Java aplicada a Redes de Computadores

Prof. Rogério Santos Pozza e Prof. Henrique Yoshikazu Shishido

Universidade Tecnológica Federal do Paraná
Campus Cornélio Procópio

Resumo Este documento apresenta dois artifícios oferecidos pela tecnologia Java: Sockets e RMI. São muito utilizadas para a criação de protocolos de comunicação e também para auxiliar na invocação de métodos remotos (RMI). Tal documento está organizado em dois capítulos: Java Socket e Java RMI.

1 A disciplina

Essa disciplina irá apresentar as API Socket e RMI do Java contidas no pacote java.net. Socket nada mais é do que um conceito implementado pelo sistema operacional para que haja conexões através de uma rede de computadores.

Nesta disciplina serão necessários todos os conhecimentos de bibliotecas aprendidas nas disciplinas anteriores, principalmente os conceitos de orientação a objetos e *threads*. Durante o transcorrer da disciplina, você perceberá que a utilização dessa API é simples. No entanto, as regras (protocolo) de transmissão de dados será definido pelo projetista do sistema.

Os protocolos TCP e UDP são protocolos que estão relacionados à Camada 4 do Protocolo TCP/IP ou Modelo OSI. Toda e qualquer comunicação em redes de computadores acabam utilizando uma desses dois protocolos para se comunicar. Porém, existem algumas peculiaridades a serem consideradas de acordo com cada protocolo.

2 Protocolo TCP

O TCP é um protocolo orientado à conexão. Assim, é preciso estabelecer uma conexão (*handshaking*), antes de qualquer transmissão de dados entre dois computadores. Dentro desse contexto, cada máquina cliente necessitará de estabelecer uma conexão ao servidor, abrindo um canal de comunicação entre os dois computadores (*hosts*) conforme apresentado na Figura 1.

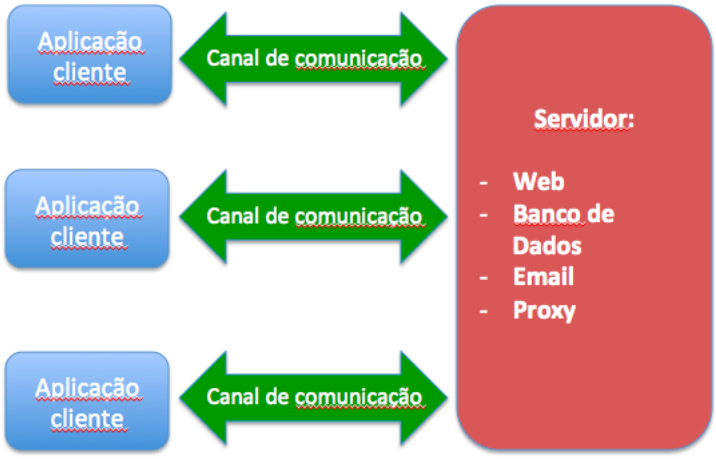

Figura 1: Comunicação TCP

Como pode se observar, pode-se conectar vários clientes a um único servidor. Desse modo, é um servidor possuir diversos serviços web, banco de dados, e-mail entre outros, e clientes se conectarem aos diversos serviços oferecidos por um servidor.

O desenvolvedor Java não precisa se preocupar com a implementação dos protocolos das camadas de rede. Para isso, o pacote `java.net` oferece classes que facilitam a comunicação entre dois computadores.

O TCP é um protocolo que possui algumas vantagens em relação ao protocolo UDP relacionadas à qualidade da comunicação, tais como:

- garantia de entrega dos pacotes ao destino;
- mantém a integridade dos dados enviados;
- realiza a entrega dos pacotes de maneira ordenada;

2.1 Portas de comunicação

Conforme mencionado anteriormente, um servidor pode oferecer diversos serviços. Desse modo, um cliente pode querer usufruir mais de um serviço. Como é possível a transmissão de dados de dois ou mais serviços diferentes a um cliente?

As aplicações realizam a transmissão de dados por meio de uma única conexão física, mas, para conseguir discernir os dados enviados/recebidos por aplicações distintas utiliza-se o conceito de **portas** de comunicação.

Da mesma forma que o endereço IP serve para identificar uma máquina, uma porta de comunicação permite identificar a comunicação de diferentes aplicações. Cada endereço IP de um computador possui 65536 portas (de 0 a 65535).

Ao configurar um serviço para ser executado na porta 80 (padrão do protocolo *http*), é possível um cliente se conectar a esse servidor por meio desta porta

de comunicação, juntamente com o endereço IP da máquina. Por exemplo, o servidor web da www.utfpr.edu.br pode ser representado por: www.utfpr.edu.br:80. Na maioria das vezes que você utilizou um servidor web, não deve ter adicionado a porta no final do endereço, mas mesmo assim você conseguiu abrir a página do site requisitado. Por quê isso?

Geralmente, os programas cliente de um determinado serviço como o navegador web, implicitamente adicionam a porta de comunicação da padrão do protocolo do serviço utilizado para facilitar a vida do usuário. Entretanto, se você abrir o seu navegador e digitar o endereço www.utfpr.edu.br ou www.utfpr.edu.br:80, notará que a página irá abrir normalmente. Existem casos em que um serviço roda em uma porta diferente da padrão, como é o caso de alguns servidores web que rodam na porta 8080, 8084, etc. Nesses casos, é obrigatório adicionar a porta de comunicação ao final do endereço.

2.2 Socket

É importante ficar claro que um serviço oferecido na porta 80 de um servidor, pode receber diversas conexões. Assim, surge a seguinte pergunta: se um cliente se conecta na porta 80 de um servidor, enquanto ele não se desconectar de tal porta, será possível que outro cliente se conecte?

A resposta é SIM. Ao aceitar uma conexão na porta 80, o servidor redireciona o cliente para uma outra porta disponível pelo sistema operacional liberando a porta 80 novamente para que receba uma nova conexão. A tecnologia Java permite aceitar múltiplas conexões através do uso de *threads*.

A implementação de Socket em Java sempre possui dois lados: o servidor e o cliente.

Servidor Para concretizar gradativamente o conhecimento de Sockets, vamos implementar um exemplo em que no programa cliente é digitado uma mensagem em caracteres minúsculos e, posteriormente, essa mensagem é enviada ao servidor que, por sua vez, converte todos os caracteres para maiúsculo e as exibe em sua tela (do servidor).

Para isso, a primeira codificação a ser realizada é colocar o servidor para abrir uma porta e ficar ouvindo até alguma aplicação cliente se conectar. Observe na linha 5 da Listagem 1.1 que é instanciado um objeto ServerSocket. Ao construtor desse objeto é passado a porta de comunicação que será utilizada para receber as conexões de clientes.

Listagem 1.1: Implementação básica de um servidor de socket

```
1 import java.net.*;
2 import java.io.*;
3
4 public class Servidor {
5   public static void main(String args[]) throws IOException{
6     ServerSocket servidor = new ServerSocket(54321);
```

```
7     System.out.println("A porta 54321 foi aberta.");
8     //Codificao continua aqui...
9   }
10 }
```

Caso o objeto tenha sido instanciado, isto indica que a porta 54321 estava disponível e fechada e foi aberta. Se qualquer outro programa estivesse fazendo uso dessa porta no mesmo no momento da instanciação, obviamente o exemplo não funcionaria, pois apenas uma única aplicação (processo no sistema operacional) pode fazer uso de uma porta.

Após a abertura da porta de comunicação, é preciso esperar por uma requisição de conexão de um cliente através da invocação do método `accept()` (linha 8) do objeto `ServerSocket`. Esse método é considerado bloqueante, pois a execução do programa só continuará, após um cliente se conectar.

Listagem 1.2: Método accept() para aceitar uma conexão de cliente

```
1 import java.net.*;
2 import java.io.*;
3
4 public class Servidor {
5   public static void main(String args[]) throws IOException {
6     ServerSocket servidor = new ServerSocket(54321);
7     System.out.println("A porta 54321 foi aberta.");
8
9     Socket conexao = servidor.accept();
10  }
11 }
```

Após o objeto `socket` receber a conexão de um cliente, um canal de comunicação é criado. No entanto, esse canal é dividido em dois sub-canais de entrada e saída de dados, representados pelas classes `DataInputStream` `DataOutputStream`, respectivamente.

A classe `DataInputStream` permite criar um objeto para receber dados vindos do cliente, enquanto a `DataOutputStream` é considerado o canal de saída de dados, ou seja, é utilizado para enviar dados ao cliente. A instanciação de objetos dessas classes é realizado através de um método oferecido pelo objeto `socket` denominado `getInputStream()` e `getOutputStream()`. No caso deste exemplo, a classe `Servidor` irá somente receber dados do cliente, assim será instanciado apenas um objeto para recepção de dados `DataInputStream`conforme apresentado na linhas 10 da Listagem 1.3.

Listagem 1.3: Criando canais de comunicação

```
1 import java.net.*;
2 import java.io.*;
3
4 public class Servidor {
```

```
5   public static void main(String args[]) throws IOException{
6     ServerSocket servidor = new ServerSocket(54321);
7     System.out.println("A porta 54321 foi aberta.");
8
9     Socket conexao = servidor.accept();
10
11    DataInputStream entrada = new DataInputStream(socket.
          getInputStream());
12
13    String mensagem = entrada.readUTF();
14    mensagem = mensagem.toUpperCase();
15
16    System.out.println("A mensagem em maiusculo e: " +
          mensagem);
17
18  }
19 }
```

Uma vez que o canal de comunicação de entrada de dados foi instanciado, o servidor está apto para receber dados do cliente. Um objeto `DataInputStream` oferece diversos métodos para receber diferentes tipos de dados primitivos, tais como: `readUTF()`, `readInt()`, `readDouble()`, `readChar()`, etc. Em nosso exemplo, o objetivo será receber uma mensagem (String), então teremos que utilizar o método `readUTF()` expressado na linha 12 da Listagem 1.3. A Figura 2 auxilia na compreensão dos canais *InputStream e OutputStream*.

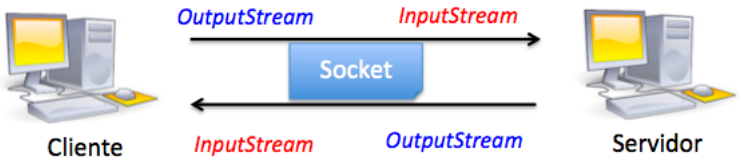

Figura 2: Esboço de fluxos de comunicação

Após da leitura da mensagem no canal de entrada (DataInputStream), pode-se manipular os dados de acordo com a necessidade. Em nosso exemplo, o objetivo é converter todos os caracteres dessa mensagem em maiúsculo. Para isso, utilizaremos o método `toUpperCase()` oferecido por um objeto de classe String (linha 13 - Listagem 1.3).

No término de uma aplicação baseada em Socket é altamente recomendado fechar o sub-canal de comunicação (os InputStreams, linha 17 - Listagem 1.4) e o canal de comunicação propriamente dito (Socket, linha 18 - Listagem 1.4). Essa ação é importante, pois é uma boa prática de programação liberar a porta de comunicação para que uma aplicação futura possa utilizá-la.

Listagem 1.4: Fechando streams e conexão

```
1 import java.net.*;
2 import java.io.*;
3
4 public class Servidor {
5   public static void main(String args[]) throws IOException {
6     ServerSocket servidor = new ServerSocket(54321);
7     System.out.println("A porta 54321 foi aberta.");
8
9     Socket conexao = servidor.accept();
10
11    DataInputStream entrada = new DataInputStream(conexao.
          getInputStream());
12
13    String mensagem = entrada.readUTF();
14    mensagem = mensagem.toUpperCase();
15
16    System.out.println("A mensagem em maiusculo e: " +
          mensagem);
17
18    entrada.close();
19    conexao.close();
20  }
21 }
```

Desse modo, você pode conferir o resultado da classe Servidor do nosso exemplo toda implementada na Listagem 1.4.

Cliente O objetivo do lado cliente do nosso exemplo é receber uma mensagem via teclado do usuário e enviá-la ao servidor. Essa codificação é tão simples quanto o cliente.

A primeira diferença entre a codificação do cliente e servidor é a ausência de um objeto `ServerSocket`. No cliente, apenas será necessário instanciar um objeto da classe `Socket` que irá representar uma conexão ao servidor. Para estabelecer uma conexão com o servidor, é preciso instanciar um objeto da classe `Socket` utilizando o construtor sobrecarregado passando o endereço e a porta de comunicação utilizada, conforme apresentado na linha 5 da Listagem 1.5.

Listagem 1.5: Aplicação cliente

```
1 import java.net.*;
2 import java.io.*;
3
4 public class Cliente {
5   public static void main(String args[]) throws IOException {
6     Socket conexao = new Socket("127.0.0.1", 54321);
7     DataOutputStream saida = new DataOutputStream(conexao.
          getOutputStream());
8
```

```
9       BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
10      String mensagem = br.readLine();
11
12      saida.writeUTF(mensagem);
13
14      saida.close();
15      conexao.close();
16    }
17  }
```

Assim que a conexão é estabelecida, pode-se instanciar um objeto `DataOutputStream` para representar o canal de saída de dados, permitindo o cliente enviar dados ao servidor. Depois de receber os dados de entrada pelo usuário, pode-se utilizar o objeto `saida` para escrever no canal de envio de dados. Para isso, a classe `DataOutputStream` oferece métodos semelhantes à classe `DataInputStream` tais como: `writeUTF(String)`,`writeDouble(Double)`, `writeInt(Integer)`, `writeChar(Char)`, `writeBoolean(Boolean)`, etc. No caso de nosso exemplo, por estarmos tratando do envio de uma String, utilizaremos o método `writeUTF(String)`, passando a variável `mensagem` como parâmetro.

Da mesma forma que fizemos com a classe `Servidor`, precisamos fechar o OutputStream representado pelo objeto `saida` e a conexão representada pelo objeto conexao, conforme apresentado na Listagem 1.5 pelas linhas 13 e 14, respectivamente.

Nesse momento, podemos compilar ambas as classes e executá-las. O servidor deve ser executado sempre antes do cliente, pois este precisa estar apto para aguardar uma nova conexão (método `accept()`).

2.3 Múltiplas conexões

No exemplo anterior, o servidor é capaz de receber uma única conexão de um cliente.

Para tratar dois ou mais clientes simultaneamente, somente a classe `Servidor` precisará passar por modificações. É preciso criar uma *thread* logo após a execução do método `accept()`. A thread criada será responsável pelo tratamento da conexão estabelecida conforme a Listagem 1.6.

A primeira modificação surge a herança da classe `Thread`(linha 4). Já que precisamos tratar várias conexões **simultâneas**, é preciso realizar o uso de múltiplos fluxos de execução para tal.

A adição de um atributo de classe `socket` permitirá que a aplicação consiga gerenciar diferentes conexões simultaneamente em cada *thread*. No construtor da classe `ServerThread` recebe-se como parâmetro a referência de um objeto `Socket` para ser atribuída ao atributo da classe `socket`.

O método `run()` será executado para cada *thread* criada. Assim, observa-se que na linha 16 é atribuído ao objeto `entrada` o canal de entrada da conexão tratada pela *thread*. Posteriormente, os demais códigos de recebimento e processamento dos dados são executados nas linhas 18, 19 e 20. Não diferente dos

exemplos anteriores, é preciso fechar os fluxos de dados e canais de comunicação (linhas 22 e 23).

Listagem 1.6: Servidor múltiplas conexões utilizando Thread

```
1  import java.net.*;
2  import java.io.*;
3
4  public class ServerThread extends Thread {
5
6    private Socket socket;
7
8    public ServerThread(Socket conn) {
9      this.socket = conn;
10   }
11
12   @Override
13   public void run() {
14
15     try {
16       DataInputStream entrada = new DataInputStream(socket.
             getInputStream());
17
18       String mensagem = entrada.readUTF();
19       mensagem = mensagem.toUpperCase();
20       System.out.println("A mensagem em maiusculo e: " +
             mensagem);
21
22       entrada.close();
23       socket.close();
24     } catch (IOException ioe) {
25       System.out.println("Erro : " + ioe.toString());
26     }
27   }
28
29   public static void main(String args[]) throws IOException {
30
31     ServerSocket servidor = new ServerSocket(54321);
32
33     while(true) {
34
35       Socket conexao = servidor.accept();
36       System.out.println("Um cliente se conectou...");
37       ServerThread thread = new ServerThread(conexao);
38       thread.start();
39     }
40   }
41 }
```

O método `main()` irá apenas definir em qual porta de comunicação o servidor irá monitorar (linha 31) e através do uso de um laço infinito, emprega-se a linha que sempre deixará o servidor pronto para receber uma nova conexão (linha 35) e assim que um cliente o fizer, esta conexão é adicionada em um nova *thread*.

2.4 Envio e recebimento de objetos

Até o momento foi apresentado as classes de fluxo de dados `DataInputStream` e `DataOutputStream`. Essas duas classes tratam somente o fluxo de dados de tipos primitivos como *double, int, char, boolean*, entre outros. Contudo, ainda não foi abordado o envio/recebimento de objetos utilizando Socket.

Aplicações que necessitam a transmissão de objetos inteiros para um outro computador, podem utilizar a classe `ObjectInputStream` e `ObjectOutputStream` no lugar das classes `DataInputStream` e `DataOutputStream`. Objetos oriundos da classe `ObjectInputStream` oferecem o método `readObject()`. Assim que invocado, este método retornará uma sequência de bytes que precisam ser convertidos por meio de um operador *cast* para a sua classe. Por exemplo:

```
ObjectInputStream entrada = new ObjectInputStream(
                conexao.getInputStream);
Pessoa p = (Pessoa) entrada.readObject();
```

A partir do momento em que o objeto é lido e formatado (convertido) a uma classe, seus métodos e atributos estarão aptos para serem usados.

Outro detalhe importante é serializar a classe do objeto que será transmitido através de uma *stream*. Isso pode ser feito implementando a interface `Serializable` conforme a linha 3 da Listagem 1.7.

Listagem 1.7: Exemplo de classe serializada

```
1  import java.io.*;
2
3  public class ClasseSerializada implements Serializable {
4
5    private String nome;
6    private int idade;
7
8    public ClasseSerializada() {
9    }
10
11   public void setNome(String nome) { this.nome = nome; }
12   public void setIdade(int idade) { this.idade = idade; }
13   public String getNome() { return this.nome; }
14   public int getIdade() { return this.idade; }
15 }
```

Para integralizar esse novo conteúdo, vamos considerar um exemplo em que o cliente define instancia e popula os dados de um objeto da classe `Pessoa` e, em seguida, envia o objeto ao servidor que acaba por mostrar os valores dos atributos do objeto.

O primeiro passo neste exemplo é implementarmos a classe `Pessoa`, conforme a Listagem 1.8 e, principalmente, não esquecendo de serializá-la.

Listagem 1.8: Classe Pessoa serializada

```
1  import java.io.*;
2
3  public class Pessoa implements Serializable {
4
5    private String nome;
6    private int idade;
7
8    public Pessoa() {
9    }
10
11   public Pessoa(String nome, int idade) {
12     this.nome = nome;
13     this.idade = idade;
14   }
15
16   public void setNome(String nome) { this.nome = nome; }
17   public void setIdade(int idade) { this.idade = idade; }
18   public String getNome() { return this.nome; }
19   public int getIdade() { return this.idade; }
20 }
```

A seguir, podemos implementar o código do Servidor através da classe `ServidorObjeto.java`, mostrada na Listagem 1.9. Note que houveram poucas mudanças. A primeira delas é na linha 4 em que a exceção `ClassNotFoundException` é repassada. A próxima refere-se à declaração de um objeto `entrada` a partir de uma classe `ObjectInputStream` (linha 11) . E a última, é a declaração de um objeto utilizando um operador *cast* ao formato do objeto que será recebido (linha 13).

Listagem 1.9: Classe Servidor

```
1  import java.net.*;
2  import java.io.*;
3
4  public class ServidorObjeto {
5
6    public static void main(String args[]) throws IOException,
         ClassNotFoundException {
7
8      ServerSocket servidor = new ServerSocket(5555);
9      Socket conexao = servidor.accept();
```

```
10
11      ObjectInputStream entrada = new ObjectInputStream(conexao
            .getInputStream());
12
13      Pessoa p = (Pessoa) entrada.readObject();
14
15      System.out.println("Nome: " + p.getNome() + "\nIdade: " +
             p.getIdade());
16
17      entrada.close();
18      conexao.close();
19    }
20  }
```

A codificação do Cliente é realizada na classe `ClienteObjeto.java`, apresentada na Listagem 1.10. As mudanças em relação à versão anterior que apenas transmite tipos primitivos, está na instanciação de um objeto `ObjectOutputStream` para o envio de um objeto (linha 9) e na linha 13 que escreve o objeto `p` da classe `Pessoa` através do método `writeObject`.

Listagem 1.10: Classe Cliente

```
1  import java.net.*;
2  import java.io.*;
3
4  public class ClienteObjeto {
5
6    public static void main(String args[]) throws IOException,
         ClassNotFoundException {
7
8      Socket conexao = new Socket("127.0.0.1", 5555);
9      ObjectOutputStream saida = new ObjectOutputStream(conexao
           .getOutputStream());
10
11     Pessoa p = new Pessoa("Henrique", 45);
12
13     saida.writeObject(p);
14
15     saida.close();
16     conexao.close();
17   }
18 }
```

2.5 Exercícios

1. Crie um programa em que o cliente receba a altura e peso de um usuário. Em seguida, o cliente deve enviar esses dois valores ao servidor. Ao receber os dois valores, o servidor deve realizar o cálculo do IMC (Índice de Massa Corporal)

cuja fórmula é IMC=peso/(altura*altura). Assim que o IMC for calculado pelo servidor, este deve enviar o resultado ao cliente que irá mostrar o valor do IMC do usuário.

2. Desenvolva uma aplicação que no lado cliente popule uma **lista** de pessoas cujos dados são: nome, cpf, endereco, telefone, email e idade. Após popular essa lista de pessoas no cliente, deve-se enviar todos os objetos dessa lista de pessoas ao servidor. O servidor ao receber cada objeto (pessoa) deverá exibir em sua tela apenas as pessoas cuja idade seja superior a 18 anos.