Programação de Redes de Computadores em Java

Rafael Santos

Material reescrito para os alunos da disciplina **CAP312 – Programação de Redes de Computadores** do programa de pós-graduação em Computação Aplicada do Instituto Nacional de Pesquisas Espaciais (INPE)

Última modificação: 17 de junho de 2006

http://www.lac.inpe.br/∼rafael.santos

Sumário

1 Introdução **1**

2 Programação Cliente-Servidor **1**
2.1 Introdução 1
2.2 Conceitos 3

3 Desenvolvimento de aplicações cliente-servidor em Java **4**

4 Clientes simples **8**
4.1 Exemplo: Cliente de `daytime` 8
4.2 Exemplo: Cliente de `echo` 10

5 Servidores simples (requisições não-simultâneas) **13**
5.1 Exemplo: Servidor de strings revertidas 13
5.2 Exemplo: Servidor e cliente de instâncias de classes 17
5.3 Exemplo: Servidor e cliente de números aleatórios 21
5.4 Exemplo: Servidor simples de arquivos 25

6 Servidores para múltiplas requisições simultâneas **31**
6.1 O problema de múltiplas requisições simultâneas 31
6.2 Linhas de execução (*threads*) 35
6.3 Exemplo: Servidor de números aleatórios (para requisições simultâneas) 39

7 Aplicações baseadas em um servidor e vários clientes **42**
7.1 Exemplo: Servidor de jogo-da-velha 43

8 Aplicações baseadas em um cliente e vários servidores **49**
8.1 Exemplo: Cliente e servidores para cálculo de integrais 51

9 Mais informações **58**

Lista de Figuras

1 Algoritmo para leitura e processamento de uma string 5
2 Algoritmo para o servidor de strings 5
3 Algoritmo integrado do cliente e servidor de strings 6
4 Outro algoritmo integrado do cliente e servidores 7
5 Protocolo de comunicação entre o cliente e o servidor de `daytime` . . . 8
6 Protocolo de comunicação entre o cliente e o servidor de `echo` 11
7 Protocolo de comunicação entre o cliente e o servidor de inversão de strings 14

8 Exemplo de acesso ao servidor de strings invertidas usando telnet . . 17
9 Protocolo de comunicação entre o cliente e o servidor de instâncias da classe Livro 18
10 Protocolo de comunicação entre o cliente e o servidor de números aleatórios 22
11 Protocolo de comunicação entre o cliente e o servidor de arquivos . . . 26
12 Protocolo de comunicação entre o cliente e o servidor de números aleatórios (segunda versão) 32
13 Posições para jogo e linhas vencedoras no jogo-da-velha 43
14 Protocolo de comunicação entre o servidor e os clientes de jogo-da-velha 44
15 Exemplo de interação entre um cliente (telnet) e servidor de jogo-da-velha 50
16 Cálculo de uma integral usando partições e trapézios 51
17 Protocolo para aplicação que faz o cálculo de uma integral 53

Lista de Listagens

1 Cliente de daytime 8
2 Cliente de echo 10
3 Servidor de strings invertidas 13
4 A classe Livro 17
5 Servidor de instâncias da classe Livro 18
6 Cliente para o servidor de livros 20
7 Servidor de números aleatórios 21
8 Cliente de números aleatórios 24
9 Servidor de arquivos 26
10 Cliente de arquivos 29
11 Segunda versão do servidor de números aleatórios 31
12 Segunda versão do cliente de números aleatórios 34
13 Classe que representa um carro de corrida para simulação. 35
14 Simulação usando instâncias de CarroDeCorrida. 36
15 Classe que representa um carro de corrida para simulação (herdando de Thread). 37
16 Simulação usando instâncias de CarroDeCorridaIndependente. . . . 38
17 Classe que implementa o serviço de números aleatórios 40
18 Terceira versão do servidor de números aleatórios 41
19 O servidor de Jogo-da-Velha 45
20 O servidor de cálculo de integrais 52
21 Uma linha de execução para o cliente de cálculo de integrais 55
22 O cliente de cálculo de integrais 57

1 Introdução

Este documento ilustra os conceitos básicos de programação cliente-servidor usando Java, usando protocolos, clientes e servidores desenvolvidos pelo programador. Para melhor compreensão dos tópicos, alguns clientes simples para servidores já existentes serão demonstrados.

Para a compreensão dos exemplos deste documento, o leitor deve ter os seguintes conhecimentos técnicos:

- Conhecer algoritmos e a implementação de diversas estruturas de controle em Java (condicionais, estruturas de repetição);
- Programação orientada a Objetos usando a linguagem Java. Em especial é necessário compreender os conceitos de encapsulamento e herança, mecanismo de exceções e o uso de métodos estáticos;
- Conhecer alguns dos mecanismos de entrada e saída em Java (*streams*);
- Conhecer conceitos básicos do funcionamento de um computador ligado em uma rede.

2 Programação Cliente-Servidor

2.1 Introdução

Consideremos os seguintes cenários:

1. Precisamos executar uma aplicação específica mas o computador ao qual temos acesso não pode executá-la por uma razão qualquer (exemplos podem ser falta de hardware adequado, como disco, processador ou memória; necessidade de acesso a um hardware específico para execução do software, etc.).
2. Precisamos de uma informação para efetuar um processamento ou tomar uma decisão, mas esta informação é atualizada frequentemente. Existe um outro computador que tem a versão mais atual desta informação.
3. Precisamos executar uma tarefa usando computadores que precise de mediação, ou seja, de uma maneira imparcial e independente de avaliar resultados. Frequentemente existirão várias fontes de informação que precisam ser avaliadas, e em muitos casos as fontes devem permanecer ocultas umas das outras.
4. Precisamos executar uma aplicação cujo tempo para conclusão seria excessivamente longo no computador ao qual temos acesso, e possivelmente muito longo mesmo usando hardware mais eficiente.

5. Precisamos obter informações que estão em algum outro computador (de forma semelhante ao caso 2) mas não sabemos onde estas informações estão localizadas. Sabemos, no entanto, que um outro computador contém um catálogo destas informações.

No caso do exemplo 1, poderíamos "emprestar" o hardware necessário sem precisarmos mudar fisicamente de lugar. Basta que exista a possibilidade de enviarmos a tarefa para este computador remoto e receber o resultado. Um exemplo deste cenário seria a consulta a bancos de dados do Genoma, que são grandes e complexos demais para serem reproduzidos em computadores pessoais.

O caso do exemplo 2 é muito similar ao uso da Internet para acesso a informações. O exemplo mais claro é o acesso a jornais *on-line*: como a informação é modificada a cada dia (ou, em alguns casos, o tempo todo), um usuário pode simplesmente recarregar, através de uma página, a informação desejada atualizada. Um outro exemplo mais simples seria de um registro global de tempo, que teria a hora certa em um computador especial, e outros computadores poderiam acertar seus relógios internos usando a hora fornecida por este computador especial.

Para o exemplo 3 podemos pensar em enviar as informações ou resultados de diversas fontes para um computador que tomaria a decisão assim que tivesse todas as fontes de dados. A aplicação mais simples deste tipo de cenário seriam jogos de computador, onde cada jogador enviaria suas informações (por exemplo, posições de um tabuleiro) e o computador mediador decidiria se as jogadas foram válidas e quem seria o vencedor.

Para o exemplo 4 podemos considerar uma solução cooperativa: se o problema a ser resolvido puder ser dividido em vários subproblemas menores independentes, poderíamos usar vários computadores diferentes para resolver cada um destes pequenos problemas. Um computador seria o responsável para separar o problema em subproblemas, enviar estes subproblemas para diversos outros computadores e integrar o resultado. Um exemplo deste tipo de cenário é a análise de dados de genoma e proteoma, onde usuários cedem parte do tempo de seus computadores para execução de uma tarefa global.

No caso do exemplo 5, poderíamos ter os computadores dos usuários consultando um computador central que não contém informações mas sabe quais computadores as contém. Este computador central teria então meta-informações ou meta-dados, ou seja, informações sobre informações. Este cenário é semelhante ao que ocorre em mecanismos de busca na Internet, que não contém páginas com uma informação específica mas podem indicar quais computadores as contém.

Em todos estes casos, estaremos usando alguma variante de aplicação cliente-servidor, onde parte do processamento dos dados é feito do lado do cliente ou usuário que deseja

processar a informação; e parte processamento do lado do servidor, ou do computador que é capaz de processar ou obter a informação desejada. Em alguns casos teremos a relação um cliente para um servidor (como nos cenários 1 e 2), mas é possível considerar arquiteturas onde existem vários clientes para um servidor (cenário 3), vários servidores para um cliente (cenário 4) ou mesmo várias camadas de servidores (cenário 5). Em todos estes casos podemos supor que os computadores estão conectados via rede local, Internet ou outro mecanismo remoto qualquer.

2.2 Conceitos

Alguns conceitos cuja compreensão se faz necessária são listados a seguir.

Servidor é o computador que contém a aplicação que desejamos executar via remota.

Serviço é uma aplicação sendo executada no servidor. Para cada tipo de serviço (aplicação) pode ser necessário ter um tipo de cliente específico, capaz de realizar a comunicação da forma adequada. É comum usar os termos "serviço" e "servidor" para designar a aplicação a ser executada (ex. servidor de FTP).

Cliente é a aplicação remota que fará a comunicação e interação com o serviço/servidor.

Endereço é a informação da localização de um computador em uma rede local ou na Internet. Podem ser usados como endereços o número IP (*Internet Protocol*) do computador ou um nome que possa ser resolvido por um servidor DNS (*Domain Name System*).

Porta é um endereço local em um computador conectado a uma rede, identificado por um número único. Todos os dados originados ou destinados a um computador na rede passam pela conexão (geralmente única) daquele computador, identificada pelo seu endereço. Como várias aplicações podem estar enviando e recebendo dados, é necessário um segundo identificador para que o computador saiba que dados devem ir para cada aplicação. Este segundo identificador é a porta, associada com um serviço.

Porta Bem Conhecida é uma porta cujo número é menor que 1024, e que corresponde a serviços bem conhecidos. Alguns exemplos são as portas 7 (serviço *echo*), 13 (serviço *daytime*), 21 (serviço *ftp*) e 80 (serviço *http*). Usuários comuns (isto é, sem privilégios de administração) geralmente não podem usar números de portas bem conhecidas para instalar serviços.

Protocolo é o diálogo que será travado entre cliente e servidor[1]. Este diálogo é muito importante quando vamos criar clientes e servidores específicos, pois dados devem ser enviados do cliente para o servidor em uma ordem e formatos predeterminados.

[1] O termo *protocolo* é mais conhecido para designar a maneira pela qual os dados serão recebidos e enviados entre cliente e servidor; alguns exemplos mais conhecidos são TCP (*Transmission Control Protocol*) e UDP (*User Datagram Protocol*).

O protocolo define que partes do algoritmo que resolve o problema em questão serão resolvidas pelo cliente, resolvidas pelo servidor ou enviadas de/para o cliente e o servidor.

Socket é o terminal de um canal de comunicações de duas vias entre cliente e servidor. Para que a comunicação entre cliente e servidor seja efetuada, o cliente e o servidor criarão cada um um *socket* que serão associados um com o outro para o envio e recebimento de informações.

3 Desenvolvimento de aplicações cliente-servidor em Java

Alguns dos exemplos comentados na seção 2.1 podem ser resolvidos com servidores e clientes já existentes – por exemplo, para receber notícias de um jornal basta usarmos um navegador que acessará o servidor HTTP da empresa que mantém o jornal. Em muitos casos, entretanto, teremos que escrever o servidor pois ele fará tarefas específicas que não são especialidade de servidores já existentes. Em alguns casos, também deveremos escrever um cliente especial pois o protocolo de comunicação com o servidor e os tipos de dados que serão enviados e recebidos devem ter tratamento especial.

Uma aplicação cliente-servidor em Java segue um padrão relativamente simples de desenvolvimento: a parte complexa é determinar que parte do processamento será feita pelo cliente e que parte pelo servidor, e como e quando os dados deverão trafegar entre o cliente e o servidor. Imaginemos uma aplicação onde o cliente deva recuperar uma linha de texto enviada pelo servidor (a linha de texto pode conter a hora local, uma cotação de moeda, etc.). O algoritmo do cliente (mostrado como um fluxograma ou diagrama semelhante a um diagrama de atividades de UML) seria como o mostrado na figura 1.

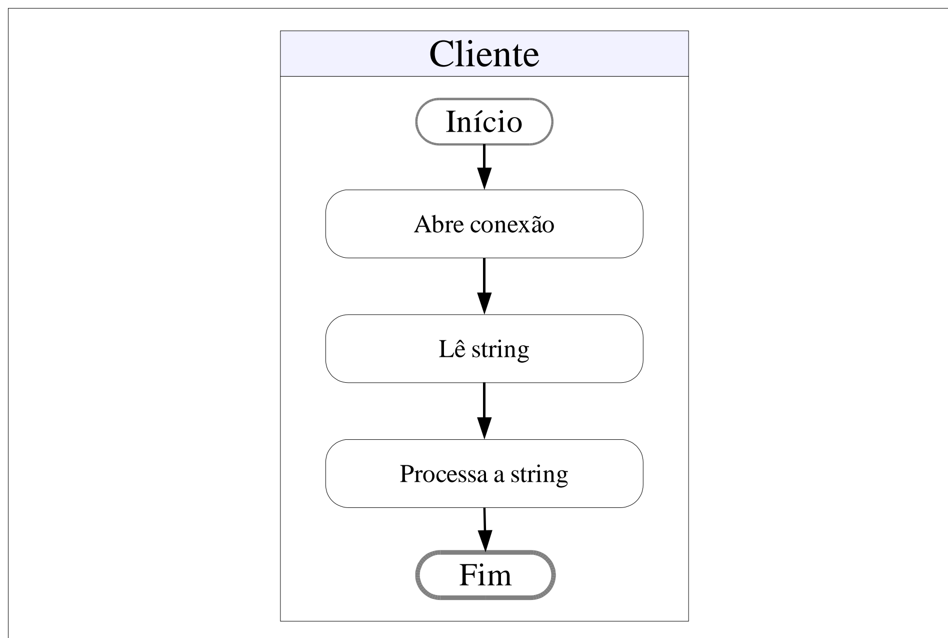

Figura 1: Algoritmo para leitura e processamento de uma string

O diagrama/algoritmo para o servidor é o mostrado na figura 2.

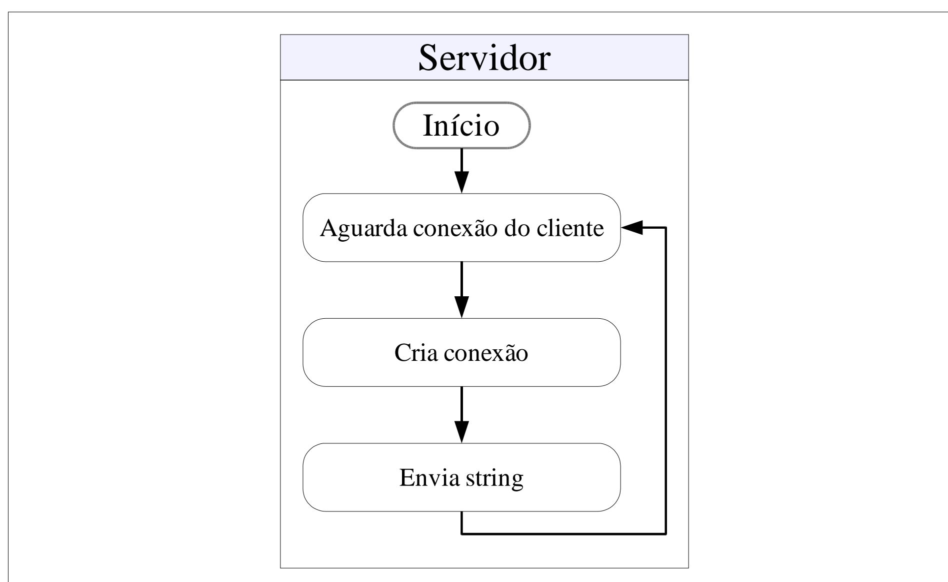

Figura 2: Algoritmo para o servidor de strings

O algoritmo mostrado na figura 1 não está completo: após o passo "Solicita conexão" a aplicação deveria aguardar a conexão entre o cliente e servidor ser estabelecida antes de ler a string enviada pelo servidor. Da mesma forma, o algoritmo do servidor (mostrado na figura 2) não cria conexões e envia strings a qualquer hora: estes passos ocorrem em resposta a solicitações do cliente. Os dois algoritmos devem então estar **sincronizados** para a aplicação funcionar corretamente.

Para descrevermos a aplicação como um todo usando diagramas similares aos de ati-

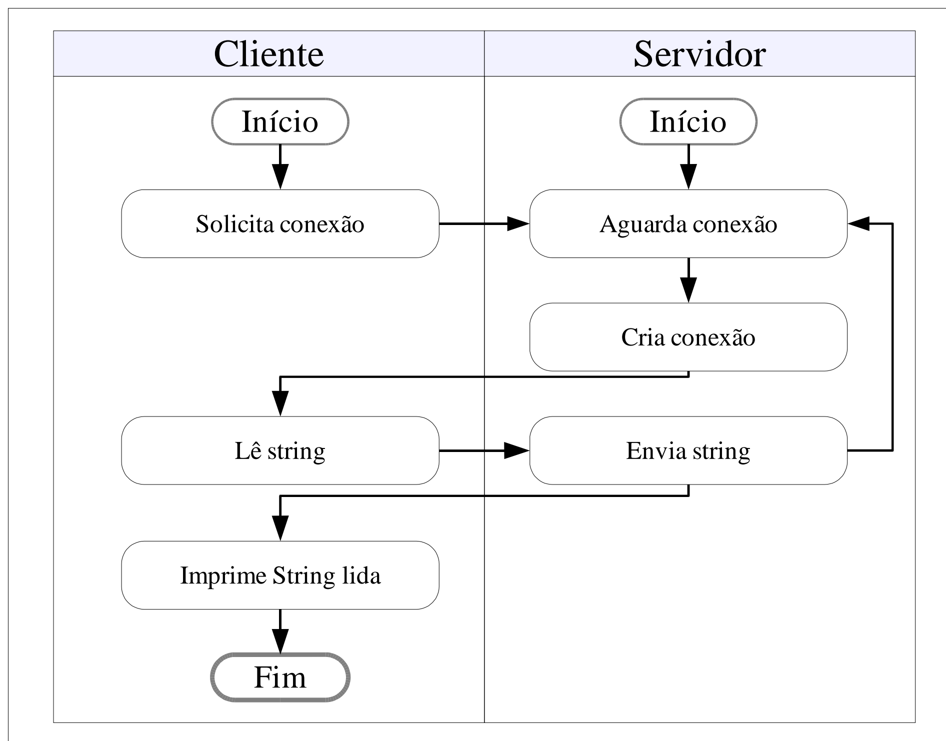

Figura 3: Algoritmo integrado do cliente e servidor de strings

vidades[2] devemos descrever os algoritmos lado a lado, inclusive registrando os passos nos quais os algoritmos devem estar sincronizados. Para isto, basta mudar o fluxo dos passos para incluir a dependência de passos no cliente e servidor. Um algoritmo que mostra as atividades do cliente e servidor ilustrando os passos que precisam ser sincronizados é mostrado na figura 3.

Um outro exemplo de diagrama simplificado que mostra a interação entre um cliente e dois servidores é mostrada na figura 4. Este algoritmo é de uma aplicação hipotética na qual um cliente verifica preços em dois servidores diferentes para comparação e decisão.

É interessante observar no algoritmo da figura 4 que alguns passos poderiam estar em ordem diferente sem afetar o funcionamento da aplicação como um todo. Por exemplo, as conexões do cliente com os servidores poderiam ser estabelecidas uma logo após a outra assim como a leitura dos dados. Como não existe um tempo de processamento grande entre o estabelecimento da conexão e a leitura dos dados, esta modificação não causaria tempo de conexão sem uso desnecessariamente longo com o servidor.

[2] Evitando usar as notações de bifurcação e união para manter os diagramas simples

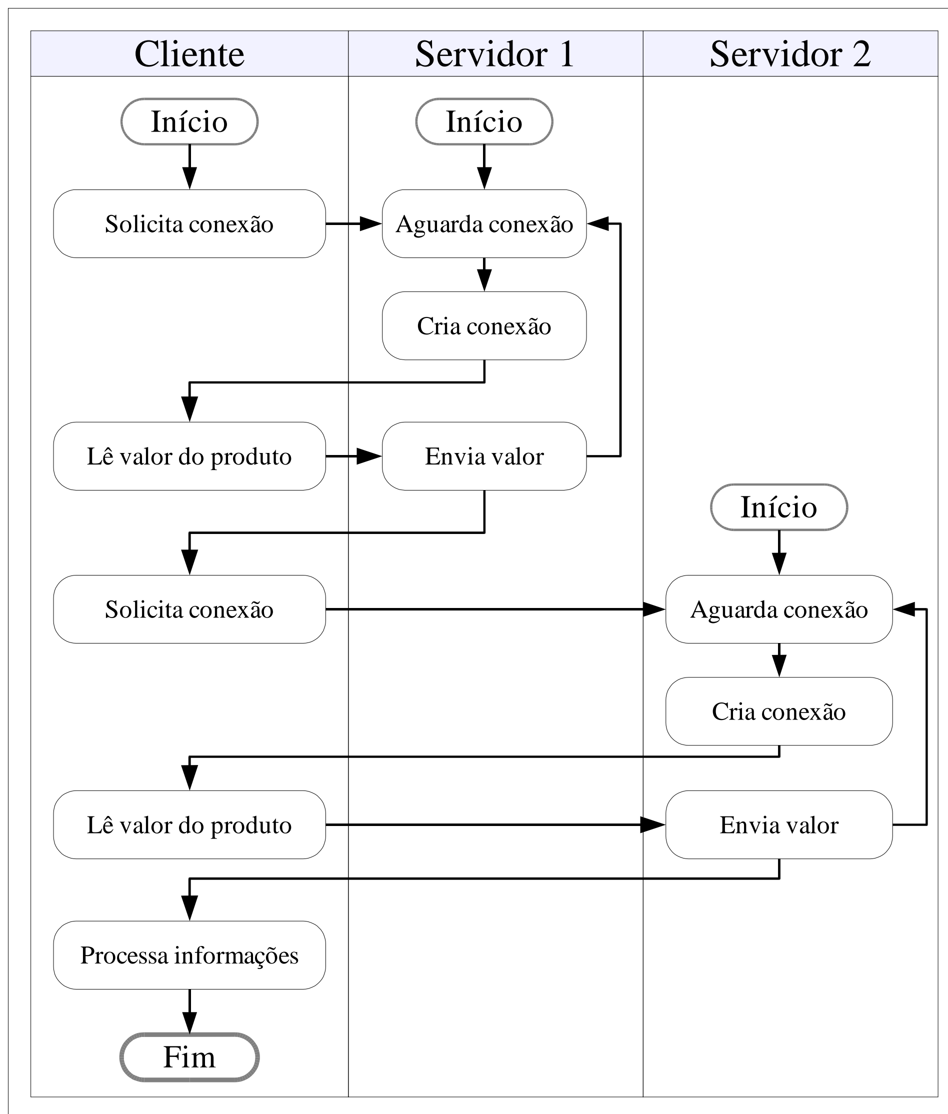

Figura 4: Outro algoritmo integrado do cliente e servidores

4 Clientes simples

Veremos nesta seção alguns clientes simples para serviços existentes[3]. Estes clientes são totalmente funcionais, e servem para demonstrar o uso das classes para comunicação entre clientes e servidores.

4.1 Exemplo: Cliente de `daytime`

O serviço `daytime` é registrado na porta 13 e retorna, quando conectado e solicitado, uma string contendo a data e hora do servidor. O protocolo de comunicação com um servidor destes é, então, muito simples: basta estabelecer a conexão, ler uma única string e encerrar a conexão. Este protocolo é mostrado na figura 5.

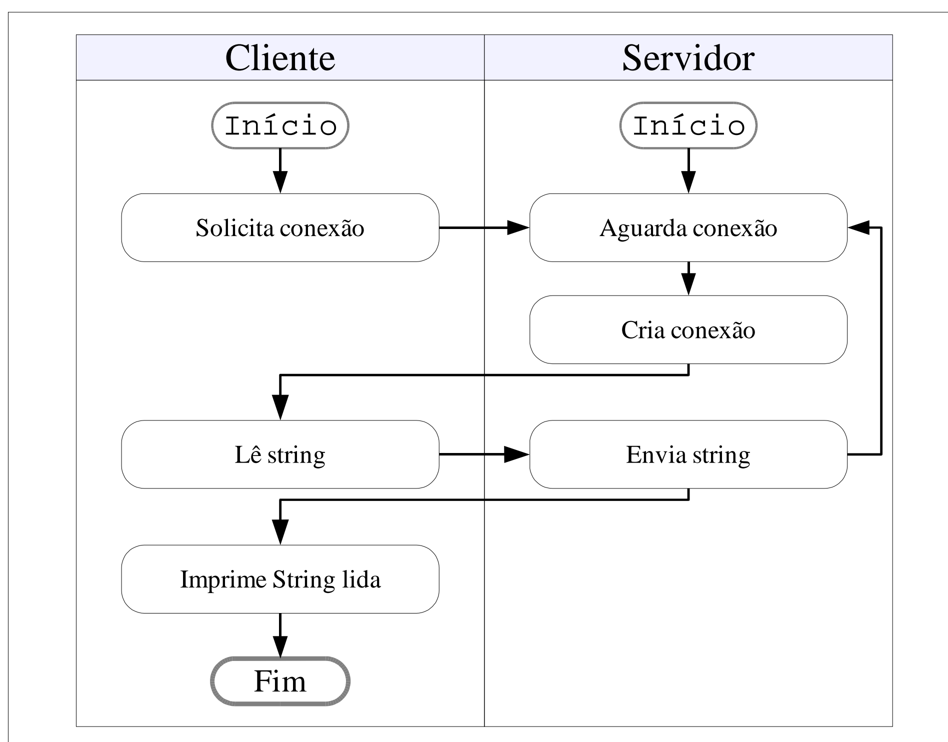

Figura 5: Protocolo de comunicação entre o cliente e o servidor de `daytime`

No sistema operacional Linux, o serviço `daytime` pode ser acessado com o aplicativo `telnet`, que deve receber como argumentos o nome do servidor e o número da porta, separados por espaço (por exemplo, `telnet localhost 13`). Ao invés de acessar o serviço desta forma, vamos escrever uma aplicação em Java que o faça para nós. A aplicação é mostrada na listagem 1.

Listagem 1: Cliente de `daytime`

```
1 package cap312;
2 import java.io.BufferedReader;
3 import java.io.IOException;
```

[3] Os serviços utilizados nesta seção são serviços padrão do sistema operacional Linux, desabilitados por *default*. Para computadores executando o sistema operacional Red Hat, verifique os arquivos no diretório `/etc/xinetd.d` para habilitação dos serviços.

```
4  import java.io.InputStreamReader;
5  import java.net.Socket;
6  import java.net.UnknownHostException;
7
8  /**
9   * Esta classe implementa um cliente simples para o serviço daytime (porta 13).
10  * A classe simplesmente cria um socket para um servidor (no exemplo,
11  * localhost) usando aquela porta, obtém um BufferedReader (usando um
12  * InputStreamReader a partir do InputStream da conexão) e lê uma linha do
13  * servidor.
14  */
15 public class ClienteDeDaytime
16   {
17   public static void main(String[] args)
18     {
19     String servidor = "localhost";
20     // Tentamos fazer a conexão e ler uma linha...
21     try
22       {
23       Socket conexão = new Socket(servidor,13);
24       // A partir do socket podemos obter um InputStream, a partir deste um
25       // InputStreamReader e partir deste, um BufferedReader.
26       BufferedReader br =
27         new BufferedReader(
28            new InputStreamReader(conexão.getInputStream()));
29       String linha = br.readLine();
30       System.out.println("Agora são "+linha+" no servidor "+servidor);
31       // Fechamos a conexão.
32       br.close();
33       conexão.close();
34       }
35     // Se houve problemas com o nome do host...
36     catch (UnknownHostException e)
37       {
38       System.out.println("O servidor não existe ou está fora do ar.");
39       }
40     // Se houve problemas genéricos de entrada e saída...
41     catch (IOException e)
42       {
43       System.out.println("Erro de entrada e saída.");
44       }
45     }
46   }
```

Os pontos interessantes e chamadas a APIs da listagem 1 são:

- O primeiro passo do algoritmo (a criação de uma conexão) é feita pela criação de uma instância da classe `Socket`. O construtor desta classe espera um nome de servidor (ou o seu IP, na forma de uma string) e o número da porta para conexão. A criação de uma instância de `Socket` pode falhar, criando a exceção `UnknownHostException` se o endereço IP do servidor não puder ser localizado ou `IOException` se houver um erro de entrada ou saída na criação do *socket*.

- Podemos obter uma *stream* de entrada a partir da instância de `Socket` (usando o método `getInputStream`), e a partir desta *stream* podemos criar uma instância de `InputStreamReader` (que transforma os bytes lidos pela *stream* em caracteres). A

partir desta instância de `InputStreamReader` (usando-a como argumento para o construtor) podemos criar uma instância de `BufferedReader` que pode ser usada para ler linhas de texto completas.

- O resto do processamento é realmente simples, lemos uma string do servidor (na verdade, solicitamos a leitura, e o servidor criará esta string para ser enviada quando for requisitado), imprimimos a string lida e fechamos a *stream* e conexão.

Uma nota interessante sobre a aplicação mostrada na listagem 1: após terminar a leitura da string vinda do servidor, devemos manualmente fechar a conexão com o mesmo (pois sabemos, de acordo com seu protocolo, que somente uma string será enviada a cada conexão). Se usarmos o aplicativo `telnet` para acessar este serviço, o mesmo automaticamente encerrará a conexão com o servidor.

4.2 Exemplo: Cliente de `echo`

Um serviço ligeiramente mais complexo que o `daytime` é o `echo`, que responde por conexões na porta 7 e retorna cada string enviada pelo cliente de volta. O servidor reenvia cada string enviada pelo cliente até que a string seja um sinal de término (o caracter de controle `Control+]` no caso do cliente `telnet`) indicando o fim da interação. O cliente precisa, então, de um mecanismo que envie e receba dados do servidor, sendo que estes dados vão ser somente texto (strings). A figura 6 ilustra o protocolo de comunicação entre um cliente e um servidor de `echo`.

A diferença principal entre o diagrama mostrado na figura 6 e outros vistos anteriormente é que este tem uma ramificação possível (condicional), dependendo da entrada do usuário um dos dois possíveis caminhos será tomado. Para a implementação em Java, ao invés de usar o código de controle do `telnet`, usaremos uma string composta do caracter ponto ("."). Se esta string for enviada, fecharemos a conexão com o servidor.

A implementação do cliente de `echo` é mostrada na listagem 2.

Listagem 2: Cliente de `echo`

```
1  package cap312;
2  import java.io.BufferedReader;
3  import java.io.BufferedWriter;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.io.OutputStreamWriter;
7  import java.net.Socket;
8  import java.net.UnknownHostException;
9
10 /**
11  * Esta classe implementa um cliente simples para o serviço echo (porta 7).
12  * A classe simplesmente cria um socket para um servidor (no exemplo,
```

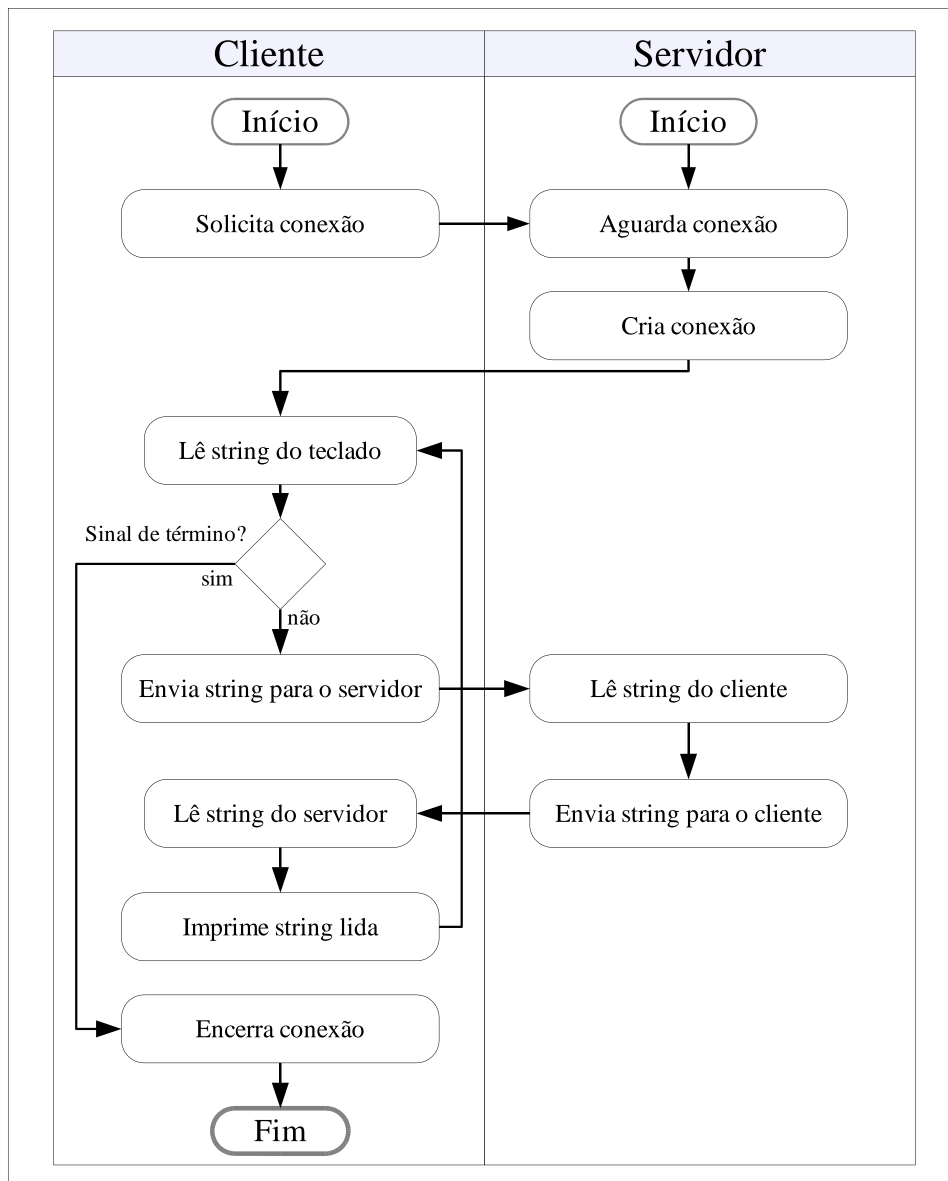

Figura 6: Protocolo de comunicação entre o cliente e o servidor de `echo`

```
13  * localhost) usando aquela porta, obtém um BufferedReader (usando um
14  * InputStreamReader a partir do InputStream da conexão) e um BufferedWriter
15  * (usando um OutputStreamWriter a partir do OutputStream da conexão), e repete
16  * o seguinte laço:
17  * 1 - Lê uma string do teclado
18  * 2 - Envia esta string para o servidor
19  * 3 - Lê uma string do servidor
20  * 4 - Imprime a string lida do servidor
21  * Se a string entrada via teclado for igual a um ponto (".") o laço será
22  * interrompido.
23  */
24 public class ClienteDeEcho
25   {
26   public static void main(String[] args)
27     {
28     String servidor = "localhost";
29     // Tentamos fazer a conexão e ler uma linha...
30     try
31       {
32       Socket conexão = new Socket(servidor,7);
```

```
33      // A partir do socket podemos obter um InputStream, a partir deste um
34      // InputStreamReader e partir deste, um BufferedReader.
35      BufferedReader br =
36        new BufferedReader(
37           new InputStreamReader(conexão.getInputStream()));
38      // A partir do socket podemos obter um OutputStream, a partir deste um
39      // OutputStreamWriter e partir deste, um Bufferedwriter.
40      BufferedWriter bw =
41        new BufferedWriter(
42           new OutputStreamWriter(conexão.getOutputStream()));
43      // Executamos este laço "para sempre":
44      while(true)
45        {
46        // Lemos uma linha do console.
47        String linhaEnviada = Keyboard.readString();
48        // Se o usuário tiver digitado Cancel, saímos do laço.
49        if (linhaEnviada.equals(".")) break;
50        // Enviamos a linha para o servidor.
51        bw.write(linhaEnviada);
52        bw.newLine();
53        bw.flush();
54        // Lemos uma linha a partir do servidor e a imprimimos no console.
55        String linhaRecebida = br.readLine();
56        System.out.println(linhaRecebida);
57        }
58      // Fechamos a conexão.
59      br.close();
60      bw.close();
61      conexão.close();
62      }
63    // Se houve problemas com o nome do host...
64    catch (UnknownHostException e)
65      {
66      System.out.println("O servidor não existe ou está fora do ar.");
67      }
68    // Se houve problemas genéricos de entrada e saída...
69    catch (IOException e)
70      {
71      System.out.println("Erro de entrada e saída.");
72      }
73    }
74  }
```

Os pontos interessantes da listagem 2 são:

- Novamente o primeiro passo é a criação de uma instância da classe `Socket`, novamente usando como exemplo o servidor `localhost` mas com a porta 7.
- Como é preciso enviar e receber strings do cliente para o servidor e vice-versa, precisamos criar mecanismos para o envio e recebimento de strings. Isto é feito separadamente: para o recebimento de strings enviadas do servidor, obtemos uma `stream` de leitura com o método `getInputStream` da instância de `Socket`, usamos esta *stream* para criar uma instância de `InputStreamReader` e usamos esta instância para criar uma instância de `BufferedReader`. Similarmente, usamos uma `stream` de escrita obtida com o método `getOutputStream` da instância de

Socket, para criar uma instância de OutputStreamWriter e usamos esta instância para criar uma instância de BufferedWriter.

- Após a criação das *streams* de entrada e saída, entramos no laço principal do cliente, que recebe uma string do teclado, compara com a string de término, e se for diferente, envia a string para o servidor, recebendo uma string de volta e repetindo o laço. Notem que após enviar uma string para o servidor, é necessário executar os métodos newLine e flush da classe BufferedWriter para que os bytes correspondentes à string sejam realmente enviados.
- Este exemplo usa a classe Keyboard para entrada de strings. Esta classe pode ser copiada do site http://www.directnet.com.br/users/rafael.santos/[4].

5 Servidores simples (requisições não-simultâneas)

5.1 Exemplo: Servidor de strings revertidas

Vamos ver um exemplo de servidor agora, ou seja, uma aplicação que vai enviar informações a um cliente. O cliente pode ser um aplicativo simples como o próprio telnet.

Considere que seja necessário por algum razão *inverter* uma string. A inversão de uma string é a simples modificação da ordem de seus caracteres para que a mesma seja lida ao contrário, ou da direita para a esquerda. Como um exemplo, a inversão da string "Java" seria "avaJ". O algoritmo de inversão é simples, basta criar uma string vazia e concatenar a esta string cada caracter da string original, lidos de trás para frente.

O protocolo de comunicação entre um cliente e o servidor de strings invertidas é mostrado na figura 7.

O diagrama mostra uma ênfase no passo "Inverte string" pois este realiza o que, em princípio, o cliente não saberia como ou teria recursos para processar. Em muitos casos de aplicações cliente-servidor este passo seria o mais crucial: o processamento da informação do lado do servidor.

A implementação do servidor de strings invertidas é mostrada na listagem 3.

Listagem 3: Servidor de strings invertidas

```
1 package cap312;
2 import java.io.BufferedReader;
3 import java.io.BufferedWriter;
4 import java.io.IOException;
5 import java.io.InputStreamReader;
```

[4] Curiosamente, se usarmos o método showInputDialog da classe JOptionPane, a aplicação não será terminada corretamente – para isso deveremos usar o método System.exit ao final da aplicação.

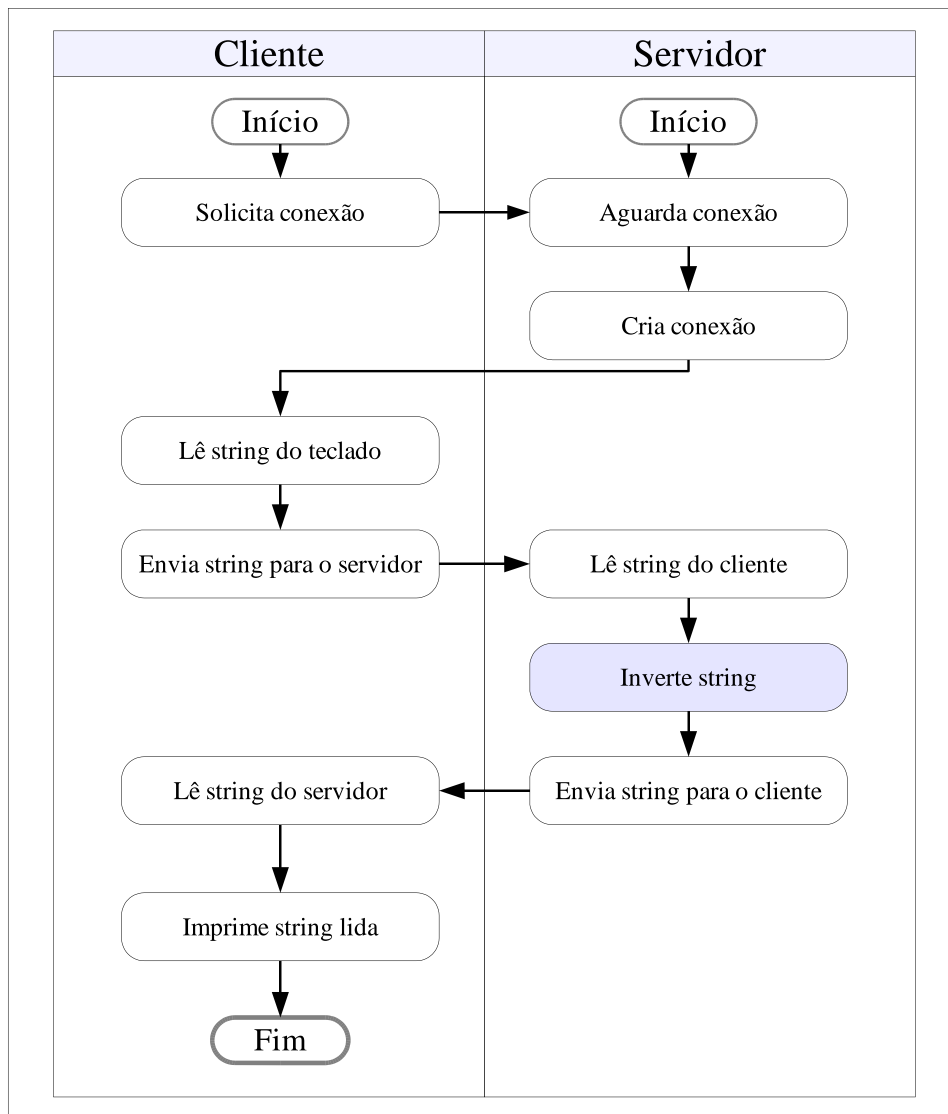

Figura 7: Protocolo de comunicação entre o cliente e o servidor de inversão de strings

```
6  import java.io.OutputStreamWriter;
7  import java.net.BindException;
8  import java.net.ServerSocket;
9  import java.net.Socket;
10
11 /**
12  * Esta classe implementa um servidor simples que fica aguardando conexões de
13  * clientes. Quando uma conexão é solicitada, o servidor recebe do cliente uma
14  * string, a inverte e envia este resuldado para o cliente, fechando a conexão.
15  */
16 public class ServidorDeStringsInvertidas
17   {
18   // Método que permite a execução da classe.
19   public static void main(String[] args)
20     {
21     ServerSocket servidor;
22     try
23       {
24       // Criamos a instância de ServerSocket que responderá por solicitações
25       // à porta 10101.
26       servidor = new ServerSocket(10101);
27       // O servidor aguarda "para sempre" as conexões.
```

```
28        while(true)
29          {
30          // Quando uma conexão é feita,
31          Socket conexão = servidor.accept();
32          // ... o servidor a processa.
33          processaConexão(conexão);
34          }
35        }
36      // Pode ser que a porta 10101 já esteja em uso !
37      catch (BindException e)
38        {
39        System.out.println("Porta já em uso.");
40        }
41      // Pode ser que tenhamos um erro qualquer de entrada ou saída.
42      catch (IOException e)
43        {
44        System.out.println("Erro de entrada ou saída.");
45        }
46      }
47
48    // Este método atende a uma conexão feita a este servidor.
49    private static void processaConexão(Socket conexão)
50      {
51      try
52        {
53        // Criamos uma stream para receber strings, usando a stream de entrada
54        // associado à conexão.
55        BufferedReader entrada =
56          new BufferedReader(new InputStreamReader(conexão.getInputStream()));
57        // Criamos uma stream para enviar strings, usando a stream de saída
58        // associado à conexão.
59        BufferedWriter saída =
60          new BufferedWriter(new OutputStreamWriter(conexão.getOutputStream()));
61        // Lemos a string que o cliente quer inverter.
62        String original = entrada.readLine();
63        String invertida = "";
64        // Invertemos a string.
65        for(int c=0;c<original.length();c++)
66          {
67          invertida = original.charAt(c)+invertida;
68          }
69        // Enviamos a string invertida ao cliente.
70        saída.write(invertida);
71        saída.newLine();
72        saída.flush();
73        // Ao terminar de atender a requisição, fechamos as streams de entrada e saída.
74        entrada.close();
75        saída.close();
76        // Fechamos também a conexão.
77        conexão.close();
78        }
79      // Se houve algum erro de entrada ou saída...
80      catch (IOException e)
81        {
82        System.out.println("Erro atendendo a uma conexão !");
83        }
84      }
85    }
```

Existem vários pontos interessantes na listagem 3, descritos a seguir:

- Para comodidade e compreensão de tópicos intermediários, a listagem foi dividida em duas partes (dois métodos): o método `main` que representa o fluxo principal do servidor (criação de *sockets* e laço principal) e o método `processaConexão` que será responsável por processar uma única conexão com um cliente.

- O método `main` cria uma instância de `ServerSocket` que será responsável por aguardar conexões do cliente. Uma instância de `ServerSocket` é criada passando-se para o seu construtor a porta na qual este servidor irá responder por conexões.

- O método `main` também contém um laço aparentemente infinito (`while(true)`) onde ficará aguardando conexões de clientes. Quando um cliente se conectar a este servidor, o método `accept` da classe `ServerSocket` criará uma instância de `Socket` para a comunicação com o cliente, e prosseguirá a execução do lado do servidor. É importante observar que enquanto o cliente não solicitar uma conexão, o método `accept` bloqueará o processamento do servidor, como se a aplicação estivesse "pausada" naquele ponto.
  Assim que uma conexão do cliente for aceita, o método `processaConexão` será executado, usando como argumento a instância recém-criada da classe `Socket`, correspondente à conexão com o cliente.

- Ainda no método `main` temos os blocos `catch` responsáveis pelo processamento das exceções que podem ocorrer neste método: `BindException` que será criada caso o endereço desejado já esteja em uso e `IOException` que será criada caso ocorra algum erro de entrada e saída genérico.

- O método `processaConexão` será responsável por processar uma única conexão com este servidor, criada quando o método `accept` foi executado. Este método criará *streams* para envio e recebimento de strings, conforme ilustrado e descrito na seção 4.2. É importante notar que a *stream* de entrada para o servidor será associado à stream de saída do cliente e vice-versa.
  O método `processaConexão` então recebe uma string do cliente, a inverte com um algoritmo simples e a envia para o cliente (garantindo que os bytes desta string serão realmente enviados usando os métodos `newLine` e `flush` da classe `BufferedWriter`).

- O método `processaConexão` também processa em um bloco `catch` a exceção `IOException` que pode ocorrer.

Para testar este servidor, um cliente simples como `telnet` ou mesmo a aplicação `ClienteDeEcho` (listagem 2) modificada para usar a porta 10101 poderia ser usado. Um exemplo de execução de interação com o servidor de strings invertidas usando o `telnet` é mostrado na figura 8.

5.2 Exemplo: Servidor e cliente de instâncias de classes

Até agora vimos clientes que se comunicam com servidores conhecidos ou servidores que podem usar clientes existentes como `telnet` Veremos agora um exemplo mais específico que exige que tanto o cliente quanto o servidor sejam escritos especialmente para atender ao protocolo de comunicação entre cliente e servidor (ou para poder receber e enviar dados que não sejam strings). Em outras palavras, o servidor será diferente dos existentes (`echo`, `daytime`) e o cliente não poderá ser o `telnet`.

Vamos ver como podemos escrever um servidor para servir instâncias de classes em Java. O cliente também deverá ser capaz de receber do servidor estas instâncias. Vamos usar a classe `Livro` (listagem 4) para criar as instâncias que serão enviadas.

Listagem 4: A classe `Livro`

```
package cap312;
import java.io.Serializable;

public class Livro implements Serializable
  {
  private String título;
  private String autores;
  private int ano;

  public Livro(String t,String aut,int a)
    {
    título = t;
    autores = aut;
    ano = a;
    }

  public void imprime()
    {
    System.out.println(título+" ("+autores+"),"+ano);
    }

  }
```

Para que instâncias de classes possam ser enviadas e recebidas por servidores e clientes é necessário que elas sejam *serializáveis*. Para isto, basta declarar a classe como implementando a interface `Serializable`, sem precisar implementar nenhum método

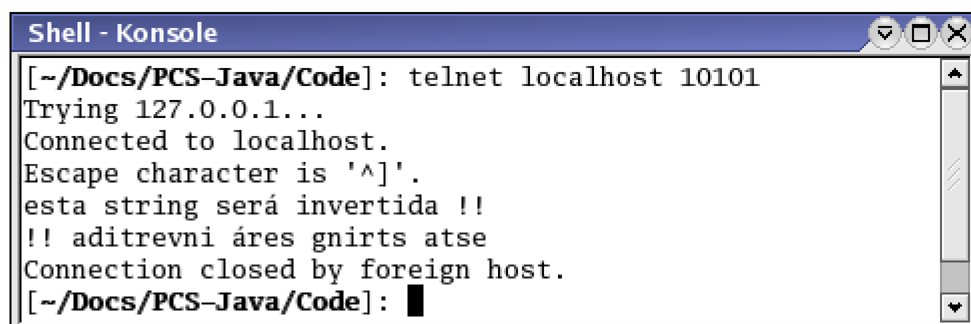

Figura 8: Exemplo de acesso ao servidor de strings invertidas usando `telnet`

adicional. A classe `Livro` (listagem 4) implementa esta interface.

O protocolo de comunicação entre o servidor e o cliente de instâncias da classe `Livro` é bem simples, e similar a outros mostrados anteriormente. Este protocolo é mostrado na figura 9.

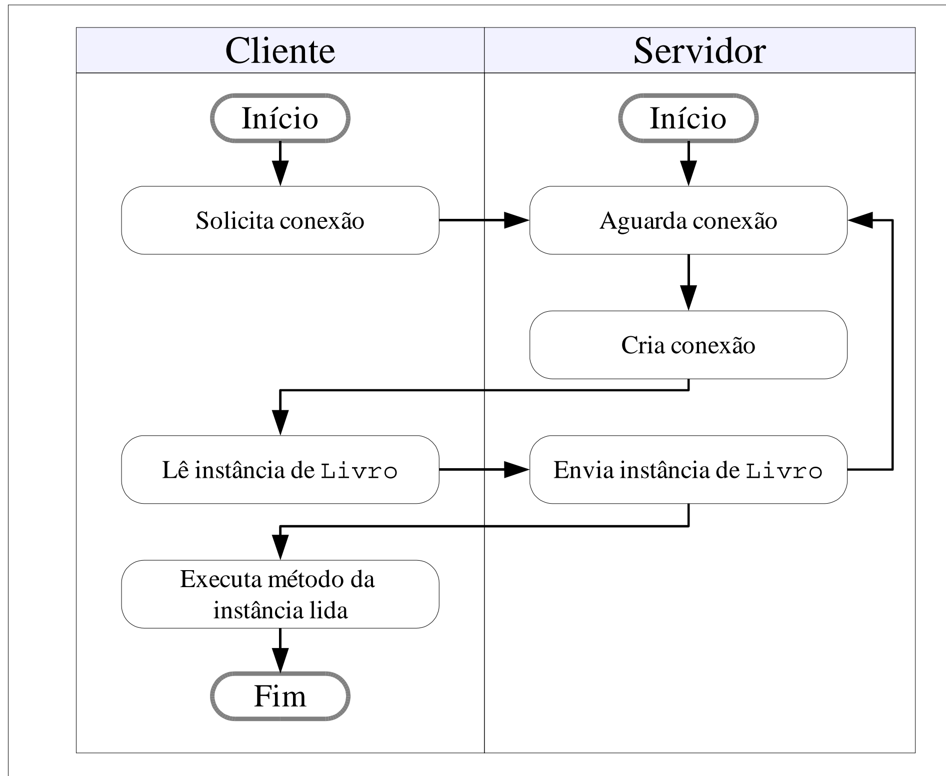

Figura 9: Protocolo de comunicação entre o cliente e o servidor de instâncias da classe `Livro`

O servidor de instâncias da classe `Livro` segue o mesmo padrão do servidor de strings invertidas (listagem 3): a classe contém um método `main` responsável por criar o *socket* do lado do servidor e atender a requisições dos clientes, que serão processadas pelo método `processaConexão`. Este método cria uma *stream* do tipo `ObjectOutputStream` para enviar uma das instâncias da classe `Livro` contidas na estrutura `coleção`.

O servidor de instâncias da classe `Livro` é implementado na classe `ServidorDeLivros`, mostrada na listagem 5.

Listagem 5: Servidor de instâncias da classe `Livro`

```
1  package cap312;
2  import java.io.IOException;
3  import java.io.ObjectOutputStream;
4  import java.net.BindException;
5  import java.net.ServerSocket;
6  import java.net.Socket;
7  import java.util.ArrayList;
8
9  /**
10  * Esta classe implementa um servidor simples que fica aguardando conexões de
```

```
11   * clientes. Este servidor, quando conectado, envia uma instância da classe
12   * Livro para o cliente e desconecta.
13   */
14  public class ServidorDeLivros
15    {
16    // Estrutura que armazenará uma coleção de livros (instâncias da classe Livro).
17    private static ArrayList<Livro> coleção;
18  
19    // Método que permite a execução da classe.
20    public static void main(String[] args)
21      {
22      // Criamos uma coleção de livros.
23      coleção = new ArrayList<Livro> ();
24      coleção.add(new Livro("Java 1.4 Game Programming",
25                            "Andrew Mulholland, Glen Murphy",2003));
26      coleção.add(new Livro("Developing Games In Java","David Bracken",2003));
27      coleção.add(new Livro("Java 2 Game Programming","Thomas Petchel",2001));
28      coleção.add(new Livro("Board And Table Games From Many Civilizations",
29                            "R.C.Bell",1969));
30      coleção.add(new Livro("A Gamut Of Games","Sid Sackson",1969));
31      try
32        {
33        // Criamos a instância de ServerSocket que responderá por solicitações
34        // à porta 12244.
35        ServerSocket servidor = new ServerSocket(12244);
36        // O servidor aguarda "para sempre" as conexões.
37        while(true)
38          {
39          // Quando uma conexão é feita,
40          Socket conexão = servidor.accept();
41          // ... o servidor a processa.
42          processaConexão(conexão);
43          }
44        }
45      // Pode ser que a porta já esteja em uso !
46      catch (BindException e)
47        {
48        System.out.println("Porta já em uso.");
49        }
50      // Pode ser que tenhamos um erro qualquer de entrada ou saída.
51      catch (IOException e)
52        {
53        System.out.println("Erro de entrada ou saída.");
54        }
55      }
56  
57    // Este método atende a uma conexão feita a este servidor.
58    private static void processaConexão(Socket conexão)
59      {
60      try
61        {
62        // Criamos uma stream para enviar instâncias de classes, usando a stream
63        // de saída associado à conexão.
64        ObjectOutputStream saída =
65          new ObjectOutputStream(conexão.getOutputStream());
66        // Escolhemos um livro aleatoriamente.
67        int qual = (int)(Math.random()*coleção.size());
68        saída.writeObject(coleção.get(qual));
69        // Ao terminar de atender a requisição, fechamos a stream de saída.
70        saída.close();
71        // Fechamos também a conexão.
```

```
72        conexão.close();
73        }
74      // Se houve algum erro de entrada ou saída...
75      catch (IOException e)
76        {
77        System.out.println("Erro atendendo a uma conexão !");
78        }
79      }
80    }
```

Para acessar este serviço, não podemos usar clientes como `telnet`: uma conexão feita via `telnet` para o servidor executando esta aplicação retornaria strings com alguns caracteres de controle (correspondente à serialização de uma instância da classe `Livro`) que torna os resultados da interação praticamente inutilizáveis. Precisamos de um cliente especializado em receber e processar instâncias da classe `Livro`.

O cliente para o servidor mostrado na listagem 5 também é simples e similar a outros clientes vistos anteriormente. A principal diferença é que ao invés de usar uma instância de `BufferedReader` para "ler" dados do servidor, iremos usar uma instância de `ObjectInputStream`. O cliente (classe `ClienteDeLivros`) é mostrado na figura 6.

Listagem 6: Cliente para o servidor de livros

```
1  package cap312;
2  import java.io.IOException;
3  import java.io.ObjectInputStream;
4  import java.net.Socket;
5  import java.net.UnknownHostException;
6
7
8  /**
9   * Esta classe implementa um cliente simples para o serviço de livros.
10  * Ele se conecta ao servidor (pela porta 12244) e recebe uma instância da classe
11  * Livro enviada pelo servidor.
12  */
13 public class ClienteDeLivros
14   {
15   public static void main(String[] args)
16     {
17     String servidor = "localhost";
18     // Tentamos fazer a conexão e ler uma linha...
19     try
20       {
21       Socket conexão = new Socket(servidor,12244);
22       // A partir do socket podemos obter um InputStream, a partir deste um
23       // ObjectInputStream.
24       ObjectInputStream dis = new ObjectInputStream(conexão.getInputStream());
25       Livro umLivro = (Livro)dis.readObject();
26       System.out.println("Livro obtido do servidor:");
27       umLivro.imprime();
28       // Fechamos a conexão.
29       dis.close();
30       conexão.close();
31       }
32     // Se houve problemas com o nome do host...
```

```
33      catch (UnknownHostException e)
34        {
35        System.out.println("O servidor não existe ou está fora do ar.");
36        }
37      // Se houve problemas genéricos de entrada e saída...
38      catch (IOException e)
39        {
40        System.out.println("Erro de entrada e saída.");
41        }
42      // Essa exceção poderia ocorrer se tivéssemos o cliente sendo executado em
43      // outro computador e a classe Livro não tivesse sido distribuída.
44      catch (ClassNotFoundException e)
45        {
46        System.out.println("A classe Livro não está presente no cliente.");
47        }
48      }
49    }
```

Existe uma outra diferença entre este cliente e outros: este deve, obrigatoriamente, ter um bloco `catch` para processar a exceção `ClassNotFoundException`, que será criada caso a classe cuja instância desejamos recuperar não existir do lado do cliente. Embora para finalidade de testes esta classe deva estar sempre presente, é importante lembrar que em aplicações mais complexas o servidor pode enviar uma instância de uma classe que não exista no cliente – quando a aplicação do lado cliente for distribuída para uso, devemos sempre lembrar de incluir todas as classes que podem ser serializadas para aquela aplicação.

5.3 Exemplo: Servidor e cliente de números aleatórios

Consideremos como um outro exemplo um servidor que envie para o cliente um número aleatório[5] entre zero e um, do tipo `double`. Embora qualquer computador que tenha a máquina virtual Java possa facilmente obter números aleatórios, podemos imaginar razões para fazer disto um serviço: uma razão comum é que o número deve ser coerente com os enviados para outros clientes (no caso de jogos). O protocolo de comunicação entre um cliente e um servidor de números aleatórios é mostrado na figura 10.

O protocolo de comunicação entre o cliente e servidor de números aleatórios é tão simples quanto o protocolo de comunicação entre cliente e servidor de `echo`, de strings invertidas ou de uma instância de uma classe, a diferença é que em vez de strings ou instâncias estaremos recebendo valores de um dos tipos nativos (no caso, `double`).

A implementação do servidor é semelhante à do servidor de strings invertidas ou de instâncias da classe `Livro`. O código-fonte do servidor é mostrado na listagem 7.

Listagem 7: Servidor de números aleatórios

[5] OK, são *pseudo-aleatórios*, mas para as finalidades deste documento vamos ignorar a diferença entre números realmente aleatórios e os gerados pelo método `Math.random`.

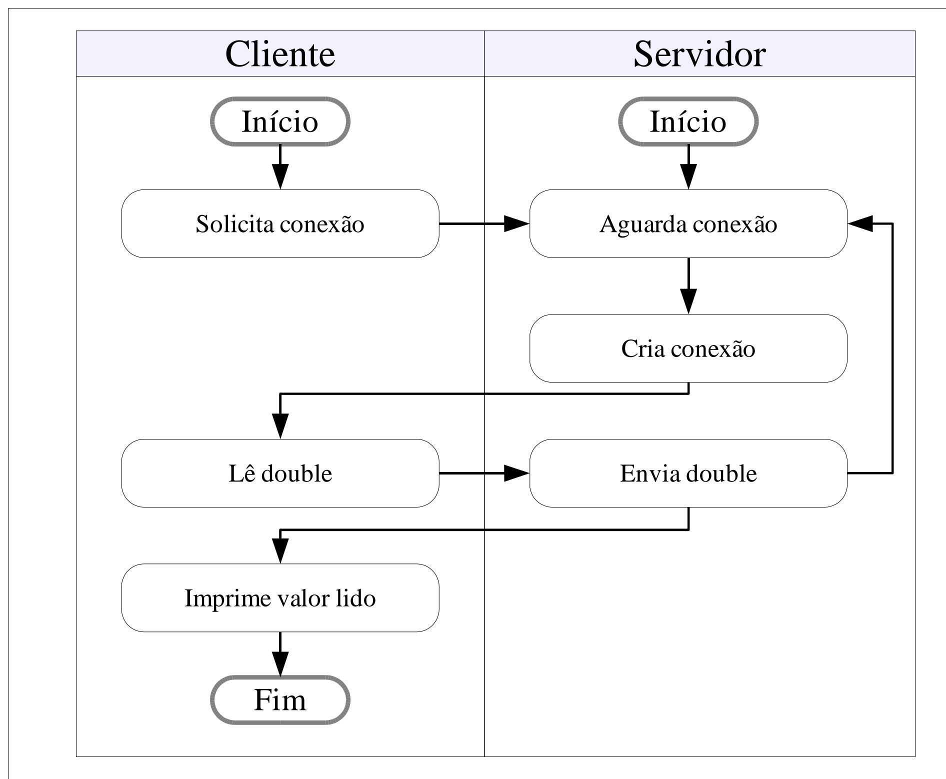

Figura 10: Protocolo de comunicação entre o cliente e o servidor de números aleatórios

```
1  package cap312;
2  import java.io.DataOutputStream;
3  import java.io.IOException;
4  import java.net.BindException;
5  import java.net.ServerSocket;
6  import java.net.Socket;
7
8  /**
9   * Esta classe implementa um servidor simples que fica aguardando conexões de
10  * clientes. Quando uma conexão é solicitada, o servidor envia para o cliente
11  * um número aleatório e fecha a conexão.
12  */
13 public class ServidorDeNumerosAleatorios
14   {
15   // Método que permite a execução da classe.
16   public static void main(String[] args)
17     {
18     ServerSocket servidor;
19     try
20       {
21       // Criamos a instância de ServerSocket que responderá por solicitações
22       // à porta 9999.
23       servidor = new ServerSocket(9999);
24       // O servidor aguarda "para sempre" as conexões.
25       while(true)
26         {
27         // Quando uma conexão é feita,
28         Socket conexão = servidor.accept();
29         // ... o servidor a processa.
30         processaConexão(conexão);
31         }
32       }
```

```
33      // Pode ser que a porta 9999 já esteja em uso !
34      catch (BindException e)
35        {
36        System.out.println("Porta já em uso.");
37        }
38      // Pode ser que tenhamos um erro qualquer de entrada ou saída.
39      catch (IOException e)
40        {
41        System.out.println("Erro de entrada ou saída.");
42        }
43      }
44
45    // Este método atende a uma conexão feita a este servidor.
46    private static void processaConexão(Socket conexão)
47      {
48      try
49        {
50        // Criamos uma stream para enviar valores nativos, usando a stream de saída
51        // associado à conexão.
52        DataOutputStream saída =
53          new DataOutputStream(conexão.getOutputStream());
54        double rand = Math.random();
55        saída.writeDouble(rand);
56        // Para demonstrar que realmente funciona, vamos imprimir um log.
57        System.out.println("Acabo de enviar o número "+rand+" para o cliente "+
58                           conexão.getRemoteSocketAddress());
59        // Ao terminar de atender a requisição, fechamos a stream de saída.
60        saída.close();
61        // Fechamos também a conexão.
62        conexão.close();
63        }
64      // Se houve algum erro de entrada ou saída...
65      catch (IOException e)
66        {
67        System.out.println("Erro atendendo a uma conexão !");
68        }
69      }
70    }
```

O código-fonte do servidor segue praticamente os mesmos passos do servidor de strings invertidas (listagem 3) exceto pela criação da classe que será responsável pelo envio dos dados do servidor para o cliente. Usaremos, para este servidor, a instância de `OutputStream` retornada pelo método `getInputStream` da classe `Socket` para criar uma instância da classe `DataOutputStream`. Para instâncias desta classe não é necessário enviar terminadores de linhas nem executar métodos que forcem o envio dos bytes.

Um ponto de interesse no código do servidor (listagem 7) é que quando um número aleatório é enviado para o cliente, uma mensagem é impressa no terminal onde o servidor está sendo executado. Isso facilita a depuração do servidor, caso seja necessário, e serve como ilustração da criação de arquivos de registro (*logs*) como é feito com servidores mais complexos.

O servidor de números aleatórios está pronto para receber conexões pela porta 9999, mas os clientes existentes não são capazes de receber bytes correspondentes a valores

do tipo `double`: se executarmos o comando `telnet localhost 9999` o mesmo retornará caracteres sem sentido. Precisamos de um cliente específico para este tipo de serviço.

Um cliente adequado para receber um único valor do tipo `double` de um servidor é mostrado na listagem 8. Novamente a diferença entre um cliente que processa strings, como `ClienteDeEcho` (listagem 2) é o tipo de classe que será criado usando as *streams* de entrada e saída obtidas da classe `Socket`. No caso do cliente, usamos uma instância de `DataInputStream` criada a partir do `InputStream` obtido pelo método `getInputStream` da classe `Socket`.

Listagem 8: Cliente de números aleatórios

```
package cap312;
import java.io.DataInputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;


/**
 * Esta classe implementa um cliente simples para o serviço de números aleatórios.
 * Ele se conecta ao servidor (pela porta 9999) e recebe um número aleatório
 * gerado no servidor.
 */
public class ClienteDeNumerosAleatorios
  {
  public static void main(String[] args)
    {
    String servidor = "localhost";
    // Tentamos fazer a conexão e ler uma linha...
    try
      {
      Socket conexão = new Socket(servidor,9999);
      // A partir do socket podemos obter um InputStream, a partir deste um
      // DataInputStream.
      DataInputStream dis = new DataInputStream(conexão.getInputStream());
      double valor = dis.readDouble();
      System.out.println("Li o valor "+valor+" do servidor "+servidor+".");
      // Fechamos a conexão.
      dis.close();
      conexão.close();
      }
    // Se houve problemas com o nome do host...
    catch (UnknownHostException e)
      {
      System.out.println("O servidor não existe ou está fora do ar.");
      }
    // Se houve problemas genéricos de entrada e saída...
    catch (IOException e)
      {
      System.out.println("Erro de entrada e saída.");
      }
    }
  }
```

5.4 Exemplo: Servidor simples de arquivos

Vamos considerar mais um exemplo de aplicação cliente-servidor, onde o tipo de dados e seu processamento será diferenciado durante a execução do protocolo de comunicação. Consideremos um servidor extremamente simples de arquivos, similar a um servidor de FTP mas com boa parte da funcionalidade (ex. acesso a diferentes diretórios, autenticação de usuários) removida. Durante a interação com o cliente, este servidor terá que se comunicar enviando e recebendo strings (comandos, listas de arquivos, etc.) que serão impressas no terminal e também enviando bytes que serão gravados em um arquivo pelo cliente.

O protocolo de comunicação entre o servidor e o cliente é simples: o servidor envia para o cliente a lista de arquivos, o cliente seleciona um, o servidor envia os bytes daquele arquivo para o cliente que armazena estes bytes em um arquivo local. Este protocolo é ilustrado na figura 11.

A implementação do servidor de arquivos é feita na classe `ServidorDeArquivos`, que é mostrada na listagem 9.

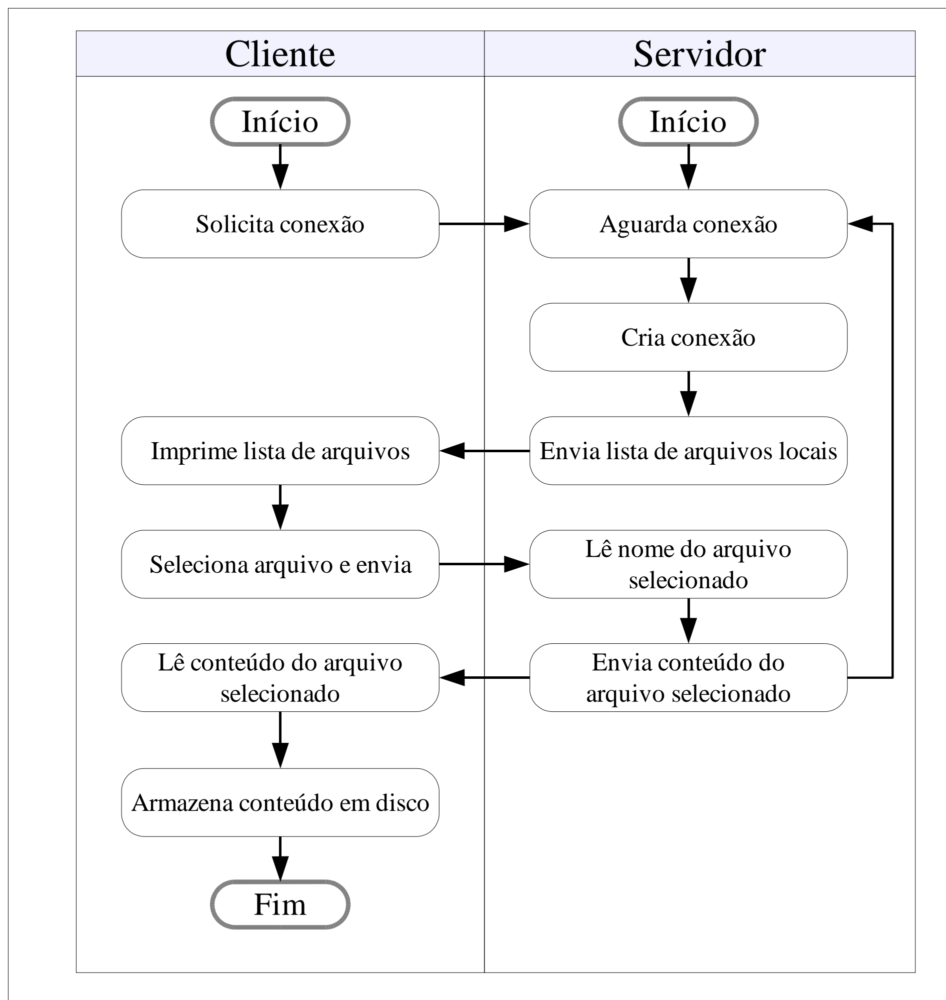

Figura 11: Protocolo de comunicação entre o cliente e o servidor de arquivos

Listagem 9: Servidor de arquivos

```
package cap312;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.BindException;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * Esta classe implementa um servidor simples que fica aguardando conexões de
 * clientes. Este servidor, quando conectado, envia uma lista de arquivos locais
 * para o cliente, aguarda a seleção de um nome de arquivo e envia o conteúdo
 * deste arquivo para o cliente.
 */
public class ServidorDeArquivos
  {
  // Método que permite a execução da classe.
  public static void main(String[] args)
    {
```

```
24      try
25        {
26        // Criamos a instância de ServerSocket que responderá por solicitações
27        // à porta 2048.
28        ServerSocket servidor = new ServerSocket(2048);
29        // O servidor aguarda "para sempre" as conexões.
30        while(true)
31          {
32          // Quando uma conexão é feita,
33          Socket conexão = servidor.accept();
34          // ... o servidor a processa.
35          processaConexão(conexão);
36          }
37        }
38      // Pode ser que a porta já esteja em uso !
39      catch (BindException e)
40        {
41        System.out.println("Porta já em uso.");
42        }
43      // Pode ser que tenhamos um erro qualquer de entrada ou saída.
44      catch (IOException e)
45        {
46        System.out.println("Erro de entrada ou saída.");
47        }
48      }
49
50    // Este método atende a uma conexão feita a este servidor.
51    private static void processaConexão(Socket conexão)
52      {
53      try
54        {
55        // Criamos uma stream para receber comandos do cliente - estes comandos
56        // serão somente strings.
57        BufferedReader entrada =
58          new BufferedReader(new InputStreamReader(conexão.getInputStream()));
59        // Criamos uma stream para enviar textos e dados para o cliente.
60        DataOutputStream saída =
61           new DataOutputStream(conexão.getOutputStream());
62        // Mandamos uma mensagem de boas vindas.
63        saída.writeUTF("Bem-vindo ao servidor de arquivos locais.");
64        saída.writeUTF("=========================================");
65        // Enviamos ao cliente a lista de arquivos locais.
66        File diretório = new File("."); // bom para Linux, será que funciona no Windows?
67        String[] arquivos = diretório.list(); // oops, inclui diretórios também !
68        for(int a=0;a<arquivos.length;a++)
69          {
70          saída.writeUTF(arquivos[a]);
71          }
72        // Aguardamos a seleção do usuário.
73        saída.writeUTF("-----------------------------------------");
74        saída.writeUTF("Selecione um dos arquivos acima.");
75        // Informamos ao cliente que terminamos de mandar texto.
76        saída.writeUTF("#####");
77        // Garantimos que as mensagens foram enviadas ao cliente.
78        saída.flush();
79        // Lemos o nome de arquivo selecionado pelo cliente.
80        String nomeSelecionado = entrada.readLine();
81        // Criamos uma representação do arquivo.
82        File selecionado = new File(nomeSelecionado);
83        // Enviamos uma mensagem esclarecedora para o cliente.
84        saída.writeUTF("Enviando arquivo "+nomeSelecionado+" ("+
```

```
85                      selecionado.length()+" bytes)");
86          saída.flush();
87          // Abrimos o arquivo localmente.
88          DataInputStream entradaLocal =
89            new DataInputStream(new FileInputStream(selecionado));
90          // Lemos todos os bytes do arquivo local, enviando-os para o cliente.
91          // Para maior eficiência, vamos ler e enviar os dados em blocos de 25 bytes.
92          byte[] arrayDeBytes = new byte[25];
93          while(true)
94            {
95            // Tentamos ler até 25 bytes do arquivo de entrada.
96            int resultado = entradaLocal.read(arrayDeBytes,0,25);
97            if (resultado == -1) break;
98            // Escrevemos somente os bytes lidos.
99            saída.write(arrayDeBytes,0,resultado);
100           }
101         // Ao terminar de ler o arquivo local, o fechamos.
102         entradaLocal.close();
103         // Ao terminar de atender a requisição, fechamos a stream de saída.
104         saída.close();
105         // Fechamos também a conexão.
106         conexão.close();
107         }
108       // Se houve algum erro de entrada ou saída...
109       catch (IOException e)
110         {
111         System.out.println("Erro atendendo a uma conexão !");
112         e.printStackTrace();
113         }
114       }
115   }
```

Alguns dos pontos de interesse na classe `ServidorDeArquivos` (listagem 9) são:

- O servidor segue o mesmo padrão de ter uma classe `main` e uma `processaConexão` – isto fará o desenvolvimento de servidores capazes de atender múltiplas requisições (veja seção 6) mais simples.
- Para receber os dados do cliente, o servidor usa uma instância de `BufferedReader`, pois assumimos que o cliente somente enviará strings para o servidor. Para enviar os dados para o cliente, usando a mesma conexão, usamos uma instância da classe `DataOutputStream`, que é capaz de enviar strings (usando o método `writeUTF`) e valores de tipos nativos, inclusive arrays de bytes, com o método `write`.
- Em determinado ponto da interação entre cliente e servidor, o servidor irá enviar um número desconhecido de linhas de texto para o cliente (os nomes de arquivos disponíveis no servidor). Para facilitar a interação com o cliente, que não sabe antecipadamente quantas linhas de texto serão enviadas, usamos um marcador para informar ao cliente que não existem mais linhas de texto a ser lidas: quando o cliente receber a string "#####" ele saberá que deve parar de ler linhas de texto do servidor.
- Quando o servidor for enviar os bytes do arquivo selecionado para o cliente, o procedimento será outro: o servidor enviará bytes enquanto existirem (em um laço

que será encerrado quando não for mais possível ler bytes do arquivo local) e o cliente monitorará estes bytes lidos para verificar o final da transmissão. Desta forma, demonstramos duas maneiras de verificar se todos os dados de uma sessão de comunicação foram enviados pelo servidor ou cliente: um usando marcadores ou *tags* que indicam final de transmissão e outro usando métodos da API de Java.

Vejamos agora o cliente para este servidor de arquivos. Não poderemos usar o cliente `telnet` pois em determinado momento da comunicação o servidor enviará bytes para o cliente, que devem ser armazenados em um arquivo e não mostrados na tela. Um cliente adequado é mostrado na classe `ClienteDeArquivos` (listagem 10).

Listagem 10: Cliente de arquivos

```
1  package cap312;
2  import java.io.BufferedWriter;
3  import java.io.DataInputStream;
4  import java.io.DataOutputStream;
5  import java.io.EOFException;
6  import java.io.FileOutputStream;
7  import java.io.IOException;
8  import java.io.OutputStreamWriter;
9  import java.net.Socket;
10 import java.net.UnknownHostException;
11
12 /**
13  * Esta classe implementa um cliente simples para o servidor de arquivos.
14  * A principal diferença entre este cliente e um cliente simples de texto
15  * (telnet, por exemplo) é que existe uma ordem no protocolo de comunicação
16  * entre cliente-servidor que pede que em determinado momento, ao invés de
17  * receber textos para mostrar no terminal, o cliente deve receber dados para
18  * armazenar em um arquivo.
19  */
20 public class ClienteDeArquivos
21   {
22   public static void main(String[] args)
23     {
24     String servidor = "localhost";
25     // Tentamos fazer a conexão...
26     try
27       {
28       Socket conexão = new Socket(servidor,2048);
29       // A partir do socket podemos obter um InputStream, a partir deste um
30       // InputStreamReader e partir deste, um BufferedReader.
31       DataInputStream entrada =
32         new DataInputStream(conexão.getInputStream());
33       // A partir do socket podemos obter um OutputStream, a partir deste um
34       // OutputStreamWriter e partir deste, um BufferedWriter.
35       BufferedWriter saída =
36         new BufferedWriter(
37            new OutputStreamWriter(conexão.getOutputStream()));
38       // Lemos a lista de arquivos do servidor. Sabemos que vamos ler um
39       // número desconhecido de linhas do servidor. Usamos um laço que
40       // lê linhas enquanto existirem dados a ser lidos. O servidor enviará
41       // a string ##### quando não houverem mais dados.
42       while(true)
43         {
44         String linha = entrada.readUTF();
```

```
45          if(linha.equals("#####")) break; // código do servidor: acabaram as linhas.
46          System.out.println(linha);
47          }
48        // Lemos o nome do arquivo que desejamos do teclado e o
49        // enviamos para o servidor.
50        String arquivo = Keyboard.readString();
51        saída.write(arquivo);
52        saída.newLine();
53        saída.flush();
54        // O servidor envia para o cliente mais uma informação como texto.
55        String linha = entrada.readUTF();
56        System.out.println(linha);
57        // Abrimos o arquivo local. Vamos dar um nome diferente para ele só
58        // para não complicar a vida de quem estiver rodando cliente e servidor
59        // no mesmo computador.
60        DataOutputStream saídaParaArquivo =
61          new DataOutputStream(new FileOutputStream("_"+arquivo));
62        // A partir deste momento o servidor vai nos enviar bytes. Lemos todos os
63        // bytes enviados do servidor, gravando-os em um arquivo local.
64        // Para maior eficiência, vamos ler os dados em blocos de 25 bytes.
65        byte[] array = new byte[25];
66        while(true)
67          {
68          int resultado = entrada.read(array,0,25);
69          if (resultado == -1) break;
70          // Escrevemos somente os bytes lidos.
71          saídaParaArquivo.write(array,0,resultado);
72          }
73        // Fechamos as streams e a conexão.
74        saída.close();
75        entrada.close();
76        conexão.close();
77        }
78      // Se houve problemas com o nome do host...
79      catch (UnknownHostException e)
80        {
81        System.out.println("O servidor não existe ou está fora do ar.");
82        }
83      // Se houve problemas genéricos de entrada e saída...
84      catch (EOFException e)
85        {
86        System.out.println("Erro de EOF.");
87        }
88      // Se houve problemas genéricos de entrada e saída...
89      catch (IOException e)
90        {
91        System.out.println("Erro de entrada e saída.");
92        }
93      }
94  }
```

O cliente mostrado na listagem 10 tem, como pontos de interesse, o esquema de leitura de várias linhas do servidor, que continua lendo linhas do servidor até que a linha `"#####"` seja enviada; e o trecho que lê bytes do servidor (em blocos de 25 bytes) e armazena estes bytes em um arquivo local.

6 Servidores para múltiplas requisições simultâneas

Até agora os servidores implementados (exceto pelos servidores já existentes como `echo` e `daytime`) são capazes de atender uma única requisição simultânea dos clientes - se um cliente já estivesse acessando o serviço e outros clientes também tentassem o acesso, estes outros clientes teriam que aguardar a conexão com o primeiro cliente ser encerrada.

Felizmente, os servidores demonstrados são muito simples, enviando dados para o cliente e imediatamente encerrando a conexão, permitindo a conexão por parte de outro cliente logo em seguida. caso os servidores, por alguma razão, demorassem demais para atender os clientes, outros clientes experimentariam uma demora no acesso ou mesmo a negação do serviço. Podemos assumir que servidores comuns (como, por exemplo, `http`) são capazes de atender múltiplas requisições simultaneamente, caso contrário um usuário teria que esperar outro terminar a sua conexão em uma espécie de fila.

Nesta seção veremos por que é que alguns servidores devem estar preparados para atender múltiplas requisições simultaneamente e como podemos preparar servidores para esta tarefa.

6.1 O problema de múltiplas requisições simultâneas

Consideremos uma pequena alteração no protocolo de comunicação entre o cliente e o servidor de números aleatórios: o cliente agora envia para o servidor o número de valores que este deseja receber. O servidor, por sua vez, ao invés de enviar somente um número aleatório, envia a quantidade de números solicitada pelo cliente.

O protocolo de comunicação para esta versão de cliente e servidor é mostrado na figura 12.

Embora o diagrama da figura 12 pareça mais complicado do que os outros, a única diferença fundamental é que existem dois laços, do lado do cliente e do lado do servidor, que garantem que o mesmo número de valores será requisitado e lido.

Considerando estas modificações, a classe servidora de números aleatórios foi reescrita como mostra a listagem 11. Esta classe segue o mesmo padrão de dois métodos, um `main` e um responsável por processar uma requisição.

Listagem 11: Segunda versão do servidor de números aleatórios

```
1 package cap312;
2 import java.io.DataInputStream;
3 import java.io.DataOutputStream;
4 import java.io.IOException;
```

```
5  import java.net.BindException;
6  import java.net.ServerSocket;
7  import java.net.Socket;
8
9  /**
10  * Esta classe implementa um servidor simples que fica aguardando conexões de
11  * clientes. Quando uma conexão é solicitada, o servidor lê do cliente a
12  * quantidade de números aleatórios desejada e envia para o cliente
13  * estes números aleatórios, terminando então a conexão.
14  */
15 public class SegundoServidorDeNumerosAleatorios
16   {
17   // Método que permite a execução da classe.
18   public static void main(String[] args)
19     {
20     ServerSocket servidor;
21     try
22       {
23       // Criamos a instância de ServerSocket que responderá por solicitações
```

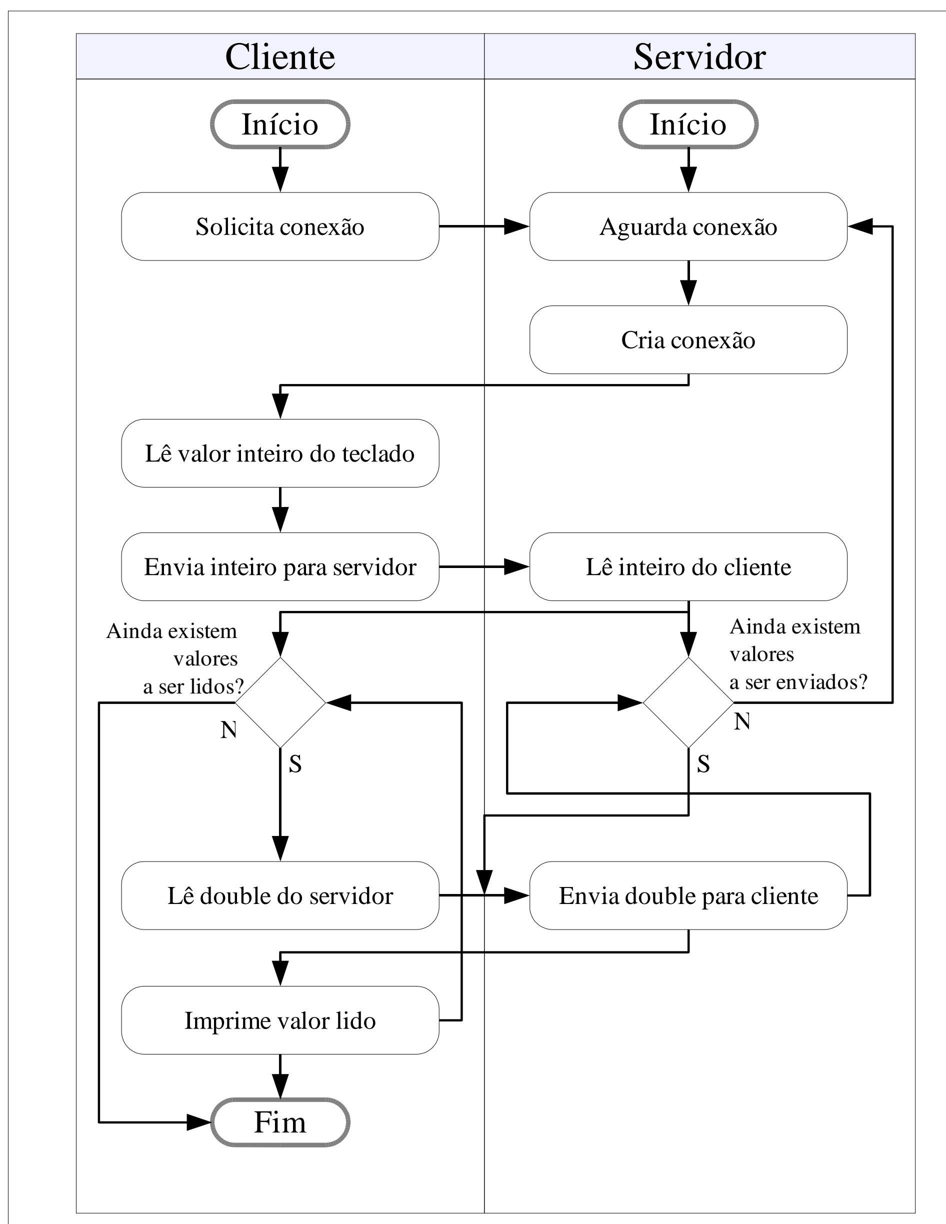

Figura 12: Protocolo de comunicação entre o cliente e o servidor de números aleatórios (segunda versão)

```
24      // à porta 9999.
25      servidor = new ServerSocket(9999);
26      // O servidor aguarda "para sempre" as conexões.
27      while(true)
28        {
29        // Quando uma conexão é feita,
30        Socket conexão = servidor.accept();
31        // ... o servidor a processa.
32        processaConexão(conexão);
33        }
34      }
35    // Pode ser que a porta 9999 já esteja em uso !
36    catch (BindException e)
37      {
38      System.out.println("Porta já em uso.");
39      }
40    // Pode ser que tenhamos um erro qualquer de entrada ou saída.
41    catch (IOException e)
42      {
43      System.out.println("Erro de entrada ou saída.");
44      }
45    }
46
47  // Este método atende a uma conexão feita a este servidor.
48  private static void processaConexão(Socket conexão)
49    {
50    try
51      {
52      // Criamos uma stream para receber valores nativos, usando a stream de entrada
53      // associado à conexão.
54      DataInputStream entrada =
55        new DataInputStream(conexão.getInputStream());
56      // Criamos uma stream para enviar valores nativos, usando a stream de saída
57      // associado à conexão.
58      DataOutputStream saída =
59        new DataOutputStream(conexão.getOutputStream());
60      // Lemos do cliente quantos números ele deseja.
61      int quantidade = entrada.readInt();
62      // Enviamos para o cliente os números.
63      for(int q=0;q<quantidade;q++)
64        {
65        double rand = Math.random();
66        saída.writeDouble(rand);
67        // Para demonstrar que realmente funciona, vamos imprimir um log.
68        System.out.println("Acabo de enviar o número "+rand+" para o cliente "+
69                           conexão.getRemoteSocketAddress());
70        }
71      // Ao terminar de atender a requisição, fechamos as streams.
72      entrada.close();
73      saída.close();
74      // Fechamos também a conexão.
75      conexão.close();
76      }
77    // Se houve algum erro de entrada ou saída...
78    catch (IOException e)
79      {
80      System.out.println("Erro atendendo a uma conexão !");
81      }
82    }
83  }
```

O cliente para este serviço também precisa ser reescrito. O cliente para a segunda versão do servidor de números aleatórios é mostrado na listagem 12.

Listagem 12: Segunda versão do cliente de números aleatórios

```
package cap312;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;

public class SegundoClienteDeNumerosAleatorios
  {
  public static void main(String[] args)
    {
    String servidor = "localhost";
    // Tentamos fazer a conexão e ler uma linha...
    try
      {
      Socket conexão = new Socket(servidor,9999);
      // A partir do socket podemos obter um InputStream, a partir deste um
      // DataInputStream.
      DataInputStream dis =
        new DataInputStream(conexão.getInputStream());
      // A partir do socket podemos obter um OutputStream, a partir deste um
      // DataOutputStream.
      DataOutputStream dos =
          new DataOutputStream(conexão.getOutputStream());
      // Quantos números aleatórios vamos querer ?
      System.out.print("Quantos números:");
      int quantos = Keyboard.readInt();
      // Enviamos esta informação para o servidor.
      dos.writeInt(quantos);
      // Lemos os números desejados do servidor.
      for(int q=0;q<quantos;q++)
        {
        double valor = dis.readDouble();
        System.out.println("O "+(q+1)+"º valor é "+valor+".");
        }
      // Fechamos a conexão.
      dis.close();
      conexão.close();
      }
    // Se houve problemas com o nome do host...
    catch (UnknownHostException e)
      {
      System.out.println("O servidor não existe ou está fora do ar.");
      }
    // Se houve problemas genéricos de entrada e saída...
    catch (IOException e)
      {
      System.out.println("Erro de entrada e saída.");
      }
    }
  }
```

O cliente modificado mostrado na listagem 12 também é bem simples, contendo somente um laço adicional quando comparado com outros clientes.

O cliente e servidor de múltiplos números aleatórios funcionam corretamente, mas podem causar um sério problema. Imagine que este seja um serviço concorrido, com várias pessoas a qualquer momento solicitando números aleatórios. Se um usuário solicitar uma quantidade exageradamente grande de números aleatórios, outro cliente estará impossibilitado de usar o serviço, pois o método `processaConexão` do servidor deve terminar o seu processamento para retornar o controle ao método `main`, que então poderá atender outra conexão quando o método `accept` for executado.

Temos algumas possibilidades para a solução deste problema em potencial:

- Fazer clientes aguardarem sua vez sem maiores informações - não existe maneira de informar aos clientes quanto tempo eles terão que esperar na fila para atendimento (analogia: telefone ocupado).
- Executar mais de um servidor simultaneamente, em portas diferentes - se a primeira tentativa de conexão fracassar, o cliente deverá descobrir, por conta própria, qual das portas que oferecem o serviço está desocupada (analogia: vários telefones para a mesma pessoa, alguns ocupados).
- Limitar a quantidade de números a serem servidos para que a conexão seja liberada o mais rápido possível para outros clientes.

Evidentemente estas opções são muito restritivas. Uma outra alternativa, mais interessante, seria fazer uso de linhas de execução ou *threads*. Basicamente *threads* permitem que aplicações executem mais de uma tarefa de forma aparentemente simultânea. Continuamos com uma aplicação que tem um único método `main`, mas durante a execução deste método ele cria várias instâncias de classes que podem ser executadas concorrentemente. Para escrever um servidor capaz de tratar várias requisições de forma aparentemente simultânea, podemos usar os conceitos de *threads* e as classes em Java que os implementam.

6.2 Linhas de execução (*threads*)

Para melhor compreender linhas de execução, consideremos um exemplo simples não relacionado com programação cliente-servidor. Imaginemos uma simulação que envolva alguns objetos que poderiam se comportar independentemente, como por exemplo, uma simulação de corrida onde cada carro tem uma velocidade e é independente de outros carros. Vamos criar uma classe `CarroDeCorrida` para representar um carro de corrida para esta simulação. Esta classe é mostrada na listagem 13.

Listagem 13: Classe que representa um carro de corrida para simulação.

```
1 package cap312;
2 /**
```

```
3    * Esta classe representa um carro da Corrida Maluca para uma simulação.
4    */
5   public class CarroDeCorrida
6     {
7     private String nome;
8     private int distância;
9     private int velocidade;
10
11    /**
12     * O construtor da classe inicializa o nome do carro e a velocidade do mesmo.
13     */
14    public CarroDeCorrida(String n,int vel)
15      {
16      nome = n;
17      distância = 0;
18      velocidade = vel;
19      }
20
21    /**
22     * Este método imprime os passos percorridos pelo carro.
23     */
24    public void executa()
25      {
26      while(distância <= 1200)
27        {
28        System.out.println(nome+" rodou "+distância+" km.");
29        distância += velocidade;
30        // Pausa o processamento com um cálculo inútil.
31        for(int sleep=0;sleep<1000000;sleep++)
32          {
33          double x = Math.sqrt(Math.sqrt(Math.sqrt(sleep)));
34          }
35        }
36      }
37
38    }
```

A classe `CarroDeCorrida` (listagem 13) é realmente simples, tendo como métodos somente o construtor e um método `executa` que executará a simulação para a instância da classe. A simulação consiste em mudar a distância percorrida pelo carro, imprimir na tela e fazer uma pequena pausa (artificialmente criada por um laço que executa uma operação matemática).

A simulação de uma corrida poderia ser feita através de uma classe com várias instâncias da classe `CarroDeCorrida`. A classe `SimulacaoSemThreads`, na listagem 14, cria uma simulação simples.

Listagem 14: Simulação usando instâncias de `CarroDeCorrida`.

```
1   package cap312;
2   /**
3    * Esta classe faz uma simulação de corrida usando instâncias da classe
4    * CarroDeCorrida.
5    */
6   public class SimulacaoSemThreads
7     {
```

```
8  // Este método permite a execução da classe.
9  public static void main(String[] args)
10   {
11   // Criamos instâncias da classe CarroDeCorrida.
12   CarroDeCorrida penélope =
13     new CarroDeCorrida("Penélope Charmosa",60);
14   CarroDeCorrida dick =
15     new CarroDeCorrida("Dick Vigarista",100);
16   CarroDeCorrida quadrilha =
17     new CarroDeCorrida("Quadrilha da Morte",120);
18   // Criados os carros, vamos executar as simulações.
19   penélope.executa();
20   dick.executa();
21   quadrilha.executa();
22   }
23 }
```

A classe `SimulacaoSemThreads` também é relativamente simples: criamos as instâncias de `CarroDeCorrida` e executamos os seus métodos `executa`. O problema, para uma simulação, é que os três métodos são executados de forma dependente um do outro: só podemos executar o método `executa` da instância `quadrilha` **depois** de ter terminado **completamente** a execução dos métodos das instâncias `penélope` e `dick`. Para uma simulação mais realista, isso não seria aceitável – os métodos deveriam ser executados em paralelo. O resultado da execução da classe `SimulacaoSemThreads` mostrará a simulação para a instância `penélope`, seguida da simulação para a instância `dick` e finalmente a simulação para a instância `quadrilha`.

A forma mais simples para fazer com que as instâncias possam ter um de seus métodos executados em paralelo com outros é fazer a classe que contém o método herdar da classe `Thread` e implementar o método `run`, que será o método a ser executado em paralelo. Isto é demonstrado na classe `CarroDeCorridaIndependente`, mostrada na listagem 15.

Listagem 15: Classe que representa um carro de corrida para simulação (herdando de `Thread`).

```
1  package cap312;
2  /**
3   * Esta classe representa um carro da Corrida Maluca para uma simulação.
4   * Esta versão da classe herda da classe Thread, então o método run de
5   * instâncias desta classe poderá ser executado independentemente.
6   */
7  public class CarroDeCorridaIndependente extends Thread
8    {
9    private String nome;
10   private int distância;
11   private int velocidade;
12
13   /**
14    * O construtor da classe inicializa o nome do carro e a velocidade do mesmo.
15    */
16   public CarroDeCorridaIndependente(String n,int vel)
17     {
18     nome = n;
```

```
19      distância = 0;
20      velocidade = vel;
21      }
22
23   /**
24    * Este método imprime a distância percorrida até agora.
25    */
26    public void run()
27      {
28      while(distância <= 1200)
29        {
30        System.out.println(nome+" rodou "+distância+" km.");
31        distância += velocidade;
32        // Pausa o processamento com um cálculo inútil.
33        for(int sleep=0;sleep<1000000;sleep++)
34          {
35          double x = Math.sqrt(Math.sqrt(Math.sqrt(sleep)));
36          }
37        }
38      }
39
40    }
```

É importante observar que a assinatura do método `run` deve obrigatoriamente ser `public void` – este método não deve receber argumentos nem relançar exceções.

A classe `SimulacaoComThreads` (listagem 16) demonstra a simulação de instâncias da classe `CarroDeCorridaIndependente`.

Listagem 16: Simulação usando instâncias de `CarroDeCorridaIndependente`.

```
1  package cap312;
2  /**
3   * Esta classe faz uma simulação de corrida usando instâncias da classe
4   * CarroDeCorridaIndependente.
5   */
6  public class SimulacaoComThreads
7    {
8    // Este método permite a execução da classe.
9    public static void main(String[] args)
10     {
11     // Criamos instâncias da classe CarroDeCorrida.
12     CarroDeCorridaIndependente penélope =
13       new CarroDeCorridaIndependente("Penélope Charmosa",60);
14     CarroDeCorridaIndependente dick =
15       new CarroDeCorridaIndependente("Dick Vigarista",100);
16     CarroDeCorridaIndependente quadrilha =
17       new CarroDeCorridaIndependente("Quadrilha da Morte",120);
18     // Criados os carros, vamos executar as simulações.
19     penélope.start();
20     dick.start();
21     quadrilha.start();
22     }
23   }
```

Quando a classe `SimulacaoComThreads` (listagem 16) for executada, veremos que as impressões dos nomes dos carros e respectivas posições aparecerão intercaladas e em

ordem aparentemente imprevisível, pois estão sendo executadas concorrentemente. Se o trecho de código que simula uma pausa no processamento for reescrito para a pausa ser maior, este efeito de ordem imprevisível será aumentado. Se o trecho for eliminado, quase sempre as simulações serão executadas na mesma ordem em que os métodos `run` forem chamados, embora não exista garantia que isso sempre vai acontecer. Mesmo se o tempo de pausa fosse simulado de forma aleatória para cada execução, o resultado da execução da classe `SimulacaoSemThreads` seria o mesmo (na ordem de chamada do método `executa`) enquanto que seguramente os resultados seriam diferentes para cada execução da classe `SimulacaoComThreads`.

É importante notar que declaramos o método `run` na classe `CarroDeCorridaIndependente` (listagem 15) como sendo o ponto de entrada do processamento a ser feito em paralelo, mas devemos executar o método `start` (veja listagem 16) para dar início à execução do método `run`.

O que possibilita a execução de métodos concorrentemente é o fato do método `start` inicializar a execução da *thread* e retornar imediatamente. No exemplo da classe `SimulacaoComThreads` (listagem 16), logo depois que a *thread* para processamento dos dados da instância `penélope` é inicializada, o controle retorna para o método `main`, que executa o método `start` para a instância `dick`, retornando novamente o controle para o método `main`, que finalmente executa o método `start` para a instância `quadrilha`. Enquanto o método `main` continua sendo executado, os métodos `run` das instâncias da classe `CarroDeCorridaIndependente` iam são executados concorrentemente.

6.3 Exemplo: Servidor de números aleatórios (para requisições simultâneas)

Vamos reescrever o servidor de números aleatórios (segunda versão) para usar linhas de execução, assim fazendo com que o servidor não bloqueie usuários que tentem se conectar enquanto uma sessão estiver em andamento. Usando o conceito de linhas de execução, o servidor, ao invés de executar um método de sua própria classe (`processaConexão`) quando tiver que atender um cliente, irá criar uma classe que herda de `Thread` cujo método `run` executa exatamente o que o método `processaConexão` faria.

Os passos para transformar uma classe que implementa um servidor sem múltiplas linhas de execução para classes que implementam servidores com múltiplas linhas de execução são, então:

1. Criar uma classe que implementa o atendimento a uma única conexão. Esta classe deverá herdar da classe `Thread`, ter um construtor (para possibilitar o armazenamento da instância de `Socket` relacionada com a conexão que deve ser atendida por uma instância desta mesma classe) e o método `run`, que deverá ser respon-

sável pelo processamento do atendimento da conexão. Em geral, o conteúdo do método `run` deve ser o mesmo dos métodos `processaConexão` usados em outros exemplos.

2. Reescrever a classe do servidor para que a mesma, ao receber uma conexão (através do método `accept` da classe `ServerSocket`, crie uma nova instância da classe responsável pelo atendimento a uma única conexão e execute o método `start` desta instância.

Usando estes passos, reescreveremos o servidor de números aleatórios. Primeiro, escrevemos a classe `ServicoDeNumerosAleatorios` cujas instâncias serão responsáveis por processar uma única conexão. A classe `ServicoDeNumerosAleatorios` é mostrada na listagem 17 e contém somente o construtor e o método `run` que executa todo o atendimento à conexão.

Listagem 17: Classe que implementa o serviço de números aleatórios

```
1  package cap312;
2  import java.io.DataInputStream;
3  import java.io.DataOutputStream;
4  import java.io.IOException;
5  import java.net.Socket;
6
7  /**
8   * Esta classe representa o processamento que deve ser feito quando uma
9   * única requisição for feita a um servidor. A classe herda de Thread para que
10  * várias instâncias dela possam ser executadas concorrentemente.
11  */
12 public class ServicoDeNumerosAleatorios extends Thread
13   {
14   // Precisamos armazenar o socket correspondente à conexão.
15   private Socket conexão = null;
16
17  /**
18   * O construtor desta classe recebe como argumento uma instância de Socket
19   * e o armazena em um campo da classe.
20   */
21   public ServicoDeNumerosAleatorios(Socket conn)
22     {
23     conexão = conn;
24     }
25
26  /**
27   * Este método executa a rotina de atendimento a uma conexão com o servidor.
28   */
29   public void run()
30     {
31     try
32       {
33       // Criamos uma stream para receber valores nativos, usando a stream de entrada
34       // associado à conexão.
35       DataInputStream entrada =
36         new DataInputStream(conexão.getInputStream());
37       // Criamos uma stream para enviar valores nativos, usando a stream de saída
38       // associado à conexão.
```

```
39        DataOutputStream saída =
40          new DataOutputStream(conexão.getOutputStream());
41        // Lemos do cliente quantos números ele deseja.
42        int quantidade = entrada.readInt();
43        // Enviamos para o cliente os números.
44        for(int q=0;q<quantidade;q++)
45          {
46          double rand = Math.random();
47          saída.writeDouble(rand);
48          // Para demonstrar que realmente funciona, vamos imprimir um log.
49          System.out.println("Acabo de enviar o número "+rand+" para o cliente "+
50                             conexão.getRemoteSocketAddress());
51          }
52        // Ao terminar de atender a requisição, fechamos as streams.
53        entrada.close();
54        saída.close();
55        // Fechamos também a conexão.
56        conexão.close();
57        }
58      // Se houve algum erro de entrada ou saída...
59      catch (IOException e)
60        {
61        System.out.println("Erro atendendo a uma conexão !");
62        }
63      }
64
65    }
```

É necessário armazenar na classe `ServicoDeNumerosAleatorios` (listagem 17) uma instância da classe `Socket` que corresponde ao *socket* que foi criado para comunicação com o cliente, pois não poderíamos passar esta instância como argumento para o método `run` – o mesmo deve ser sempre executado sem argumentos.

A classe que representa o servidor também foi modificada de acordo com os passos indicados. A classe `TerceiroServidorDeNumerosAleatorios` contém agora somente o método `main` que cria novas instâncias de `ServicoDeNumerosAleatorios` quando novas conexões são estabelecidas, e que executa o método `start` destas instâncias, retornando o controle ao método `main` imediatamente. A classe `TerceiroServidorDeNumerosAleatorios` é mostrada na listagem 18.

Listagem 18: Terceira versão do servidor de números aleatórios

```
1  package cap312;
2  import java.io.IOException;
3  import java.net.BindException;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6
7  /**
8   * Esta classe implementa um servidor simples que fica aguardando conexões de
9   * clientes. Quando uma conexão é solicitada, o servidor lê do cliente a
10  * quantidade de números aleatórios desejada e envia para o cliente
11  * estes números aleatórios, terminando então a conexão.
12  */
13 public class TerceiroServidorDeNumerosAleatorios
```

```
14   {
15   // Método que permite a execução da classe.
16   public static void main(String[] args)
17     {
18     ServerSocket servidor;
19     try
20       {
21       // Criamos a instância de ServerSocket que responderá por solicitações
22       // à porta 9999.
23       servidor = new ServerSocket(9999);
24       // O servidor aguarda "para sempre" as conexões.
25       while(true)
26         {
27         // Quando uma conexão é feita,
28         Socket conexão = servidor.accept();
29         // ... criamos um serviço para atender a esta conexão...
30         ServicoDeNumerosAleatorios s = new ServicoDeNumerosAleatorios(conexão);
31         // ... e executamos o serviço (que será executado independentemente).
32         s.start();
33         }
34       }
35     // Pode ser que a porta 9999 já esteja em uso !
36     catch (BindException e)
37       {
38       System.out.println("Porta já em uso.");
39       }
40     // Pode ser que tenhamos um erro qualquer de entrada ou saída.
41     catch (IOException e)
42       {
43       System.out.println("Erro de entrada ou saída.");
44       }
45     }
46   }
```

Quando o método `main` da classe `TerceiroServidorDeNumerosAleatorios` (listagem 18) for executado, aguardará uma conexão do cliente. Quando esta conexão for criada (ou seja, quando o método `accept` for executado e retornar uma instância de `Socket`, uma instância da classe `ServicoDeNumerosAleatorios` será criada para tratar desta conexão, e o seu método `start` será executado. Imediatamente (isto é, sem aguardar o final da execução do método `start` ou `run`) o controle passará de volta para o método `main`, que estará liberado para aceitar novas conexões.

O cliente desta classe não precisa de modificações, como cliente podemos usar a classe `SegundoClienteDeNumerosAleatorios` (listagem 12).

7 Aplicações baseadas em um servidor e vários clientes

Veremos agora como funciona uma aplicação onde existe um servidor e vários clientes conectados simultaneamente, mas de forma que cada cliente tenha a sua vez de interagir com o servidor (os clientes não são independentes). Este exemplo de interação ilustra o papel de um servidor como mediador em um jogo, por exemplo.

7.1 Exemplo: Servidor de jogo-da-velha

Vejamos um exemplo de aplicação cliente-servidor com um servidor e dois clientes sincronizados (isto é, que tem a sua vez para interagir com o servidor determinadas e em ordem). O jogo-da-velha é um bom exemplo: dois jogadores, através dos seus clientes, se conectam com o servidor, e um depois do outro, enviam comandos para este. Além de receber e processar os movimentos feitos pelos clientes, o servidor é responsável por analisar estes movimentos (isto é, verificar se eles são válidos), informar os jogadores dos movimentos de ambos e determinar quando o jogo tiver um vencedor (terminando as conexões com os clientes).

O algoritmo para controlar as jogadas e verificar se houve vencedores no jogo-da-velha é simples. Consideremos que o jogo é implementado como um tabuleiro de 3x3 posições. Internamente este tabuleiro será um array bidimensional de caracteres, onde cada posição (caracter) poderá ser igual a um `X` ou `O` se estiver ocupado ou espaço se estiver livre. Para que seja mais fácil para o jogador indicar a posição onde quer jogar, as linhas do tabuleiro serão indicadas por **A**, **B** e **C** e as colunas por **1**, **2** e **3**. A figura 13 mostra o tabuleiro com os indicadores de linhas e colunas.

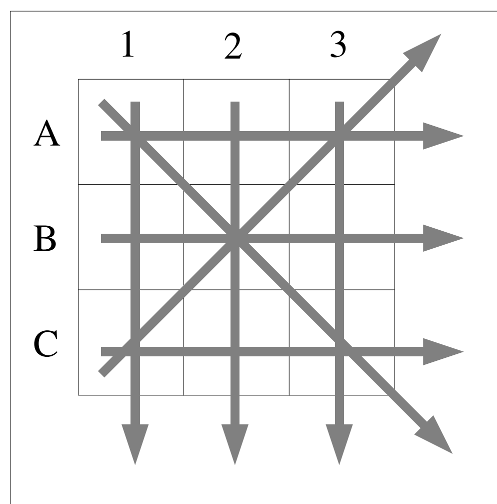

Figura 13: Posições para jogo e linhas vencedoras no jogo-da-velha

O algoritmo que verifica se houve um vencedor para o jogo também é simples. Existem oito combinações de posições que se forem ocupadas pelo mesmo jogador determinam a vitória deste jogador: se as três linhas, três colunas ou duas diagonais do tabuleiro forem totalmente preenchidas pelo mesmo jogador, este ganhou o jogo. As setas mostradas na figura 13 indicam estas combinações.

Para que um jogo possa ser executado, são necessários dois jogadores (dois clientes). Conforme mencionado anteriormente, os clientes não se conectam e enviam/recebem informações do servidor simultaneamente (usando linhas de execução): a comunicação é feita por vezes, uma vez sendo a de um cliente/jogador e a vez seguinte sendo do outro.

O protocolo do jogo é simples: o servidor aguarda a conexão de dois clientes, e alterna o recebimento de comandos (posições para jogo) de um cliente para o outro. Depois da jogada de cada cliente, o servidor verifica se a jogada é válida, modificando o tabuleiro do jogo se o for. Se houver vencedor, ou se não houver mais posições onde o jogador da vez possa jogar, o resultado é apresentado e a conexão com os clientes é encerrada. O protocolo de interação entre os clientes e o servidor é mostrado na figura 14.

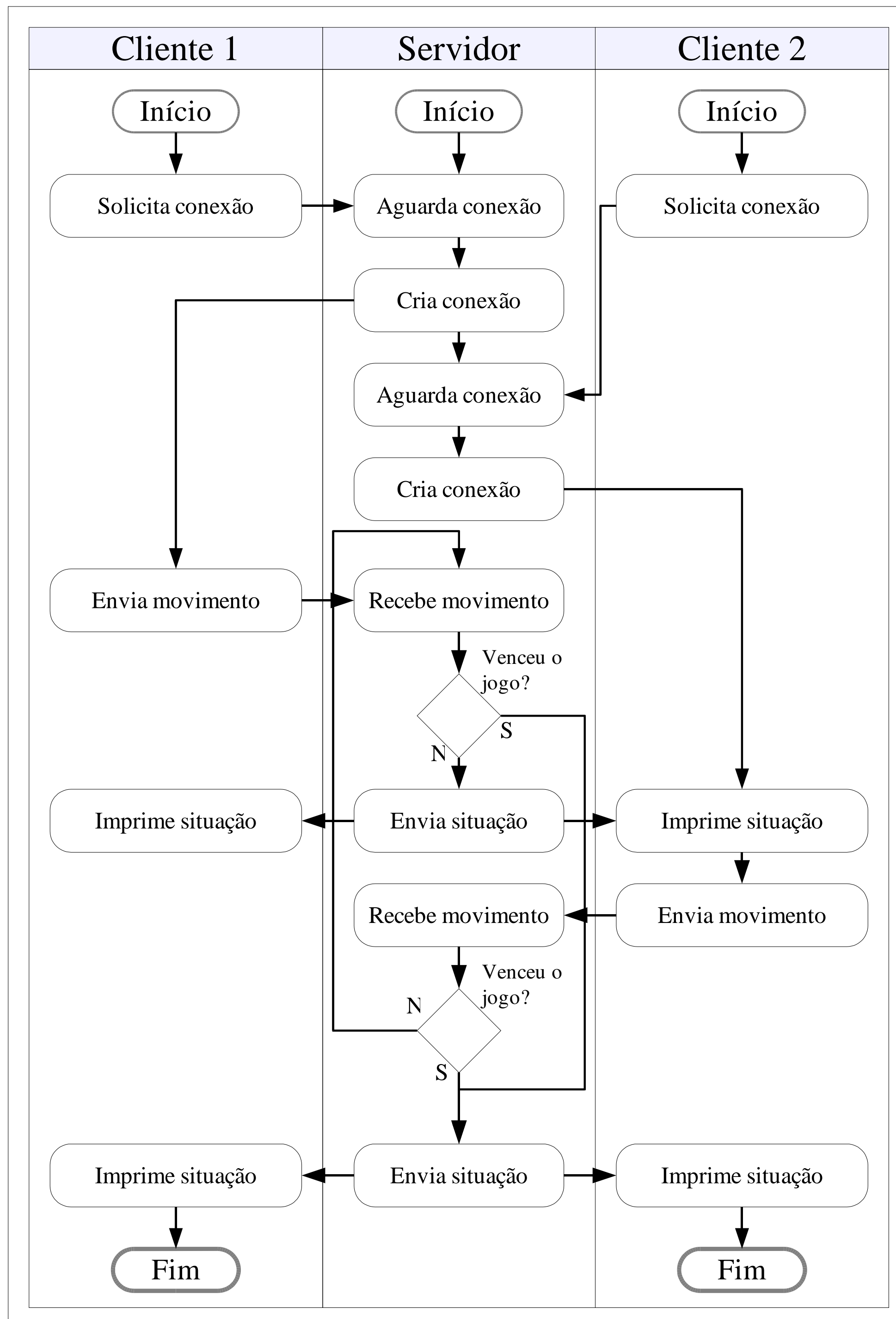

Figura 14: Protocolo de comunicação entre o servidor e os clientes de jogo-da-velha

A implementação de uma classe que atua como servidora de jogo-da-velha (classe `ServidorDeJogoDaVelha`) é mostrada na listagem 19.

Listagem 19: O servidor de Jogo-da-Velha

```
package cap312;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.BindException;
import java.net.ServerSocket;
import java.net.Socket;

/**
 * Esta classe implementa um servidor de jogo-da-velha. Este servidor aguarda
 * a conexão de dois clientes e gerencia o jogo entre eles.
 */
public class ServidorDeJogoDaVelha
  {
  // Declaramos uma matriz de caracteres como sendo o tabuleiro do jogo.
  private static char[][] tabuleiro;

  // Assumimos que toda a entrada e saída será feita por strings.
  // Declaramos instâncias de BufferedReader e BufferedWriter para cada cliente.
  private static BufferedReader entrada1,entrada2;
  private static BufferedWriter saída1,saída2;

  // Método que permite a execução da classe.
  public static void main(String[] args)
    {
    // Alocamos memória para o tabuleiro do jogo.
    tabuleiro = new char[3][3];
    try
      {
      // Criamos a instância de ServerSocket que responderá por solicitações
      // à porta 12345.
      ServerSocket servidor = new ServerSocket(12345);
      // O servidor aguarda "para sempre" as conexões.
      while(true)
        {
        // Aguardamos a primeira conexão...
        Socket conexão1 = servidor.accept();
        System.out.println("Conexão 1 aceita para o cliente "+
                           conexão1.getRemoteSocketAddress());
        // Criamos as streams de entrada e saída para o cliente 1.
        entrada1 =
          new BufferedReader(new InputStreamReader(conexão1.getInputStream()));
        saída1 =
          new BufferedWriter(new OutputStreamWriter(conexão1.getOutputStream()));
        // Mandamos uma mensagem de boas vindas.
        enviaMensagem(saída1,"Olá, você será o primeiro jogador (X).");
        enviaMensagem(saída1,"Por favor aguarde o outro jogador...");
        // Aguardamos a segunda conexão...
        Socket conexão2 = servidor.accept();
        System.out.println("Conexão 2 aceita para o cliente "+
                           conexão2.getRemoteSocketAddress());
        // Criamos as streams de entrada e saída para o cliente 1.
        entrada2 =
          new BufferedReader(new InputStreamReader(conexão2.getInputStream()));
        saída2 =
          new BufferedWriter(new OutputStreamWriter(conexão2.getOutputStream()));
        // Mandamos uma mensagem de boas vindas.
```

```
60         enviaMensagem(saída2,"Olá, você será o segundo jogador (O).");
61         enviaMensagem(saída1,"Segundo jogador conectado.");
62         // Quando as duas conexões tiverem sido estabelecidas e as streams
63         // criadas, iniciamos o processamento.
64         processaJogo();
65         // Ao terminar de atender a requisição de jogo, fechamos os streams.
66         entrada1.close();
67         entrada2.close();
68         saída1.close();
69         saída2.close();
70         // Fechamos também as conexões.
71         conexão1.close();
72         conexão2.close();
73         }
74       }
75     // Pode ser que a porta 12345 já esteja em uso !
76     catch (BindException e)
77       {
78       System.out.println("Porta já em uso.");
79       }
80     // Pode ser que tenhamos um erro qualquer de entrada ou saída.
81     catch (IOException e)
82       {
83       System.out.println("Erro de entrada ou saída.");
84       }
85     }
86 
87   // Este método executa a lógica do jogo, tendo acesso aos mecanismos de entrada
88   // e saída de duas conexões.
89   private static void processaJogo()
90     {
91     // A primeira coisa a fazer é "limpar" o tabuleiro.
92     for(int linha=0;linha<3;linha++)
93       for(int coluna=0;coluna<3;coluna++)
94         tabuleiro[linha][coluna] = ' ';
95     // String que será reusada para as entradas de comandos.
96     String jogada;
97     // Coordenadas da jogada.
98     int linha,coluna;
99     // Peça do vencedor.
100    char vencedor;
101    try
102      {
103      // Enviamos o tabuleiro para os dois jogadores pela primeira vez.
104      mostraTabuleiro();
105      // Executamos um laço "eterno"
106      while(true)
107        {
108        // Lemos e verificamos a jogada do primeiro jogador.
109        enviaMensagem(saída2,"Aguarde movimento do jogador X.");
110        enviaMensagem(saída1,"Jogador X, entre seu movimento.");
111        jogada = entrada1.readLine();
112        enviaMensagem(saída1,"Jogador X escolheu "+jogada);
113        enviaMensagem(saída2,"Jogador X escolheu "+jogada);
114        // Quais são as coordenadas da jogada ?
115        linha = jogada.charAt(0)-'A'; // 'A' = 0; 'B' = 1; 'C' = 2;
116        coluna = jogada.charAt(1)-'1'; // '1' = 0; '2' = 1; '3' = 2;
117        // A jogada é válida ?
118        if ((linha >= 0) && (linha <= 2) &&
119            (coluna >= 0) && (coluna <= 2) &&
120            (tabuleiro[linha][coluna] == ' '))
```

```
121              {
122              tabuleiro[linha][coluna] =  'X';
123              }
124            else
125              {
126              enviaMensagem(saída1,"Jogada do jogador X em posição inválida.");
127              enviaMensagem(saída2,"Jogada do jogador X em posição inválida.");
128              }
129            // Enviamos o tabuleiro para os dois jogadores.
130            mostraTabuleiro();
131            // Verificamos se alguém venceu o jogo.
132            vencedor = verificaVencedor();
133            if (vencedor == 'V')
134              {
135              mostraTabuleiro();
136              enviaMensagem(saída1,"Empate!");
137              enviaMensagem(saída2,"Empate!");
138              break;
139              }
140            else if (vencedor != ' ')
141              {
142              mostraTabuleiro();
143              enviaMensagem(saída1,"Vencedor: jogador "+vencedor);
144              enviaMensagem(saída2,"Vencedor: jogador "+vencedor);
145              break;
146              }
147            // Lemos e verificamos a jogada do segundo jogador.
148            enviaMensagem(saída1,"Aguarde movimento do jogador O.");
149            enviaMensagem(saída2,"Jogador O, entre seu movimento.");
150            jogada = entrada2.readLine();
151            enviaMensagem(saída1,"Jogador O escolheu "+jogada);
152            enviaMensagem(saída2,"Jogador O escolheu "+jogada);
153            // Quais são as coordenadas da jogada ?
154            linha = jogada.charAt(0)-'A'; // 'A' = 0; 'B' = 1; 'C' = 2;
155            coluna = jogada.charAt(1)-'1'; // '1' = 0; '2' = 1; '3' = 2;
156            // A jogada é válida ?
157            if ((linha >= 0) && (linha <= 2) &&
158                (coluna >= 0) && (coluna <= 2) &&
159                (tabuleiro[linha][coluna] == ' '))
160              {
161              tabuleiro[linha][coluna] =  'O';
162              }
163            else
164              {
165              enviaMensagem(saída1,"Jogada do jogador O em posição inválida.");
166              enviaMensagem(saída2,"Jogada do jogador O em posição inválida.");
167              }
168            // Enviamos o tabuleiro para os dois jogadores.
169            mostraTabuleiro();
170            // Verificamos se alguém venceu o jogo.
171            vencedor = verificaVencedor();
172            if (vencedor == 'V')
173              {
174              mostraTabuleiro();
175              enviaMensagem(saída1,"Empate!");
176              enviaMensagem(saída2,"Empate!");
177              break;
178              }
179            else if (vencedor != ' ')
180              {
181              enviaMensagem(saída1,"Vencedor: jogador "+vencedor);
```

```
182            enviaMensagem(saída2,"Vencedor: jogador "+vencedor);
183            break;
184            }
185          } // fim do while true
186        } // fim do bloco try
187      // Se houve algum erro de entrada ou saída...
188      catch (IOException e)
189        {
190        System.out.println("Erro executando o laço principal do jogo !");
191        }
192      }
193
194   /**
195    * Este método mostra o tabuleiro do jogo para os dois clientes. Ele monta
196    * uma string contendo o tabuleiro (inclusive movimentos já jogados) com alguma
197    * decoração para o usuário saber que posições ainda podem ser jogadas. A string
198    * é então enviada para os dois clientes.
199    */
200    private static void mostraTabuleiro()
201      {
202      String tabuleiroFormatado = "";
203      String tempLinha;
204      tabuleiroFormatado += "        \n";
205      tabuleiroFormatado += "  1 2 3 \n";
206      tabuleiroFormatado += " +-+-+-+\n";
207      for(int linha=0;linha<3;linha++)
208        {
209        tempLinha = (char)('A'+linha)+"|";
210          for(int coluna=0;coluna<3;coluna++)
211          tempLinha += tabuleiro[linha][coluna]+"|";
212          tempLinha += "\n";
213          tabuleiroFormatado += tempLinha;
214        tabuleiroFormatado += " +-+-+-+\n";
215        }
216      enviaMensagem(saída1,tabuleiroFormatado);
217      enviaMensagem(saída2,tabuleiroFormatado);
218      }
219
220   /**
221    * Este método verifica se houve algum vencedor considerando a situação atual do
222    * tabuleiro. Se houver vencedor, o método retorna o caracter correspondente ao
223    * vencedor (X ou O), senão retorna espaço.
224    */
225    private static char verificaVencedor()
226      {
227      // Vamos verificar as 8 possíveis combinações para ganhar o jogo.
228      // Primeira linha.
229      if ((tabuleiro[0][0] == tabuleiro[0][1]) &&
230          (tabuleiro[0][1] == tabuleiro[0][2])) return tabuleiro[0][0];
231      // Segunda linha.
232      else if ((tabuleiro[1][0] == tabuleiro[1][1]) &&
233          (tabuleiro[1][1] == tabuleiro[1][2])) return tabuleiro[1][0];
234      // Terceira linha.
235      else if ((tabuleiro[2][0] == tabuleiro[2][1]) &&
236          (tabuleiro[2][1] == tabuleiro[2][2])) return tabuleiro[2][0];
237      // Primeira coluna.
238      else if ((tabuleiro[0][0] == tabuleiro[1][0]) &&
239          (tabuleiro[1][0] == tabuleiro[2][0])) return tabuleiro[0][0];
240      // Segunda coluna.
241      else if ((tabuleiro[0][1] == tabuleiro[1][1]) &&
242          (tabuleiro[1][1] == tabuleiro[2][1])) return tabuleiro[0][1];
```

```
243      // Terceira coluna.
244      else if ((tabuleiro[0][2] == tabuleiro[1][2]) &&
245          (tabuleiro[1][2] == tabuleiro[2][2])) return tabuleiro[0][2];
246      // Diagonal descendente.
247      else if ((tabuleiro[0][0] == tabuleiro[1][1]) &&
248          (tabuleiro[1][1] == tabuleiro[2][2])) return tabuleiro[0][0];
249      // Diagonal ascendente.
250      else if ((tabuleiro[2][0] == tabuleiro[1][1]) &&
251          (tabuleiro[1][1] == tabuleiro[0][2])) return tabuleiro[2][0];
252      else // Nenhuma das combinações existe.
253        {
254        // Ainda existe posições abertas no tabuleiro ?
255        int posiçõesAbertas = 0;
256        for(int linha=0;linha<3;linha++)
257          for(int coluna=0;coluna<3;coluna++)
258            if(tabuleiro[linha][coluna] == ' ') posiçõesAbertas++;
259        if (posiçõesAbertas == 0) return 'V'; // Velha !
260        else return ' '; // Ninguém venceu até agora.
261        }
262      }
263
264  /**
265   * Este método facilita o envio de mensagens para os clientes, executando
266   * o métodos write, newLine e flush de uma instância de BufferedWriter.
267   */
268   private static void enviaMensagem(BufferedWriter bw,String mensagem)
269     {
270     try
271       {
272       bw.write(mensagem);
273       bw.newLine();
274       bw.flush();
275       }
276     catch (IOException e)
277       {
278       System.out.println("Erro enviando mensagem.");
279       }
280     }
281
282   }
```

Um exemplo de interação de um cliente (`telnet`) com o servidor de jogo-da-velha pode ser visto na figura 15.

8 Aplicações baseadas em um cliente e vários servidores

Se considerarmos que *servidor* é o lado da aplicação que fornece uma informação ou executa um processamento e *cliente* é o lado que consume esta informação ou usa o recurso do servidor, podemos imaginar aplicações aonde ao invés de ter um ou mais clientes usando os serviços de um servidor, podemos ter um único cliente usando os serviços de vários servidores.

Consideremos uma tarefa computacional cujo tempo para execução seja potencialmente longo, mas que possa ser dividida em subtarefas menores. Podemos escrever uma apli-

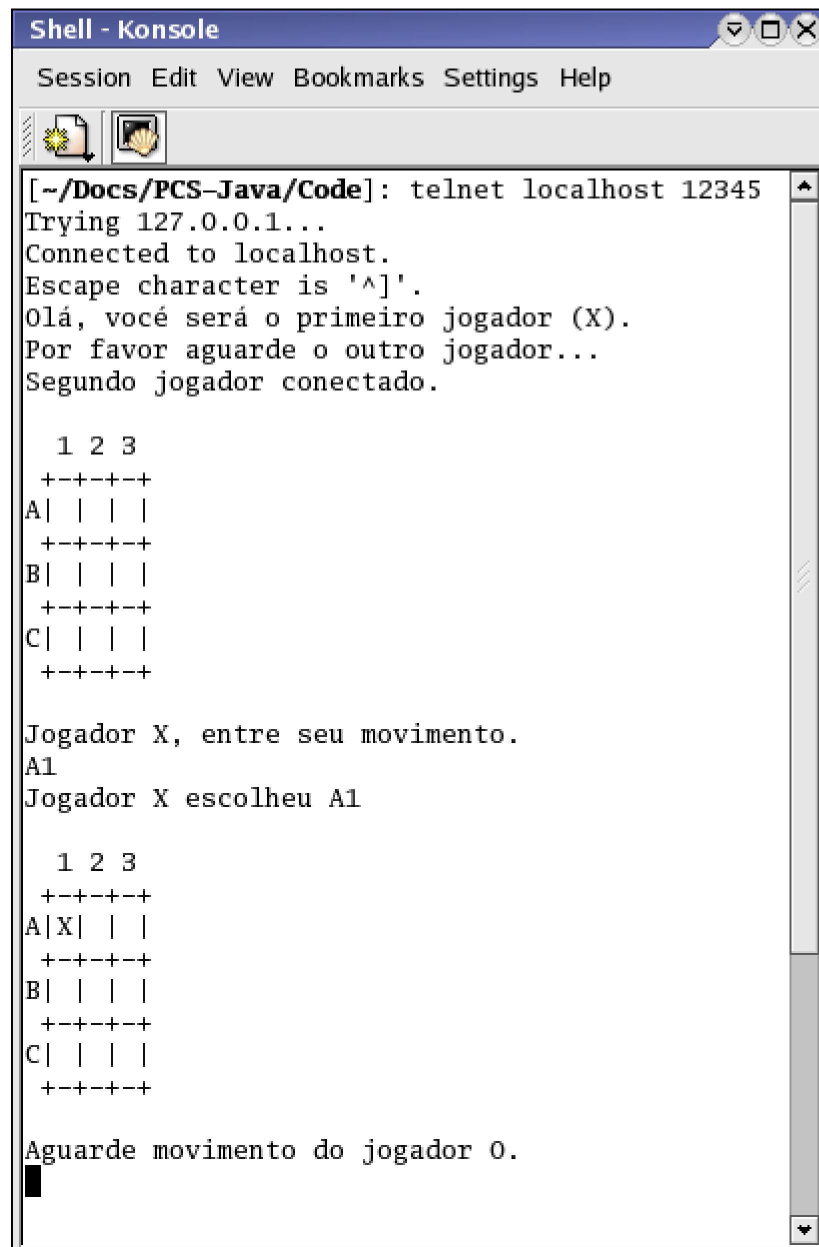

Figura 15: Exemplo de interação entre um cliente (`telnet`) e servidor de jogo-da-velha

cação servidora que será executada ao mesmo tempo em vários computadores, cada uma responsável pela solução de uma parte do problema. Podemos criar uma aplicação cliente que solicita a cada aplicação servidora a execução de sua parte, e que integra as soluções em uma só.

Evidentemente este tipo de aplicação deve envolver o processamento de dados cuja transmissão entre cliente e servidor gaste muito menos tempo do que o processamento, caso contrário a execução da aplicação será mais lenta. Cuidados especiais também devem ser tomados para garantir a integridade do resultado, caso um dos servidores falhe. O lado cliente das aplicações também precisa ser criado de forma a dar suporte a múltiplas linhas de execução (seção 6.2), caso contrário ele não poderá enviar requisições a vários servidores simultaneamente. Também será necessário usar algum mecanismo que informe à aplicação cliente que todos os servidores já terminaram o processamento

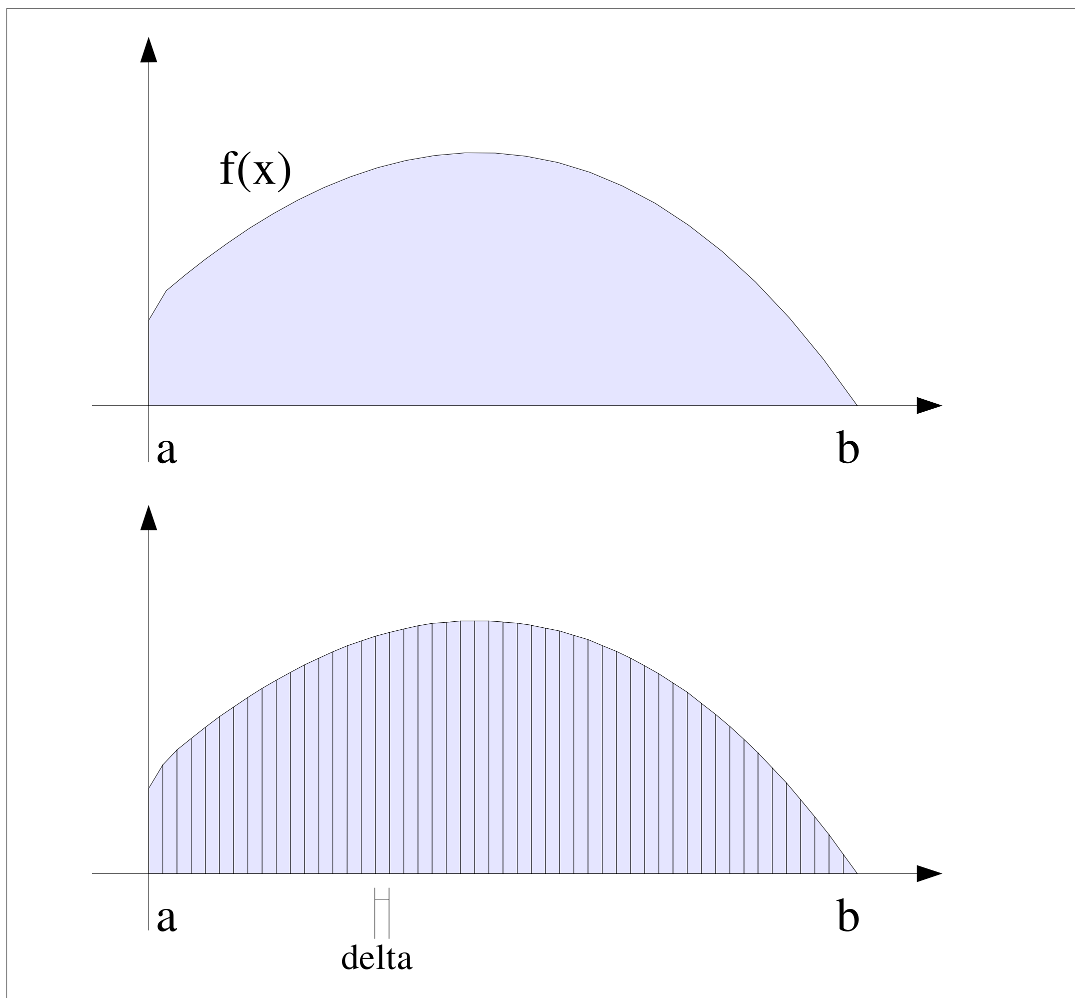

Figura 16: Cálculo de uma integral usando partições e trapézios

de seus dados.

Nesta seção veremos um exemplo de tarefa computacional que pode ser dividida, enviada a servidores para processamento e integrada pelo cliente, alcançando assim um possível tempo menor de execução quando comparado com a execução em um único computador.

8.1 Exemplo: Cliente e servidores para cálculo de integrais

Um bom exemplo de problema que pode ser dividido em pequenos problemas menores é o cálculo de integrais. Consideremos uma função matemática qualquer, considerada entre os valores A e B. O valor da integral desta função entre A e B é a área que a função projeta sobre o eixo X, como mostrado na figura 8.1.

Na parte superior da figura 8.1 temos uma função $f(x)$ definida entre os pontos A e B. O valor da integral de $f(x)$ entre A e B é a área mostrada em cinza na figura 8.1. Uma maneira de calcular esta área de forma aproximada é dividir a área em muitos peque-

nos trapézios que cobrem a mesma área, como mostrado na parte inferior da figura 8.1. Para calcular a área coberta pela função basta então calcular a somatória das áreas dos trapézios. Considerando a função $f(x)$ entre os pontos A e B e que a área da função será dividida de forma que a largura da base de cada trapézio seja igual a $delta$, podemos calcular a área do mesmo como sendo a somatória da área do trapézio cuja base é $delta$, lado direito é igual a $f(a)$ e lado esquerdo igual a $f(a+delta)$, ou seja, igual a $delta \times (f(a)+f(a+delta))/2$.

De posse do método e equações acima, é fácil ver que podemos dividir o problema da somatória em problemas menores, onde cada um deles, para ser executado, deverá receber o valor de A (primeiro valor, ou valor inferior para cálculo), B (segundo valor, ou valor superior), $delta$ e claro, a função a ser integrada.

O protocolo de comunicação entre cliente e servidor(es) de cálculo de integrais é mostrado na figura 17. Nesta figura mostramos somente um cliente (o normal para este tipo de aplicação) e um servidor, mas em casos normais o cálculo deverá ser efetuado usando vários servidores. Uma linha cinza é usada para mostrar, do lado do cliente, que trecho do código será executado concomitantemente pelas linhas de execução do cliente. Para cada linha de execução do lado do cliente, teremos uma conexão com um dos servidores.

O código para o servidor de cálculo de integrais é mostrado na listagem 20.

Listagem 20: O servidor de cálculo de integrais

```
1  package cap312;
2  import java.io.DataInputStream;
3  import java.io.DataOutputStream;
4  import java.io.IOException;
5  import java.net.BindException;
6  import java.net.ServerSocket;
7  import java.net.Socket;
8
9  /**
10  * Esta classe implementa um servidor simples que fica aguardando conexões de
11  * clientes. Quando uma conexão é solicitada, o servidor lê do cliente o
12  * intervalo e precisão para o cálculo da função, efetua o cálculo e retorna
13  * para o cliente o resultado.
14  */
15 public class ServidorDeCalculoDeIntegrais
16   {
17   // Método que permite a execução da classe.
18   public static void main(String[] args)
19     {
20     // O número da porta será obtido da linha de comando.
21     int porta = Integer.parseInt(args[0]);
22     ServerSocket servidor;
23     try
24       {
25       // Criamos a instância de ServerSocket que responderá por solicitações
26       // à porta.
27       servidor = new ServerSocket(porta);
```

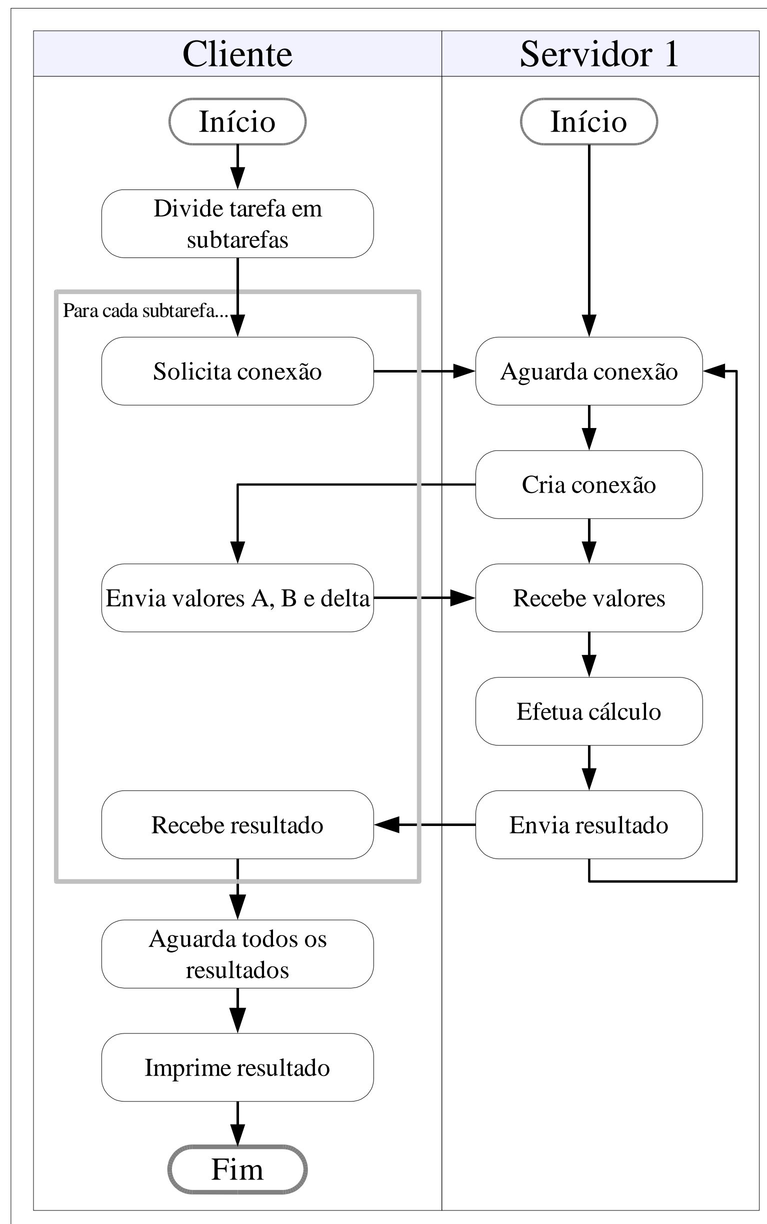

Figura 17: Protocolo para aplicação que faz o cálculo de uma integral

```
28        // O servidor aguarda "para sempre" as conexões.
29        while(true)
30          {
31          // Quando uma conexão é feita,
32          Socket conexão = servidor.accept();
33          // ... o servidor a processa.
34          processaConexão(conexão);
35          }
36        }
37      // Pode ser que a porta já esteja em uso !
38      catch (BindException e)
39        {
40        System.out.println("Porta "+porta+" já em uso.");
41        }
42      // Pode ser que tenhamos um erro qualquer de entrada ou saída.
43      catch (IOException e)
44        {
45        System.out.println("Erro de entrada ou saída.");
```

```
46          }
47        }
48
49      // Este método atende a uma conexão feita a este servidor.
50      private static void processaConexão(Socket conexão)
51        {
52        try
53          {
54          // Criamos uma stream para receber valores nativos, usando a stream de entrada
55          // associado à conexão.
56          DataInputStream entrada =
57            new DataInputStream(conexão.getInputStream());
58          // Criamos uma stream para enviar valores nativos, usando a stream de saída
59          // associado à conexão.
60          DataOutputStream saída =
61            new DataOutputStream(conexão.getOutputStream());
62          // Lemos do cliente o intervalo da função.
63          double início = entrada.readDouble();
64          double fim = entrada.readDouble();
65          // Lemos do cliente a largura do trapézio para cálculo da função.
66          double delta = entrada.readDouble();
67          // Fazemos o cálculo.
68          double resultado = calculaIntegral(início,fim,delta);
69          // Enviamos para o cliente o valor do cálculo.
70          saída.writeDouble(resultado);
71          // Para demonstrar que realmente funciona, vamos imprimir um log.
72          System.out.println("Acabo de enviar o resultado "+resultado+" para o cliente "+
73                             conexão.getRemoteSocketAddress());
74          // Ao terminar de atender a requisição, fechamos as streams.
75          entrada.close();
76          saída.close();
77          // Fechamos também a conexão.
78          conexão.close();
79          }
80        // Se houve algum erro de entrada ou saída...
81        catch (IOException e)
82          {
83          System.out.println("Erro atendendo a uma conexão !");
84          }
85        }
86
87     /**
88      * Este método calcula a integral de uma função entre os valores passados
89      * usando como delta o valor também passado. O método usa a regra do
90      * trapézio.
91      */
92      private static double calculaIntegral(double início,double fim,double delta)
93        {
94        double a = início;
95        double somaDasÁreas = 0;
96        // Vamos percorrendo os valores para argumentos da função...
97        while((a+delta) <= fim)
98          {
99          double alturaA = função(a);
100         double alturaB = função(a+delta);
101         // Calculamos a área do trapézio local.
102         double áreaTrapézio = delta*(alturaA+alturaB)/2;
103         // Somamos à área total.
104         somaDasÁreas += áreaTrapézio;
105         a += delta;
106         }
```

```
107     return somaDasÁreas;
108     }
109
110  /**
111   * Este método calcula uma função em determinado ponto.
112   */
113   private static double função(double argumento)
114     {
115     return Math.sin(argumento)*Math.sin(argumento);
116     }
117
118   }
```

Alguns pontos notáveis no código da classe `ServidorDeCalculoDeIntegrais` (listagem 20) são:

- Quando executarmos a aplicação servidora devemos passar o número da porta como parâmetro. Em algumas situações isso pode ser incômodo (o cliente deverá saber em que endereços **e** em que portas ele deve se conectar), o uso de uma porta padrão é recomendado (isso não foi feito para finalidades de testes).
- O servidor usa o mesmo padrão de desenvolvimento de outros servidores: um laço "eterno" que aguarda conexões do cliente para receber seus dados, processá-los e retorná-los. Desta forma o mesmo servidor pode ser reutilizado por um ou mais clientes, embora somente processe uma requisição de cada vez – o servidor não executa múltiplas linhas de execução.
- O método `processaConexão` processa uma única conexão do cliente, onde serão recebidos os valores de A, B e $delta$; e enviado o valor da integral entre A e B. Este método executa o método `calculaIntegral` usando estes parâmetros, e calculando a somatória das áreas de todos os trapézios entre A e B. Para isto, o método `calculaIntegral` executa o método `função` que calcula o valor da função em um determinado ponto. Para este exemplo, a função foi codificada diretamente no servidor, e é a função $sin(x) * sin(x)$, escolhida pois sua integral entre os valores 0 e $\pi/2$ é exatamente[6] igual a $\pi/4$, facilitando a verificação do algoritmo.

Conforme mencionado anteriormente, o cliente para esta aplicação deverá ser capaz de tratar com linhas de execução. Mais exatamente, o cliente deverá criar uma linha de execução para cada subtarefa que será enviada por sua parte para um servidor. A classe que representa uma subtarefa ou linha de execução é a classe `LinhaDeExecucaoDeCalculoDeIntegrais`, mostrada na listagem 21.

Listagem 21: Uma linha de execução para o cliente de cálculo de integrais

```
1 package cap312;
2 import java.io.DataInputStream;
```

[6]Usaremos o valor do resultado desejado ($\pi/4$) para comparar o resultado obtido, mas já sabendo de antemão que os resultados não serão exatamente iguais por causa de vários fatores: imprecisão do valor $\pi/4$, erros de arredondamento naturais e precisão do algoritmo de cálculo dependente do valor $delta$.

```
3  import java.io.DataOutputStream;
4  import java.io.IOException;
5  import java.net.Socket;
6  import java.net.UnknownHostException;
7
8  /**
9   * Esta classe representa o processamento que deve ser feito quando uma
10  * requisição de cálculo de intgrais for feita a um servidor.
11  * A classe herda de Thread para que várias instâncias dela possam ser
12  * executadas concorrentemente.
13  */
14 public class LinhaDeExecucaoDeCalculoDeIntegrais extends Thread
15   {
16   // Precisamos armazenar o socket correspondente à conexão.
17   private Socket conexão = null;
18   // Também precisamos armazenar os valores para cálculo.
19   private double início,fim,delta,resultado;
20
21  /**
22   * O construtor desta classe recebe como argumento o número da porta do
23   * servidor e o armazena em um campo da classe. O construtor também recebe
24   * como argumentos o intervalo e delta para cálculo da integral.
25   */
26   public LinhaDeExecucaoDeCalculoDeIntegrais(String servidor,int porta,
27                                              double i,double f,double d)
28     {
29     try
30       {
31       conexão = new Socket(servidor,porta);
32       }
33     catch (UnknownHostException e)
34       {
35       System.out.println("Host desconhecido !");
36       }
37     catch (IOException e)
38       {
39       System.out.println("Erro de entrada ou saída !");
40       }
41     início = i;
42     fim = f;
43     delta = d;
44     }
45
46  /**
47   * Este método executa a rotina de atendimento a uma conexão com o servidor.
48   */
49   public void run()
50     {
51     try
52       {
53       // Criamos uma stream para receber valores nativos, usando a stream de entrada
54       // associado à conexão.
55       DataInputStream entrada =
56         new DataInputStream(conexão.getInputStream());
57       // Criamos uma stream para enviar valores nativos, usando a stream de saída
58       // associado à conexão.
59       DataOutputStream saída =
60         new DataOutputStream(conexão.getOutputStream());
61       // Enviamos ao servidor o intervalo da função.
62       saída.writeDouble(início);
63       saída.writeDouble(fim);
```

```
64        // Enviamos ao servidor o número de intervalos da função.
65        saída.writeDouble(delta);
66        // Lemos do servidor o resultado do cálculo.
67        resultado = entrada.readDouble();
68        // Ao terminar de atender a requisição, fechamos as streams.
69        entrada.close();
70        saída.close();
71        // Fechamos também a conexão.
72        conexão.close();
73        }
74      // Se houve algum erro de entrada ou saída...
75      catch (IOException e)
76        {
77        System.out.println("Erro atendendo a uma conexão !");
78        }
79      }
80   /**
81    * Este método permite recuperar o valor calculado.
82    */
83    public double getResultado()
84      {
85      return resultado;
86      }
87
88    }
```

A classe `LinhaDeExecucaoDeCalculoDeIntegrais` (listagem 21) contém campos para armazenar o nome e porta do servidor com o qual irá se comunicar, além dos parâmetros para cálculo da integral (A, B e $delta$). Estes campos são inicializados pelo construtor da classe. Como a classe herda de `Thread` deve implementar seu método `run` para que suas instâncias possam ser executadas concomitantemente. O método `run` simplesmente cria a conexão com o servidor, envia os dados, recebe o valor da somatória e o armazena para que outra aplicação o possa recuperar com o método `getResultado`.

A classe que implementa o cliente, que por sua vez irá criar algumas instâncias da classe `LinhaDeExecucaoDeCalculoDeIntegrais`, é a classe `ClienteDeCalculoDeIntegrais`, mostrada na listagem 22.

Listagem 22: O cliente de cálculo de integrais

```
1  package cap312;
2  /**
3   * Esta classe implementa um cliente simples para o serviço de cálculo de
4   * integrais.
5   * Ele se conecta a dois servidores (pela portas nas quais estes servidores
6   * estão respondendo), envia trechos para cálculo da integral e calcula a soma
7   * dos resultados.
8   */
9  public class ClienteDeCalculoDeIntegrais
10   {
11   public static void main(String[] args)
12     {
13     String servidor = "localhost";
14     // Dividimos a tarefa em duas metades que serão executadas com diferentes
15     // precisões.
```

```
16      LinhaDeExecucaoDeCalculoDeIntegrais l1 =
17        new LinhaDeExecucaoDeCalculoDeIntegrais("localhost",11111,
18                                            0,Math.PI/4,0.000001);
19      LinhaDeExecucaoDeCalculoDeIntegrais l2 =
20        new LinhaDeExecucaoDeCalculoDeIntegrais("localhost",11112,
21                                            Math.PI/4,Math.PI/2,0.0000001);
22      // Iniciamos a execução das duas tarefas.
23      l1.start();
24      l2.start();
25      // Aguardamos até que as duas instâncias tenham terminado de processar
26      // o método run.
27      while(l1.isAlive() || l2.isAlive())
28        {
29        }
30      // Calculamos o resultado final e comparamos com o esperado.
31      double resFinal = l1.getResultado()+l2.getResultado();
32      System.out.println("Resultado:"+resFinal);
33      System.out.println("Esperado :"+Math.PI/4);
34      }
35    }
```

A classe `ClienteDeCalculoDeIntegrais` (listagem 22) simplesmente cria duas instâncias da classe `LinhaDeExecucaoDeCalculoDeIntegrais`, usando (para demonstração), o servidor local e duas portas previamente definidas, onde servidores estão aguardando conexões. Cada linha de execução calculará metade dos trapézios correspondentes à área da integral, mas a segunda linha de execução calculará a sua parte da integral com um valor de *delta* mais preciso, para que o tempo de execução seja definitivamente diferente.

Em algum ponto precisaremos verificar se as linhas de execução já terminaram o seu processamento, para evitar erros grosseiros de cálculo. Estes erros podem acontecer facilmente pois a partir do momento em que o método `start` da classe `LinhaDeExecucaoDeCalculoDeIntegrais` for executado, já será possível executar o método `getResultado` das mesmas, que pode retornar zero se o cálculo não tiver sido terminado – em outras palavras, se o método `run` ainda estiver sendo executado.

Para garantir que o método `getResultado` somente será executado depois que o método `run` tiver terminado seu processamento, podemos criar um laço "infinito" que aguarda até que o resultado dos métodos `isAlive` das classes que herdam de `Thread` sejam ambos `false` – este método verifica se uma linha de execução ainda está "viva", se o método retorna `true` se a linha de execução ainda está sendo executada. Ao final do processamento basta somar os resultados obtidos pelas duas linhas de execução e apresentar o resultado esperado para comparação.

9 Mais informações

Considero este documento ainda inacabado – existem outros tópicos interessantes que podemos explorar, como por exemplo:

- Criação de aplicações que usem servidores em múltiplas camadas, como servidores de meta-dados, por exemplo. Estas aplicações tem clientes que precisam de dados mas não sabem onde estes dados estão: estes clientes se conectam a um servidor de meta-dados, que por sua vez se comunica com vários servidores (conhecidos) de dados para descobrir qual deles contém a informação que o cliente necessita.
- Balanceamento simples de carga: consideremos o exemplo do cálculo de integrais apresentado na seção 8.1. Duas linhas de execução foram criadas, cada uma efetuando um cálculo desbalanceado: a primeira linha de execução demorou muito mais do que a segunda. Podemos considerar esquemas simples de balanceamento, que fazem com que servidores aparentemente ociosos recebam carga a mais, para tentar chegar ao final do processamento como um todo em menos tempo.
- Serviços de descoberta de servidores e balanceamento dinâmico, que combinam as idéias apresentadas nos dois tópicos acima. O cliente de cálculo de integrais poderia então tentar descobrir quais servidores existem e estão à disposição, mesmo enquanto estiver sendo executado, para obter melhor performance. Esta descoberta seria mediada por um servidor de meta-dados, que estaria sendo atualizado permanentemente com informações sobre os servidores e sua carga.
- Explorar outros exemplos como renderização de cenas por computadores distribuídos, processamento de imagens distribuído, sreviços de análise e descoberta de informações, jogos multi-usuários, serviços *Peer to Peer*, etc. – como estes exemplos envolvem outras tecnologias e conhecimentos além de programação, possivelmente serão tratados caso a caso, se houver interesse por parte de alguém.

Como existiram restrições de tempo quando escrevi este documento, e o conteúdo do mesmo atende plenamente à finalidade original (servir de subsídio aos alunos da disciplina Linguagem de Programação Orientada a Objetos do curso de Tecnólogo em Redes de Computadores do Instituto Brasileiro de Tecnologia Avançada - IBTA, Unidade de São José dos Campos), encerrei o documento com somente os tópicos apresentados. Em outra ocasião, se houver demanda, poderei cobrir alguns dos tópicos sugeridos acima.

Rafael Santos
Março de 2004