) a ser utilizado no chat e o envia ao servidor;
- Inicia a thread (***run***) de recebimento das mensagens (strings de texto) recebidas do servidor e as imprime na tela do usuário. Caso esta mensagem seja nula (NULL), informa ao usuário que a conexão com o servidor foi quebrada;
- A outra thread (execução padrão da ***main***) realiza a leitura das mensagens do teclado do usuário e as envia ao servidor;
- Caso a mensagem do servidor seja nula (NULL), a thread de recebimento se encarrega de notificar o término do programa para a ***main*** através da variável compartilhada ***done***.

2.2 Chat usando RMI

Quando se desenvolve uma aplicação em Java RMI, alguns elementos básicos devem criados, tais como:

- Uma interface que disponibilize os métodos no servidor.
- Uma classe que fique localizada na JVM (*Java Virtual Machine*) do servidor e que implemente os métodos definidos na interface.
- Classes que implementem o protocolo de comunicação (*Skel* e *Stub*)[3] e que sejam responsáveis por fazer com que a chamada de um método no cliente seja passada ao servidor de maneira transparente, assim como fazer com que o servidor responda de maneira conveniente a essa chamada, passando de volta ao cliente o valor de retorno.
- Um programa cliente que invoque os métodos remotos do servidor.
- Um serviço de nomes (*rmiregistry*) responsável por informar ao cliente onde está o servidor e que relacione corretamente a implementação deste ao stub do cliente.



[3] As classes Skel e Stub não precisam ser programadas manualmente, pois são geradas automaticamente por uma ferramenta própria do SDK (rmic).

Figura 2: Diagrama de classes do chat em RMI.

No caso da versão em RMI da aplicação do chat, alguns desses elementos podem ser conferidos através do diagrama de classes da figura 2.

Foram definidas duas interfaces, a interface do servidor (**InterfaceServidor**) e a interface do cliente (**InterfaceProc**), além de duas classes que implementam essas interfaces, a classe **Servidor** e a classe **Proc**. No lado do servidor existem três métodos que implementam a interface definida e que possuem as seguintes funcionalidades:

- **estabeleceConexao():** método usado para que o servidor identifique quais clientes estão conectados no chat, além de criar uma referência individual por processo dos serviços que estes disponibilizam (linha 38 da figura 3). Com essa referência, torna-se possível ao servidor chamar os métodos dos processos clientes.

```
34      public int estabeleceConexao() throws RemoteException{
35          idCliente ++;
36
37          try{
38              //Busca referência dos serviços do cliente;
39              clientesConectados[idCliente] = (InterfaceProc) Naming.lookup("cliente");
40
41          }catch(Exception e) {
42              System.out.println( "Exception: " + e.toString() );
43          }
44
45          return idCliente;
46      }
```

Figura 3: Método estabeleceConexao()

- **liberaConexao():** método usado para desfazer a conexão estabelecida por um cliente através do método **estabeleceConexao()**. Este método exclui o cliente conectado do vetor "**clientesConectados**" (linha 53 da figura 4).

```
48      public String liberaConexao(int _idCliente) throws RemoteException{
49
50          //Loop para excluir usuário do chat;
51          for(int i = 0; i < clientesConectados.length; i++){
52              if(i == _idCliente){
53                  clientesConectados[idCliente] = null;
54              }
55          }
56          idCliente --;
57
58          return "Conexao liberada!";
59      }
```

Figura 4: Método liberaConexao()

- **sendToAll():** método usado para enviar a mensagem digitada por um usuário do chat para todos os demais usuários. Esse método é acessado remotamente pelo cliente (classe Proc) o qual envia como parâmetros a mensagem digitada (_msgm), o seu id (_idCliente) conseguido no método estabeleceConexao(), e o nome do usuário do chat (_nome).

Quando esse método é executado no servidor, ele acessa o método remoto **atribuirMensagem()** (na linha 67 da figura 5) disponibilizado pelos clientes através da classe **Proc,** o que permite o envio da mensagem aos demais usuários.

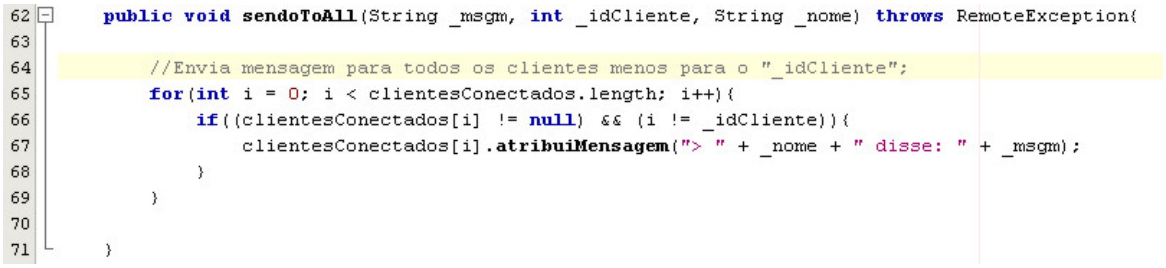


Figura 5: Método sendToAll()

Já no lado do cliente, foi definido apenas o método **atribuirMensagem(),** o qual é responsável por receber a mensagem enviada pelos usuários do chat e escrever na tela a string correspondente (figura 6).

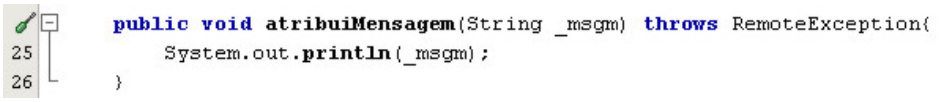


Figura 6: Método atribuirMensagem()

2.2.1 Comportamento da execução do chat em RMI

Abaixo será apresentado o comportamento da execução da aplicação do chat em RMI.

2.2.1.1 Servidor

O servidor (classe **Servidor**) é basicamente um processo que deve ficar sempre executando e disponibilizando seus métodos até que seja encerrado. Ele é uma classe que implementa os métodos remotos definidos na classe **InterfaceServidor** e funciona como um servidor de RMI. Para isso, seu comportamento pode ser definido como:

- Através da execução do método **main()**, este chama o método construtor da classe (linha 81 da figura 7), construindo as implementações do servidor e registrando os métodos remotos no servidor de nomes do RMI (linha 83 da figura 7).

```
74      /** Método principal do Servidor. Chama o método construtor da classe */
75      public static void main(String args[]) throws RemoteException{
76
77          try {
78              //Disponibiliza os serviços do servidor;
79              System.setSecurityManager(new RMISecurityManager());
80              System.out.println("Construindo as implementacoes do Servidor...");
81              Servidor servidor = new Servidor();
82              System.out.println("Registrando as implementacoes...");
83              Naming.rebind("servidor", servidor);
84              System.out.println("Servidor iniciado!");
85
86              //System.exit(0);
87
88          }
89          catch( Exception e ) {
90              System.out.println( "Problema: " + e );
91          }
92
93      }
```

Figura 7: Método Main() do servidor

- Após a execução do método **main()**, o servidor que já disponibilizou os seus métodos remotos, fica rodando infinitamente até que alguém encerre o seu processo de execução.

2.2.1.2 Cliente

O processo cliente (classe **Proc**), ao contrário do servidor, é um processo que interage com o usuário do chat, uma vez que ele lê os dados digitados (linhas 56, 57, 58 e 61 da figura 9) e imprime na tela as mensagens digitadas e recebidas (linha 60 da figura 9). O funcionamento desse processo é o seguinte:

- Através da execução do método **main()**, o método construtor da classe é executado (linha 42 da figura 8) construindo as implementações do cliente. Com as implementações criadas, o processo disponibiliza os seus serviços (linha 43 da figura 8) registrando os seus métodos remotos no servidor de nomes do RMI. Após isso, o processo busca no serviço de nomes os serviços disponibilizados pelo servidor e cria uma referência de acesso a tais serviços (linhas 46 e 47 da figura 8). Tendo-se essa referência criada, o processo chama o método remoto **estabeleceConexao()** (linha 49 da figura 8) e se identifica no servidor, recebendo como retorno o **idCliente** (linha 49 da figura 8). Cria-se então o objeto de leitura do teclado (linha 53 da figura 8) e inicia-se um laço onde o processo: solicita o nome do usuário (linhas 56 da figura 9) (i), lê os dados informados pelo usuário (linhas 57, 58 e 61 da figura 9) (ii), e fica enviando a mensagem digitada para os demais processos (linha 60 da figura 9) (iii) até que o usuário digite “end”, fazendo com que o processo seja interrompido.

```
30      /** Método principal do "processo". */
31      public static void main(String args[]) throws RemoteException{
32
33          int num1, num2, idCliente;
34          char oper;
35          double resultado;
36          String msgmDigitada;
37
38          try{
39
40              //Disponibiliza os serviços do cliente;
41              System.setSecurityManager(new RMISecurityManager());
42              Proc cliente = new Proc();
43              Naming.rebind("cliente", cliente);
44
45              //Busca referência dos serviços do servidor;
46              Remote referenciaRemota = Naming.lookup("servidor");
47              InterfaceServidor a = (InterfaceServidor) referenciaRemota;
48
49              idCliente = a.estabeleceConexao();
50              System.out.println("Id do cliente: " + idCliente);
51
52              //Cria objeto de leitura do console (teclado);
53              BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
```

Figura 8: Método Main() do cliente parte 1

```
53              BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
54
55              //Loop para cliente ficar lendo do teclado e enviando para todos;
56              System.out.print("Entre com um nome: ");
57              String nome = teclado.readLine();
58              msgmDigitada = teclado.readLine();
59              while(msgmDigitada.compareTo("end") != 0){
60                  a.sendoToAll(msgmDigitada, idCliente, nome);
61                  msgmDigitada = teclado.readLine();
62              }
63
64              System.out.print("Saiu...");
65              System.out.println(a.liberaConexao(idCliente));
66              System.exit(1);
67
68          }catch(Exception e) {
69              System.out.println( "Exception: " + e.toString() );
70          }
71
72      }
```

Figura 9: Método Main() do cliente parte 2 (continuação)

3. Aplicação da Calculadora

A aplicação da calculadora trata-se de um programa cliente-servidor o qual permite que diversos clientes executem operações aritméticas remotamente em um servidor. O cálculo das operações requeridas pelo cliente deve ser executado no servidor, desta forma o cliente apenas recebe do usuário os operandos e a operação, e envia uma mensagem com este conteúdo ao servidor.

3.1 Calculadora usando Sockets

Neste exemplo será abordada a implementação do aplicativo da calculadora usando sockets como meio de comunicação. O aplicativo é baseado em um modelo cliente-servidor, onde o servidor

recebe os operandos e a operação do cliente, executa o cálculo da operação e retorna o resultado ao cliente. A Figura 10 mostra o diagrama de classes da aplicação em sockets.

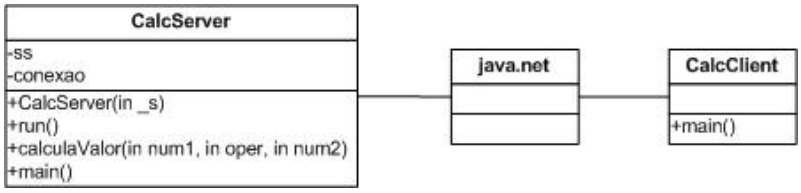


Figura 10: Diagrama de classe da calculadora em sockets

3.1.1 Servidor

Servidor implementado em Multithreads, em que cada novo cliente conectado a ele dispara uma thread para executar a operação enviada por este.

A execução do servidor (***main*** do servidor na figura 10) ocorre da seguinte forma:

- O servidor cria uma socket na porta 5000;
- Aguarda uma conexão de algum cliente (**conexão**);
- Ao receber um pedido de conexão, deve aceita-la;
- Dispara uma nova thread (***run***) para o tratamento de mensagens do cliente;
- Esta thread executa a operação aritmética enviada pelo cliente sobre os operandos recebidos da mesma forma (**calculaValor**);
- A seguir envia o resultado da operação de volta para o cliente.

3.1.2 Cliente

O cliente é responsável por adquirir do usuário, os operandos e a operação a ser executada, montar uma mensagem com estas informações e a enviar para o servidor. Ao receber a reposta, o cliente deve imprimir o resultado na tela do usuário.

A execução do cliente (***main*** do cliente na figura 10) ocorre da seguinte forma:

- O cliente cria uma conexão com o servidor (socket) na porta 5000;
- Pede que o usuário informe os operandos e a operação a ser executada;
- Monta uma mensagem com estes dados para ser enviada ao servidor;
- Envia esta mensagem e aguarda a resposta do servidor;
- Ao receber a resposta, imprime na tela o resultado calculado pelo servidor.

3.2 Calculadora usando RMI

No caso da versão em RMI da aplicação da calculadora, os elementos envolvidos podem ser conferidos na figura 11. Nessa aplicação foi definida apenas uma interface (**InterfaceServidor**) e duas classes, a classe **Servidor**, que implementa o método definido na interface, e a classe **Proc**, que acessa remotamente o método do servidor.

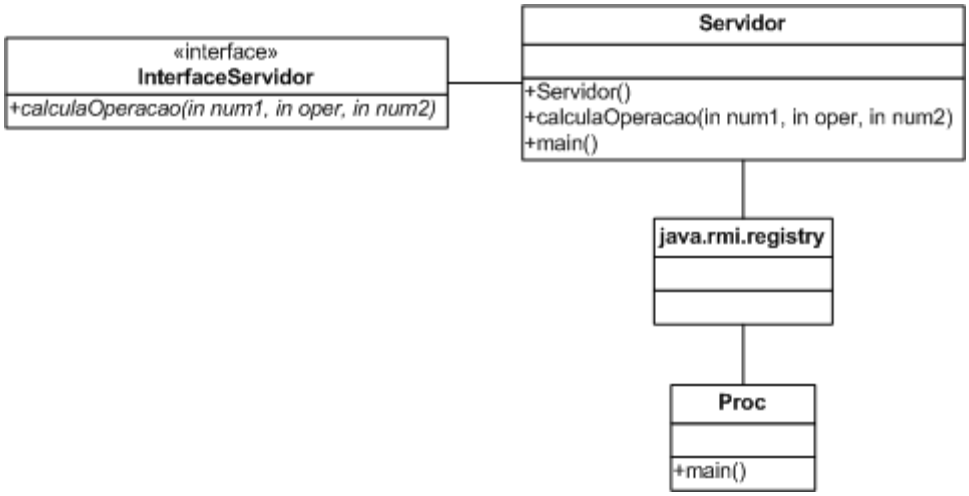


Figura 11: Diagrama de classes da calculadora em RMI

O método **calculaOperacao()** (figura 12) que é implementado pelo servidor, é o método que calcula a operação solicitada pelo cliente. Este método recebe como entrada três parâmetros, dois inteiros (num1 e num2) e um caracter (oper), devolvendo como retorno do método um valor inteiro que é o resultado da operação.

```
20    /** Interface implementada para método remoto */
21    public double calculaOperacao(int num1, char oper, int num2) throws RemoteException{
22
23        double resultado = 0.0;
24        switch(oper){
25            case '+':
26                resultado = num1 + num2;
27                break;
28            case '-':
29                resultado = num1 - num2;
30                break;
31            case '.':
32                resultado = num1 * num2;
33                break;
34            case '/':
35                resultado = num1 / num2;
36                break;
37        }
38        return resultado;
```

Figura 12: Método calculaOperacao()

Já no lado do cliente (classe **Proc**), não existem métodos além do método principal (**main()**) da classe.

3.2.1 Comportamento da execução da calculadora em RMI

Abaixo será apresentado o comportamento da execução da calculadora em RMI.

3.2.1.1 Servidor

O servidor (classe Servidor), assim como no caso da aplicação do chat, é também um processo que deve ficar sempre executando e disponibilizando seus métodos até que seja encerrado. Ele é uma

classe que implementa o método remoto definido na interface **InterfaceServidor** e funciona como um servidor de RMI. O seu comportamento durante a execução é o seguinte:

- Através da execução do método **main(),** este chama o método construtor da classe (linha 53 da figura 13), construindo as implementações do servidor e registrando os métodos remotos no servidor de nomes do RMI (linha 55 da figura 13).

```
43      /** Método principal do Servidor. Chama o método construtor da classe */
44      public static void main(String args[]) throws RemoteException{
45
46          try {
47              //Definir o security manager;
48              if(System.getSecurityManager() == null){
49                  System.setSecurityManager(new RMISecurityManager());
50              }
51
52              System.out.println("Construindo as implementacoes do Servidor...");
53              Servidor servidor = new Servidor();
54              System.out.println("Registrando as implementacoes...");
55              Naming.rebind("servidor", servidor);
56              System.out.println("Servidor iniciado!");
57
58              //System.exit(0);
59
60          }
61          catch( Exception e ) {
62              System.out.println( "Problema: " + e );
63          }
64
65      }
```

Figura 13: Método Main() do servidor

- Após a execução do método **main()**, o servidor que já disponibilizou os seus métodos remotos, fica rodando infinitamente até que alguém encerre o seu processo de execução.

3.2.1.1 Cliente

O processo cliente (classe **Proc**), ao contrário do servidor, é um processo que recebe os parâmetros a serem calculados diretamente na execução, os envia para o cliente e imprime na tela o resultado retornado pelo servidor. O funcionamento desse processo é o seguinte:

- Através da execução do método **main(),** o processo busca no servidor de nomes os serviços disponibilizados pelo servidor (linha 27 da figura 14). Após isso, ele cria o objeto de leitura do console (linha 31 da figura 14), lê os parâmetros de execução (num1, oper, num2), chama o método remoto **calculaOperacao()** (linha 43 da figura 14), armazenando o resultado da operação que é obtido através do retorno desse método em uma variável do tipo *double*. Após o retorno do resultado da operação, ele imprime o resultado e encerra a sua execução.

```
24              System.setSecurityManager(new RMISecurityManager());
25              //Cria referência para o servidor remoto.
26              System.out.println("Registrando-se no servidor...");
27              Remote referenciaRemota = Naming.lookup("servidor");
28              InterfaceServidor a = (InterfaceServidor) referenciaRemota;
29
30              //Cria objeto de leitura do console (teclado);
31              BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));
32
33              //Cria interface com console (teclado) e lê dados do console (teclado);
34              System.out.println("Sequencia esperada: Num1 (inteiro) + operador (+, -, ., /) + Num2 (inteiro)");
35              System.out.print("\nNum1 > " );
36              num1 = Integer.parseInt(teclado.readLine());
37              System.out.print("Oper > " );
38              oper = teclado.readLine().charAt(0);
39              System.out.print("Num2 > " );
40              num2 = Integer.parseInt(teclado.readLine());
41
42              //Envia dados lidos do console (teclado) para o servidor através do canal de saída e recebe o resultado do servidor;
43              resultado = a.calculaOperacao(num1, oper, num2);
44              System.out.println("----------");
45              System.out.println("resultado > " + resultado);
46
```

Figura 14: Método Main() do cliente

4. Conclusões

Neste artigo foram apresentados dois exemplos de aplicações usando Java RMI. Ambas as implementações em RMI (calculador e chat) foram criadas também usando-se sockets, visando comparar as funcionalidades em ambas as implementações.

Basicamente foram feitas as seguintes comparações: número de linhas de código, abstração da camada de comunicação, facilidade de uso e funcionalidades apresentadas. Em relação ao número de linhas de código implementado, tanto RMI quanto sockets apresentam números semelhantes, no entanto RMI ainda leva vantagem. Quanto à capacidade de abstração da camada de comunicação, em sockets é necessário criar explicitamente todas essas interfaces de comunicação, além, é claro, da necessidade de criar código para manipular essa comunicação. Por outro lado, o RMI abstrai toda essa comunicação, exigindo apenas que seja identificado o serviço remoto (*Naming.lookup()*), o qual permite, mesmo que de forma abstrata, o estabelecimento de uma conexão com outros processos.

Quanto a facilidade de uso e funcionalidades, ambas apresentam um bom conjunto de características, embora RMI seja um pouco limitado nas questões de gerenciamento e controle dos processos, ou seja, não existe uma camada que controle o acesso remoto dos processos.

Referências

[1] I. L. M. Ricarte, “Programação Orientada a Objetos: uma abordagem com Java”, Universidade Estatudal de Campinas, 2001, Capturado em www.dca.fee.unicmap.br/cursos/PooJava/Aulas/index.html, abril de 2007.

[2] E. R. Harold, "Java Networking Programming", Sebastopol: O' Relly & Associates, Inc, 1997, 1º edição, pg. 347-375.

[3] D. Destro, "Introdução ao RMI", Grupo de Usuários Java (GUJ), Capturado em: http://www.guj.com.br/java.tutorial.artigo.37.1.guj, abril 2007.