Remote Method Invocation: RMI na prática

Desenvolvendo aplicações com objetos distribuídos

Em aplicações distribuídas, é comum organizar os processos em processo servidor e processo cliente, de forma que os processos possam estar localizados em plataformas diferentes, em computadores diferentes. Segundo esse modelo, o processo servidor oferece serviços aos processos clientes, porém a implementação do modelo de comunicação entre tais processos pode ser difícil devido à complexidade natural da utilização de *sockets* ou outras *streams* de comunicação de dados. A complexidade aumenta quando é necessário acessar ou enviar objetos pela internet, pois os mesmos apresentam atributos e métodos que precisam ser convertidos para a forma binária para serem enviados por uma *stream*.

A fim de facilitar a implementação de aplicações distribuídas entre diversas plataformas de sistemas operacionais ou hardware, foi criado o padrão CORBA (*Common Object Request Architecture*), disponibilizado nas linguagens mais adotadas. Sua implementação oferece recursos para serem utilizados para facilitar a comunicação entre objetos de aplicações diferentes, dessa forma, permitindo invocar métodos de objetos remotos, mover objetos entre computadores diferentes e outros.

CORBA é um padrão multilinguagem, mas existe outro padrão criado especificamente para a linguagem Java, chamado de Java RMI (*Java Remote Method Invocation*), onde são oferecidos recursos avançados para o desenvolvimento de aplicações com objetos distribuídos, de forma a disponibilizar classes e interfaces simples de usar, mas poderosas; deste modo, possibilitando que a utilização de tais recursos seja totalmente contida na linguagem Java, sem precisar de bibliotecas de terceiros e sem sobrecarregar os programas com sintaxe adicional, e facilitando em muito o desenvolvimento de aplicações com objetos distribuídos.

Geralmente em aplicações Java RMI, o servidor é usado para criar objetos remotos cujos métodos são invocados pelos clientes da mesma forma como se fossem objetos locais, simplificando bastante as aplicações que realizam comunicação entre objetos distribuídos. Dessa forma, o Java RMI esconde a complexidade inerente à transmissão de dados por *streams* pela internet.

Fique por dentro

Este artigo apresenta os principais recursos disponibilizados pelo Java RMI para realizar a invocação de métodos de objetos remotos, fazendo com que a comunicação entre aplicações Java localizadas em diferentes JVMs seja de simples implementação, mas muito eficiente, principalmente por que Java RMI cria uma abstração na qual o objeto remoto é acessado como se estivesse localizado na JVM local.

Através de sua forma natural e completamente integrada com a linguagem Java, o Java RMI se mostra como sendo um poderoso padrão que é utilizado no desenvolvimento de aplicações com objetos distribuídos, oferecendo uma abstração simples de comunicação entre processos, escondendo a complexidade do uso de sockets e outros mecanismos de baixo nível usados em aplicações em que objetos precisam receber ou enviar informações a outros objetos.

A fim de que os objetos remotos sejam encontrados, um registro de objetos remotos é utilizado, e ele possibilita vincular objetos a nomes no lado do servidor, operação chamada de *bind*. No lado do cliente, o registro é consultado para que se obtenham referências a objetos remotos.

O pacote java.rmi

O pacote **java.rmi** contém a implementação do Java RMI pertencente ao JDK e disponibiliza classes, interfaces e subpacotes para serem utilizados no desenvolvimento de aplicações com objetos distribuídos. Dentre estes recursos, os principais são:

- **Remote:** É uma interface que indica se um objeto possui métodos que podem ser invocados remotamente por outras JVMs;
- **MarshalledObject:** É uma classe, e o seu construtor recebe um objeto e o converte para um vetor de bytes (*marshalling*). Suporta também a reconstrução do objeto (*unmarshalling*);
- **Naming:** É uma classe que oferece métodos para armazenar e consultar referências a objetos remotos no registro de objetos remotos do Java RMI;
- **RMISecuretyManager:** É uma classe utilizada por aplicações para verificar os requisitos de segurança para acessar classes descarregadas pela internet para serem executadas na máquina local;

• **AccessException:** É uma exceção que pode ser lançada por vários métodos da classe **Naming**, por exemplo, quando um servidor tenta se registrar no registro de objetos remotos e ocorre um erro;
• **AlreadyBoundException:** É uma exceção que ocorre quando se tenta registrar um nome no registro de objetos remotos, porém esse nome já se encontra registrado;
• **ConnectException:** É uma exceção que é lançada quando não é possível realizar uma conexão entre objetos remotos;
• **ConnectIOException:** É uma exceção que é lançada quando ocorre uma **IOException** na tentativa de conexão entre dois objetos remotos;
• **MarshallException:** É uma exceção que é lançada quando ocorre erro na operação de *marshalling*;
• **NoSuchObjectException:** É uma exceção que é lançada quando a aplicação tenta acessar um objeto remoto que não existe mais;
• **RemoteException:** É a superclasse comum a um grande número de exceções que podem ocorrer durante a invocação de um método remoto;
• **ServerException:** É uma exceção que é lançada no servidor se ocorrer erro no processamento de uma requisição de invocação de um método remoto por um cliente remoto;
• **UnmarshallException:** É uma exceção que é lançada quando ocorre erro na operação de *unmarshalling*;
• **java.rmi.activation:** É um pacote que oferece suporte à ativação de objetos remotos. Tais objetos são objetos que necessitam de acesso persistente e ininterrupto;
• **java.rmi.registry:** É um pacote que oferece suporte ao registro de objetos remotos do Java RMI. Suas principais classes são **LocateRegistry** e **Registry**;
• **java.rmi.server:** É um pacote que oferece classes e interfaces para a implementação do lado servidor do Java RMI. Suas principais classes são **UnicastRemoteObject**, **RemoteServer**, **RemoteObject**, entre outros.

A classe Naming

A classe **Naming** tem função primordial no controle do registro remoto RMI, sendo disponibilizada no pacote **java.rmi**. Ela é usada para armazenar e consultar referências a objetos remotos em um dado servidor. Seus principais métodos são:
• **static void bind(String name, Object obj):** Liga o objeto informado (**obj**) ao nome informado (**name**) no registro remoto RMI;
• **static String[] list(String name):** Retorna um vetor com os nomes encontrados no registro;
• **static Remote lookup(String name):** Retorna uma referência ao objeto remoto que está ligado ao nome informado como parâmetro;
• **static void rebind(String name, Object obj):** Liga o objeto informado (**obj**) ao nome informado (**name**) no registro remoto RMI, sobrescrevendo a ligação anterior que houver com o mesmo nome;
• **static void unbind(String name):** Elimina a ligação que existia do nome informado ao objeto que ele referenciava.

A classe LocateRegistry

É uma das principais classes presentes no pacote **java.rmi.registry**, pois ela é responsável por criar o registro RMI ou encontrá-lo em uma máquina virtual remota. Seus principais métodos são:
• **static Registry createRegistry(int port):** Cria um registro remoto RMI na máquina local usando a porta indicada;
• **static Registry getRegistry():** Retorna o registro remoto RMI presente na máquina local. Procura na porta padrão 1099;
• **static Registry getRegistry(int port):** Retorna o registro remoto RMI presente na máquina local na porta indicada;
• **static Registry getRegistry(String host):** Retorna o registro remoto RMI presente na máquina *host*. Procura na porta padrão 1099;
• **static Registry getRegistry(String host, int port):** Retorna o registro remoto RMI presente na máquina *host*. Procura na porta indicada.

A classe Registry

Situa-se no pacote **java.rmi.registry** e tem a função de manipular o registro remoto RMI, ligando ou desligando objetos remotos a nomes. Objetos da classe **Registry** podem ser obtidos quando realizadas consultas ao registro pela classe **LocateRegistry**. Os principais métodos de **Registry** são:
• **void bind(String nome, Remote object):** Liga o nome indicado (**name**) ao objeto remoto informado (**object**);
• **String[] list():** Retorna um vetor com todos os nomes presentes nesse registro;
• **Remote lookup(String name):** Consulta pelo objeto que esteja ligado ao nome indicado no registro;
• **void rebind(String nome, Remote object):** Sobrescreve a ligação do nome indicado, vinculando-o ao objeto remoto informado;
• **void unbind(String nome):** Elimina a ligação do nome indicado ao objeto que ele estava ligado no registro.

A interface Remote

A interface **Remote**, disponibilizada no pacote **java.rmi**, é usada para indicar objetos que contém métodos que são acessíveis remotamente por JVMs em clientes remotos. Para definir os métodos remotos é necessário que seja criada uma interface que contém a assinatura dos mesmos. Tal interface deve ser filha de **Remote**, garantindo assim, que o Java RMI considere essa interface como sendo uma interface de acesso remoto aos métodos do objeto remoto.

Portanto, para se criar um objeto no servidor cujos métodos sejam acessíveis remotamente, a classe desse objeto deve implementar uma interface filha de **Remote**, sendo que cada método dessa interface deve lançar a exceção **RemoteException**, pois podem ocorrer erros de comunicação no acesso remoto a tais métodos.

A interface **Remote** por si só não tem métodos, mas deve ser estendida diretamente ou indiretamente, caso contrário o objeto não poderá ser acessado remotamente. Adicionalmente, se um método de um objeto remoto retornar um objeto, este objeto deve estender a interface **java.io.Serializable**, pois de outra forma ocorrerá um erro na invocação deste método.

A classe UnicastRemoteObject

Para criar uma classe que seja acessível remotamente, pode-se fazer com que ela estenda **UnicastRemoteObject** (presente no pacote **java.rmi.server)**, pois assim a classe remota ganha por herança as funcionalidades necessárias.

Adicionalmente, a classe remota deve oferecer uma interface de acesso aos clientes remotos, pela qual são determinadas as assinaturas dos métodos acessíveis remotamente. Com esse objetivo, a classe remota deve implementar uma interface filha de **Remote**, como comentado anteriormente; dessa forma, definindo precisamente as assinaturas dos métodos remotos que são utilizadas tanto no lado do cliente como no lado do servidor.

A seguir é apresentada uma aplicação exemplo que demonstra a utilização dos principais recursos do pacote **java.rmi**.

Aplicação exemplo – Lado do servidor

Para abordar o RMI na prática, o funcionamento das principais classes do Java RMI será apresentado no desenvolvimento de duas aplicações, uma com o papel de cliente e outra com o papel de servidor. A aplicação servidora tem a função de simular o controle dos eletrodomésticos de uma casa inteligente, disponibilizando funções de ligar ou desligar TV, som, ar condicionado e ainda de abrir ou fechar cortinas.

A aplicação cliente, por sua vez, tem a função de usar Java RMI para acessar tais recursos do servidor, possibilitando ao usuário da casa utilizar as funções de automação remotamente em qualquer plataforma que tenha uma implementação da JVM, através de acesso à internet ou estando na mesma rede local.

Primeiro, vamos à definição das funções da casa remota que são disponibilizadas no servidor. Isso é feito através da codificação da interface **InterfaceRemota**, apresentada na **Listagem 1**.

Listagem 1. Declaração da interface InterfaceRemota.

```
01. package pkg.rmi;
02.
03. import java.rmi.Remote;
04. import java.rmi.RemoteException;
05.
06. public interface InterfaceRemota extends Remote {
07.     public void ligarTV (int id) throws RemoteException;
08.     public void desligarTV (int id) throws RemoteException;
09.     public void ligarSom (int id) throws RemoteException;
10.     public void desligarSom (int id) throws RemoteException;
11.     public void abrirCortina (int id) throws RemoteException;
12.     public void fecharCortina (int id) throws RemoteException;
13.     public void ligarAr (int id) throws RemoteException;
14.     public void desligarAr (int id) throws RemoteException;
15.     public int getId() throws RemoteException;
16.     public void desconectarCliente(int idCliente) throws RemoteException;
17.     public ObjetoStatus getStatus() throws RemoteException;
18. }
```

Dessa forma, são definidos todos os métodos que são acessíveis remotamente, uma vez que a interface **InterfaceRemota** estende **Remote** e os seus métodos lançam **RemoteException**. Observando esse código, pode-se verificar que existem métodos para ligar ou desligar TV, som e ar condicionado, além de abrir ou fechar cortinas.

Ainda analisando **InterfaceRemota**, o método **getID()** é responsável por retornar um identificador único ao cliente que recém se conectou no servidor (linha 15) e o método **desconectarCliente()** é responsável por sinalizar ao servidor que o cliente está sendo fechado (linha 16). Por fim, o método **getStatus()** é responsável por retornar o estado de cada eletrodoméstico da casa (incluindo as cortinas). Note que o estado é dado pela classe **ObjetoStatus**, apresentada na **Listagem 2**.

A classe **ObjetoStatus** tem a função de manter um registro do status dos eletrodomésticos da casa. Para isso, ela usa as variáveis **tvLigada**, **somLigado**, **cortinaAberta** e **arLigado** (linhas 7 a 10). Os *getters* e *setters* de tais atributos são definidos nas linhas 12 a 42. O método **toString()** próprio é definido de forma a retornar uma representação na forma de *string* desse objeto (linhas 44 a 68). Note que na criação da representação textual é utilizada a classe **StringBuilder** de forma a otimizar o desempenho e evitar a fragmentação da memória, causada nos casos em que a concatenação de strings é feita com o operador **+**.

Antes de apresentar a implementação dos métodos remotos, no entanto, é necessário apresentar a GUI do servidor, que tem a função de manter um log das operações recebidas, registrando a conexão e desconexão de clientes e o uso das funções de automação da casa inteligente, como apresentado na **Listagem 3**.

A GUI servidora é apresentada na **Figura 1** e mostra que um cliente se conectou, realizou a operação de ligar a TV e em seguida se desconectou.

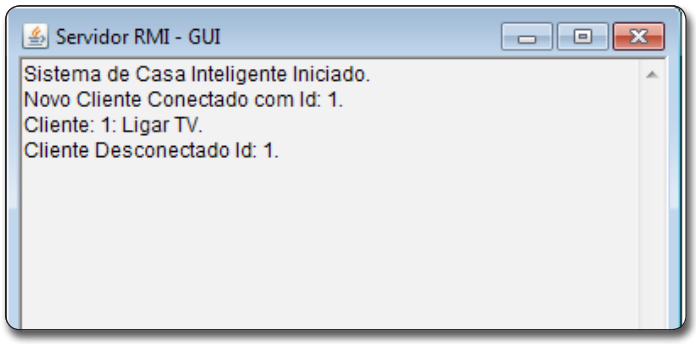

Figura 1. GUI no lado do servidor

Observe que a GUI servidora é uma interface gráfica que contém apenas um componente de texto, usado para o registro de mensagens. O método **imprimirMensagem()** recebe uma **String** e a escreve na tela (linhas 24 a 27). Neste método é importante notar o uso da palavra-chave **synchronized** (linha 25), de forma a definir um bloco sincronizado que é usado para evitar problemas de acesso concorrente ao componente de texto.

Agora, é apresentado o código do servidor na **Listagem 4**, onde a classe **ServidorRmi** (linha 8) estende **UnicastRemoteObject**, se tornando um objeto acessível remotamente, e implementa **InterfaceRemota**, assim, definindo os métodos que são remotos, e também suas implementações.

Listagem 2. Declaração da classe ObjetoStatus.

```
01. package pkg.rmi;
02.
03. import java.io.Serializable;
04.
05. public class ObjetoStatus implements Serializable {
06.
07.  private boolean tvLigada;
08.  private boolean somLigado;
09.  private boolean cortinaAberta;
10.  private boolean arLigado;
11.
12.  public boolean isTvLigada() {
13.   return tvLigada;
14.  }
15.
16.  public void setTvLigada(boolean tvLigada) {
17.   this.tvLigada = tvLigada;
18.  }
19.
20.  public boolean isSomLigado() {
21.   return somLigado;
22.  }
23.
24.  public void setSomLigado(boolean somLigado) {
25.   this.somLigado = somLigado;
26.  }
27.
28.  public boolean isCortinaAberta() {
29.   return cortinaAberta;
30.  }
31.
32.  public void setCortinaAberta(boolean cortinaAberta) {
33.   this.cortinaAberta = cortinaAberta;
34.  }
35.
36.  public boolean isArLigado() {
37.   return arLigado;
38.  }
39.
40.  public void setArLigado(boolean arLigado) {
41.   this.arLigado = arLigado;
42.  }
43.
44.  public String toString () {
45.   StringBuilder builder = new StringBuilder(200);
46.   builder.append("Status: TV[");
47.   if (tvLigada)
48.    builder.append("Ligada");
49.   else
50.    builder.append("Desligada");
51.   builder.append("] Som[");
52.   if (somLigado)
53.    builder.append("Ligado");
54.   else
55.    builder.append("Desligado");
56.   builder.append("] Cortina[");
57.   if (cortinaAberta)
58.    builder.append("Aberta");
59.   else
60.    builder.append("Fechada");
61.   builder.append("] Ar[");
62.   if (arLigado)
63.    builder.append("Ligado");
64.   else
65.    builder.append("Desligado");
66.   builder.append("]");
67.   return builder.toString();
68.  }
69. }
```

Listagem 3. Declaração da classe ServidorGui.

```
01. package pkg.rmi;
02.
03. import java.awt.Frame;
04. import java.awt.TextArea;
05. import java.awt.event.WindowAdapter;
06. import java.awt.event.WindowEvent;
07.
08. public class ServidorGui extends Frame {
09.
10.  private TextArea textArea = new TextArea();
11.
12.  public ServidorGui () {
13.   super("Servidor RMI - GUI");
14.   setSize(400, 500);
15.   add(textArea);
16.   textArea.setEditable(false);
17.   addWindowListener( new WindowAdapter() {
18.    public void windowClosing(WindowEvent we) {
19.     System.exit(0);
20.    }
21.   } );
22.  }
23.
24.  public void imprimirMensagem (String mensagem) {
25.   synchronized (this) {
26.    textArea.append(mensagem + "\r\n");
27.   }
28.  }
29. }
```

A GUI servidora é dada separadamente pela classe **ServidorGui**, e o servidor contém uma referência para acessá-la instanciada na linha 10. Também é mantido um objeto com os status dos eletrodomésticos (linha 11). A variável estática **id** (linha 12) registra o identificador do último cliente criado, e a cada vez que um cliente se conecta, o **id** é incrementado, como ocorre no método **getId()** (linhas 90 a 96), onde, através de um bloco sincronizado é gerado o novo **id**, que é escrito na tela e retornado para o cliente.

O método **main()** (linha 18) é o código executável que inicia a aplicação servidor. Primeiramente, ele cria um registro remoto RMI na porta 1099 (linha 21), em seguida, instancia o objeto remoto (linha 27) e o liga ao nome **RmiServer** (linha 28). Já nas linhas 30 e 31, é criada uma mensagem na GUI servidora informando que a aplicação foi iniciada e a mesma é exibida com o comando **setVisible()**.

O método **ligarTV()** é implementado nas linhas 34 a 39, originário da interface **InterfaceRemota**. Ele marca a TV como ligada (no objeto **status**, linha 36) e imprime uma mensagem na tela mostrando o **id** do cliente e a operação que foi realizada (linha 37). O mesmo padrão é repetido para os demais eletrodomésticos para ligar ou desligar até a linha 88.

No final da classe encontra-se o método **desconectarCliente()**. Este imprime na tela uma mensagem informando o **id** do cliente que foi desconectado (linhas 98 na 100). Por último, o método

getStatus() tem a função de retornar para o cliente o status dos eletrodomésticos (linhas 102 a 104).

Com isso, foram apresentadas todas as classes localizadas no lado do servidor. A seguir serão apresentadas as classes que ficam localizadas no lado do cliente, que invoca os métodos remotos do objeto registrado com o nome **RmiServer**.

Aplicação exemplo – Lado do cliente

Apresentamos a aplicação servidora, porém ela não tem utilidade sem a existência da aplicação cliente, com a qual se comunica. Neste exemplo a aplicação cliente é representada pela classe **ClienteRmi**, que contém um método **main()** com a função de obter uma referência ao objeto remoto **RmiServer** (linha 9) fazendo uma consulta ao registro remoto RMI no *host localhost* (máquina local) na porta padrão 1099. Observe que o servidor poderia estar localizado em outro computador. Portanto, para acessá-lo, basta informar o nome ou o endereço IP do computador remoto no local da palavra *localhost*.

Na **Listagem 5**, a GUI cliente é instanciada na linha 10 e como ela é um *thread* **Runnable** (como visto na **Listagem 6**), inicia

Listagem 4. Declaração da classe ServidorRmi.

```
01. package pkg.rmi;
02.
03. import java.rmi.Naming;
04. import java.rmi.RemoteException;
05. import java.rmi.server.UnicastRemoteObject;
06. import java.rmi.registry.*;
07.
08. public class ServidorRmi extends UnicastRemoteObject implements
    InterfaceRemota {
09.
10.  private static ServidorGui gui = new ServidorGui();
11.  private ObjetoStatus status = new ObjetoStatus();
12.  private static int id = 0;
13.
14.  public ServidorRmi() throws RemoteException {
15.   super(0);
16.  }
17.
18.  public static void main(String args[]) throws Exception {
19.   System.out.println("Servidor RMI iniciado.");
20.   try {
21.    LocateRegistry.createRegistry(1099);
22.    System.out.println("Registro Java RMI criado.");
23.   } catch (RemoteException e) {
24.    System.out.println("Registro Java RMI já existente.");
25.   }
26.
27.   ServidorRmi servidor = new ServidorRmi();
28.   Naming.rebind("//localhost/RmiServer", servidor);
29.   System.out.println("RmiServer Registrado.");
30.   gui.imprimirMensagem("Sistema de Casa Inteligente Iniciado.");
31.   gui.setVisible(true);
32.  }
33.
34.  public void ligarTV(int idCliente) throws RemoteException {
35.   synchronized (this) {
36.    status.setTvLigada(true);
37.    gui.imprimirMensagem("Cliente: " + idCliente + ": Ligar TV.");
38.   }
39.  }
40.
41.  public void desligarTV(int idCliente) throws RemoteException {
42.   synchronized (this) {
43.    status.setTvLigada(false);
44.    gui.imprimirMensagem("Cliente: " + idCliente + ": Desligar TV.");
45.   }
46.  }
47.
48.  public void ligarSom(int idCliente) throws RemoteException {
49.   synchronized (this) {
50.    status.setSomLigado(true);
51.    gui.imprimirMensagem("Cliente: " + idCliente + ": Ligar Som.");
52.   }
53.  }
54.
55.  public void desligarSom(int idCliente) throws RemoteException {
56.   synchronized (this) {
57.    status.setSomLigado(false);
58.    gui.imprimirMensagem("Cliente: " + idCliente + ": Desligar Som.");
59.   }
60.  }
61.
62.  public void abrirCortina(int idCliente) throws RemoteException {
63.   synchronized (this) {
64.    status.setCortinaAberta(true);
65.    gui.imprimirMensagem("Cliente: " + idCliente + ": Abrir Cortina.");
66.   }
67.  }
68.
69.  public void fecharCortina(int idCliente) throws RemoteException {
70.   synchronized (this) {
71.    status.setCortinaAberta(false);
72.    gui.imprimirMensagem("Cliente: " + idCliente + ": Fechar Cortina.");
73.   }
74.  }
75.
76.  public void ligarAr(int idCliente) throws RemoteException {
77.   synchronized (this) {
78.    status.setArLigado(true);
79.    gui.imprimirMensagem("Cliente: " + idCliente + ": Ligar Ar.");
80.   }
81.  }
82.
83.  public void desligarAr(int idCliente) throws RemoteException {
84.   synchronized (this) {
85.    status.setArLigado(false);
86.    gui.imprimirMensagem("Cliente: " + idCliente + ": Desligar Ar.");
87.   }
88.  }
89.
90.  public int getId(){
91.   synchronized (this) {
92.    id++;
93.    gui.imprimirMensagem("Novo Cliente Conectado com Id: " + id + ".");
94.    return id;
95.   }
96.  }
97.
98.  public void desconectarCliente(int idCliente) throws RemoteException {
99.   gui.imprimirMensagem("Cliente Desconectado Id: " + idCliente + ".");
100. }
101.
102. public ObjetoStatus getStatus() throws RemoteException {
103.  return status;
104. }
105.}
```

seu processamento concorrente na chamada do método **start()** (linha 11), e passa a ser visível na linha 12 para iteração com o usuário.

Listagem 5. Declaração da classe ClienteRmi.

```
01. package pkg.rmi;
02.
03. import java.rmi.Naming;
04.
05. public class ClienteRmi {
06.
07.  public static void main(String args[]) throws Exception {
08.
09.   InterfaceRemota servidor = (InterfaceRemota)
      Naming.lookup("//localhost/RmiServer");
10.   ClienteGui gui = new ClienteGui(servidor);
11.   new Thread(gui).start();
12.   gui.setVisible(true);
13.  }
14. }
```

A última classe pertencente à aplicação é a GUI do cliente, apresentada na **Listagem 6**. Esta GUI tem a função de mostrar na sua parte superior o status dos eletrodomésticos e na sua parte inferior, os botões, para realizar as operações de ativar ou desativar os mesmos. A fim de simplificar a sua implementação, a classe **ClienteGui** foi definida como sendo um frame AWT, pois estende **Frame**, e além disso, é um *thread*, pois implementa **Runnable**. Dessa forma, ela pode ficar obtendo o status atual dos eletrodomésticos localizados no servidor para informar ao usuário (cliente), pois tal informação é exibida na GUI, como será mostrado mais adiante.

A interface gráfica provida pela classe **ClienteGui** é apresentada na **Figura 2**, sendo separada em duas partes: status e operações. Na área de status, localizada na parte superior, nota-se que a TV já se encontra ligada, pois o usuário solicitou anteriormente a operação de ligar a TV, clicando no botão de ação correspondente na parte inferior, que contém as operações disponíveis.

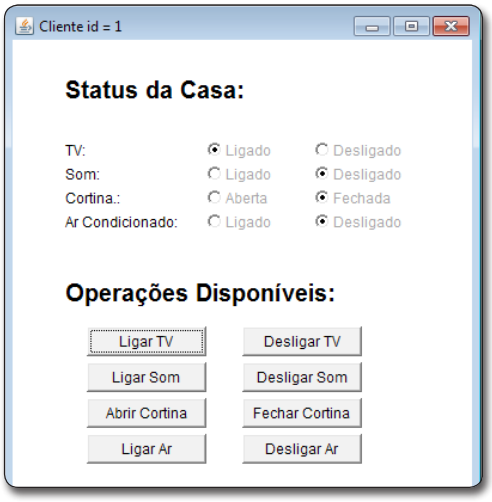

Figura 2. GUI no lado do cliente

A classe **ClienteGui** apresenta diversos atributos de instância, onde o primeiro, chamado de **idCliente** (declarado na linha 17 da **Listagem 6**), tem a função de manter a identificação desse cliente junto ao servidor. Este atributo é inicializado somente mais tarde (linha 61 da **Listagem 6**), quando o servidor for acessado, pois tal valor é definido somente uma vez, já que se trata de um atributo **final**, como requerido mais adiante por ser acessado dentro de classes anônimas internas (dadas pelos *listeners*, como exemplo, o *listener* para ligar a TV, na linha 89 da **Listagem 7**). Pelos mesmos motivos, a variável **servidor** também é declarada como **final** (linha 18 da **Listagem 6**). Por fim, nas linhas 19 e 20 da **Listagem 6** são declarados os *labels* para serem exibidos na GUI.

A classe **ClienteGui** apresenta muitos outros atributos referentes a cada eletrodoméstico, sendo nas linhas 22 a 27 declarados os componentes da interface gráfica referentes à TV: um *label*, um botão de rádio para mostrar o status do aparelho e dois botões (um para ligar a TV e outro para desligar). Analogamente, cada outro eletrodoméstico contém componentes equivalentes que são instanciados até a linha 48.

Depois da declaração de todos os campos da interface gráfica, o construtor da GUI cliente é declarado, recebendo como parâmetro um objeto remoto servidor (linha 50), que é mantido como variável de instância para que possa ser acessível aos *listeners* (linha 52). Em seguida, o método **getId()** do servidor é consultado para obter o **id** desse cliente (linhas 53 a 61). Logo após, são adicionados os *labels* que organizam a interface gráfica em status e operações (linhas 66 e 67), de modo que suas posições são definidas nas linhas 68 e 69. Observe nas linhas 70 e 71 o uso do método **setFont()** da classe **Label**. Este modifica a fonte de exibição para exibir as palavras com tamanho maior.

Feito isso, são inseridos e posicionados os componentes gráficos que representam cada eletrodoméstico. A **Listagem 7** apresenta o restante do construtor, que inicia pelos componentes da TV e define de forma análoga os outros eletrodomésticos.

Observe que nas linhas 73 a 77 os campos da GUI referentes à TV são adicionados ao *frame* para exibição. Os botões de rádio correspondentes ao status da TV são desabilitados para edição pelo usuário nas linhas 78 e 79, pois eles têm apenas a finalidade de exibição de status (sendo atribuído aos botões de ação a realização de operações sobre os eletrodomésticos). Depois disso, nas linhas 80 a 84, os componentes da interface gráfica da TV são ajustados na tela com o comando **setBounds()**, definindo a posição e o tamanho de cada um.

O *listener* do botão de ligar TV é definido nas linhas 85 a 95, sendo uma classe anônima interna, pois a declaração do seu código-fonte é contida dentro da declaração de outra classe (**ClinteGui**), sem definição explícita de nome. No processamento do evento do *listener* é usado um bloco sincronizado (linha 88) a fim de evitar problemas de concorrência no acesso ao objeto servidor e aos campos da GUI. Na linha 89, o servidor é chamado para ligar a TV, invocando o método remoto **ligarTV()**, e nas duas próximas linhas, a mudança de status é aplicada à interface gráfica através do método **setState()**

do botão de rádio. De forma semelhante, nas linhas 96 a 106 é criado o *listener* para o evento de clique no botão de ação para desligar a TV, que invoca outro método remoto, o **desligarTV()**.

Até agora foi apresentada a criação dos componentes da interface gráfica para a TV. Adicionalmente, os componentes gráficos dos demais eletrodomésticos da casa inteligente são declarados até a linha 212 dessa listagem, porém as linhas de código estão omitidas para evitar a repetição de instruções semelhantes. O código fonte completo está disponível para download.

Listagem 6. Primeira parte da declaração da classe ClienteGui.

```
01. package pkg.rmi;
02.
03. import java.awt.Button;
04. // alguns imports omitidos...
13. import java.rmi.RemoteException;
14.
15. public class ClienteGui extends Frame implements Runnable {
16.
17.  private final int idCliente;
18.  private final InterfaceRemota servidor;
19.  private Label status = new Label("Status da Casa:");
20.  private Label operacoes = new Label("Operações Disponíveis:");
21.
22.  private Label labelTV = new Label("TV:");
23.  private CheckboxGroup tv = new CheckboxGroup();
24.  private Checkbox tv1 = new Checkbox("Ligado", tv, true);
25.  private Checkbox tv2 = new Checkbox("Desligado", tv, true);
26.  private Button ligarTV   = new Button ("Ligar TV");
27.  private Button desligarTV = new Button ("Desligar TV");
28.
29.  private Label labelSom = new Label("Som:");
30.  private CheckboxGroup som = new CheckboxGroup();
31.  private Checkbox som1 = new Checkbox("Ligado", som, true);
32.  private Checkbox som2 = new Checkbox("Desligado", som, true);
33.  private Button ligarSom   = new Button ("Ligar Som");
34.  private Button desligarSom = new Button ("Desligar Som");
35.
36.  private Label labelCortina = new Label("Cortina.:");
37.  private CheckboxGroup cortina = new CheckboxGroup();
38.  private Checkbox cortina1 = new Checkbox("Aberta", cortina, true);
39.  private Checkbox cortina2 = new Checkbox("Fechada", cortina, true);
40.  private Button ligarCortina   = new Button ("Abrir Cortina");
41.  private Button desligarCortina = new Button ("Fechar Cortina");
42.
43.  private Label labelAr = new Label("Ar Condicionado:");
44.  private CheckboxGroup ar = new CheckboxGroup();
45.  private Checkbox ar1 = new Checkbox("Ligado", ar, true);
46.  private Checkbox ar2 = new Checkbox("Desligado", ar, true);
47.  private Button ligarAr   = new Button ("Ligar Ar");
48.  private Button desligarAr = new Button ("Desligar Ar");
49.
50.  public ClienteGui (final InterfaceRemota servidor) {
51.
52.   this.servidor = servidor;
53.   int id = 0;
54.   try {
55.    id = servidor.getId();
56.   } catch (RemoteException e) {
57.    e.printStackTrace();
58.    System.out.println("Erro de Conexão");
59.    System.exit(0);
60.   }
61.   idCliente = id;
62.   setTitle("Cliente id = " + idCliente);
63.
64.   setLayout(null);
65.   setSize(400, 440);
66.   add(status);
67.   add(operacoes);
68.   status.setBounds(50, 60, 300, 25);
69.   operacoes.setBounds(50, 230, 300, 25);
70.   status.setFont(new Font("Times", 1, 20));
71.   operacoes.setFont(new Font("Times", 1, 20));
72.
```

Listagem 7. Segunda parte da declaração da classe ClienteGui.

```
73.    add(labelTV);
74.    add(tv1);
75.    add(tv2);
76.    add(ligarTV);
77.    add(desligarTV);
78.    tv1.setEnabled(false);
79.    tv2.setEnabled(false);
80.    labelTV  .setBounds( 50, 110, 100, 25);
81.    tv1      .setBounds(170, 110, 80, 25);
82.    tv2      .setBounds(260, 110, 100, 25);
83.    ligarTV  .setBounds(70, 270, 100, 25);
84.    desligarTV.setBounds(200, 270, 100, 25);
85.    ligarTV.addActionListener(new ActionListener() {
86.     public void actionPerformed(ActionEvent evt) {
87.      try {
88.       synchronized (this) {
89.        servidor.ligarTV(idCliente);
90.        tv1.setState(true);
91.        tv2.setState(false);
92.       }
93.      } catch (RemoteException e) {
94.       e.printStackTrace();
95.    }}});
96.    desligarTV.addActionListener(new ActionListener() {
97.     public void actionPerformed(ActionEvent evt) {
98.      try {
99.       synchronized (this) {
100.       servidor.desligarTV(idCliente);
101.       tv1.setState(false);
102.       tv2.setState(true);
103.      }
104.     } catch (RemoteException e) {
105.      e.printStackTrace();
106.   }}});
107.
      ...
214.   addWindowListener( new WindowAdapter() {
215.   public void windowClosing(WindowEvent we) {
216.    try {
217.     servidor.desconectarCliente(idCliente);
218.    } catch (RemoteException e) {
219.     System.out.println("Erro ao desconectar cliente: " + e.getMessage());
220.    }
221.    System.exit(0);
222.   }
223.  });
224. }
```

Após todo esse código, é declarado o *listener* para fechamento da GUI (linhas 214 a 222), que avisa o servidor que o cliente está sendo fechado invocando o método remoto **desconectarCliente()**. Na **Listagem 8** é apresentada a última parte da classe **ClienteGui**, o código a ser executado na forma de *thread*.

Listagem 8. Última parte da declaração da classe ClienteGui.

```
225. public void run() {
226.   while (true) {
227.     try {
228.       ObjetoStatus status = servidor.getStatus();
229.       tv1.setState(status.isTvLigada());
230.       tv2.setState(!status.isTvLigada());
231.       som1.setState(status.isSomLigado());
232.       som2.setState(!status.isSomLigado());
233.       cortina1.setState(status.isCortinaAberta());
234.       cortina2.setState(!status.isCortinaAberta());
235.       ar1.setState(status.isArLigado());
236.       ar2.setState(!status.isArLigado());
237.     } catch (RemoteException e) {
238.       System.out.println("Erro ao acessar status:" + e.getMessage());
239.     }
240.     synchronized (this) {
241.       try {
242.         wait(1000);
243.       } catch (InterruptedException e) {
244.         e.printStackTrace();
245.       }
246.     }
247.   }
248. }
249.}
```

Veja que é declarado o método **run()** e como a classe **ClienteGui** implementa a interface **Runnable**, ela pode ser executada como *thread* (linha 11 da **Listagem 5**). Nesse caso, durante a execução de **run()**, o processamento fica preso em um laço infinito (linha 226), e a cada iteração o método remoto **getStatus()** é invocado para obter o status dos eletrodomésticos da casa.

Uma vez que o servidor tenha retornado o status dos eletrodomésticos, a interface gráfica é atualizada para refletir o status obtido (linhas 229 a 236), marcando cada botão de rádio de acordo com o status obtido de cada eletrodoméstico. Por fim, na linha 242, é chamado o método **wait()** para fazer o thread perder a CPU durante um segundo. Dessa forma, a cada iteração, ele busca o status, atualiza os botões de rádio e espera um segundo antes de iniciar a nova iteração.

Stubs e Skeletons

Em versões mais antigas do Java era necessário usar o compilador **rmic** para gerar as classes *stubs* e *skeletons*, as quais são classes implícitas que têm a função de realizar a comunicação entre servidor e cliente, de forma a implementar a interface **Remote** dos objetos remotos, adicionando as funcionalidades de comunicação. Por definição, o *stub* é localizado no lado do cliente e o *skeleton* é localizado no lado do servidor. Felizmente, após a versão 5 do Java, não é mais necessário usar o comando **rmic** para gerar os *stubs*, pois tal tarefa é feita automaticamente pela JDK no processo de compilação do código fonte.

Porém, implicitamente, ainda são utilizados *stubs e skeletons*, e quando um cliente faz uma invocação de um método remoto, internamente, o *stub* é chamado, iniciando uma sequência de passos onde, primeiramente, cria-se uma conexão com a JVM remota que contém o objeto chamado. Em seguida, os dados da requisição são transformados em um vetor de bytes (*marshalling*) e transmitidos à JVM remota e o *stub* aguarda até o processamento do método terminar, que é quando ele recebe o retorno e realiza a operação de *unmarshalling*, e finalmente devolve o retorno do método ao cliente.

A maioria das aplicações utilizadas na atualidade contém algum tipo de comunicação em rede ou pela internet, demandando recursos de comunicação entre processos. Para solucionar este requisito, existem diversos métodos de comunicação que podem ser utilizados, como por exemplo, o protocolo HTTP para acesso a *websites* ou *sockets* para transmissão física de dados entre processos de plataformas diferentes.

Comparado com outros modelos de comunicação entre processos de plataformas diferentes, o modelo de comunicação baseado em objetos distribuídos é mais refinado, pois as clássicas diretivas de *sockets* para enviar e receber mensagens são substituídas pela invocação de métodos de objetos remotos. Sendo assim, explore a fundo os recursos do RMI, para que quando encontre requisitos nos quais ele possa ser empregado você já tenha domínio sobre o assunto e saiba empregá-lo de forma diferenciada.

Autor

John Soldera
johnsoldera@gmail.com

É bacharel em Ciências da Computação pela UCS (Universidade de Caxias do Sul), mestre em Computação Aplicada pela Unisinos e cursa atualmente doutorado em Ciências da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul). Trabalha com Java há 12 anos e possui a certificação SCJP.

Links:

Javadoc da plataforma Java SE 7.
http://docs.oracle.com/javase/7/docs/api/

Site do projeto CORBA.
http://www.corba.org/

Documentação com informações sobre Java RMI.
http://docs.oracle.com/javase/7/docs/api/java/rmi/package-summary.html