Java aplicada a Redes de Computadores

Prof Rogério Santos Pozza e Prof. Henrique Yoshikazu Shishido

Universidade Tecnológica Federal do Paraná
Campus Cornélio Procópio

1 Protocolo UDP

Algumas aplicações escritas para se comunicarem na rede podem não precisam ser confiáveis. E ainda mais, se uma aplicação não precisa de garantia de entrega dos pacotes em ordem, pode-se explorar o protocolo UDP. O UDP (*User Datagram Protocol*) confere uma comunicação em que aplicações enviam dados, chamados de datagramas, uma para outra. Um datagrama é independente dos demais e cada um contém a mensagem, os endereços de destino e o horário de entrega.

O Java oferece a implementação de comunicação UDP por meio das classes `DatagramPacket` e `DatagramSocket` contidas no pacote `java.net`. Antes de qualquer implementação, é importante estar ciente das principais características do protocolo UDP:

- **Sem garantia de entrega**: um datagrama enviado pode se perder durante a sua transmissão e o UDP não faz nenhuma verificação se este foi entregue ao seu destino final ou não;
- **Sem entrega ordenada**: quando uma mensagem é enviada, esta pode ser dividida em vários datagramas para que possa caber nos "envelopes"cujo tamanho da mensagem é limitado. Quando enviados, esses datagramas podem chegar de modo desordenado e o UDP não possui nenhuma regra de ordenação das sequências;
- **Maior velocidade que o protocolo TCP**: uma vez que não é necessário controlar a sequêcia dos pacotes a serem entregues, muito menos garantir a entrega e verificar a integridade dos dados, o UDP é um pacote com bom desempenho.

Com essas características em mente, você não será surpreendido caso alguma mensagem enviada não chegue ao seu destino, levando-o a acreditar que houve um erro de implementação. Porém, cuidado, para não atribuir qualquer erro de não entrega do pacote ao protocolo UDP.

Igualmente ao capítulo TCP, vamos implementar um exemplo baseado no modelo cliente-servidor. O exemplo para apresentarmos o UDP em Java é uma aplicação em que o servidor irá oferecer uma única piada. A cada solicitação, o servidor enviará a piada (`String`) ao clientes.

Para isso implementaremos uma classe `Servidor.java` e `Cliente.java`. Primeiramente, a classe `Servidor` será implementada e é apresentada na Listagem 1.1.

1.1 Servidor

Igualmente à classe `Socket`, objetos da classe `DatagramSocket` são específicos para enviar/receber datagramas UDP. A linha 8 da Listagem 1.1 instancia um objeto `dgSocket` da classe `DatagramSocket`, passando como parâmetro ao construtor o número da porta de comunicação. Após instanciado, o objeto `dgSocket` será capaz de utilizar os métodos `send()` e `receive()` para o envio/recebimento de datagramas .

Listagem 1.1: Recepção de datagramas

```
1 import java.net.*;
2 import java.io.*;
3
4 public class ServidorPiada1 {
5
6   public static void main() throws IOException {
7
8     DatagramSocket dgSocket = new DatagramSocket(7777);
9     byte[] mensagem = new byte[128];
10
11    DatagramPacket dgPacket = new DatagramPacket(mensagem,
          mensagem.length);
12
13    dgSocket.receive(dgPacket);
14
15    String msg = new String(dgPacket.getData());
16    System.out.println("A mensagem recebida e: " + msg);
17
18  }
19 }
```

A linha 9 da Listagem 1.1 declara um vetor de *bytes*. Esse vetor irá armazenar a mensagem em sí. Observe que neste exemplo foi declarado um vetor de 128 posições. Neste caso, considerando a mensagem formada por uma `String`, em que cada caracter ocupa 1 byte, logo poderá ser enviada uma mensagem de 128 caracteres. É importante ter em mente que quanto menor o tamanho da mensagem em cada datagrama, menores são os riscos da mensagem se perder durante a transmissão ou chegar corrompida ao seu destino.

Já na linha 11 da Listagem 1.1 instancia um objeto `dgPacket`. Esse objeto servirá para armazenar tudo que estiver relacionado a um datagrama. Foi utilizado o construtor em que são passados dois parâmetros: a variável onde será armazenada a mensagem e o sua capacidade. Neste caso, tal datagrama servirá apenas para o recebimento de dados.

Depois de instanciados os objetos, é possível receber um datagrama através do método `receive()` do objeto `dgSocket` (linha 13 - Listagem 1.1. Este método recebe como parâmetro o objeto `dgPacket` onde será armazenada a mensagem.

Durante o transcorrer desse exemplo, você observará um outro tipo de objeto-datagrama que será instanciado com outro construtor para o envio de datagramas.

Assim que o datagrama foi recebido na linha 13, pode-se atribuir o seu conteúdo a uma String conforme realizado na linha 15 da Listagem 1.1.

O próximo passo é preparar o envio da piada ao computador solicitante. Para isso, é preciso obter o endereço IP e a porta de comunicação deste computador. Toda vez em que um datagrama é recebido, além da mensagem, o endereço/porta do computador emissor também são adicionados ao datagrama. Deste modo, pode-se explorar os métodos `getAddress()` e `getPort()` de um objeto `DatagramPacket`(linha 21 e 22 da Listagem 1.2).

Listagem 1.2: Envio de datagramas

```
1  import java.net.*;
2  import java.io.*;
3
4  public class ServidorPiada2 {
5
6    private static final String piada = "O que cai de pe e
         corre deitado? R: A chuva!";
7
8    public static void main(String args[]) throws IOException {
9
10     DatagramSocket dgSocket = new DatagramSocket(7777);
11     byte[] mensagem = new byte[128];
12
13     DatagramPacket dgPacket = new DatagramPacket(mensagem,
           mensagem.length);
14
15     dgSocket.receive(dgPacket);
16
17     String msg = new String(dgPacket.getData());
18
19     System.out.println("A mensagem do datagrama e: " + msg);
20
21     InetAddress ia = dgPacket.getAddress();
22     int porta = dgPacket.getPort();
23
24     mensagem = piada.getBytes();
25
26     dgPacket = new DatagramPacket(mensagem, mensagem.length,
           ia, porta);
27
28     dgSocket.send(dgPacket);
29
30   }
31 }
```

Para enviar a mensagem definida na String `piada` (linha 6 - Listagem 1.2), é preciso convertê-la para bytes usando o método `getBytes()` que a classe String oferece (linha 24). Depois de convertido, é preciso reinstanciar o objeto `dgPacket` utilizando um outro construtor em que são passados a variável da mensagem, o seu tamanho, o endereço de destino e a porta de comunicação (linha 26).

O datagrama `dgPacket` poderá ser enviado através do método `send()`- linha 28 da Listagem 1.2.

1.2 Cliente

A implementação da aplicação cliente (Listagem 1.3) segue o mesmo raciocínio, porém com uma pequena mudança no construtor do objeto socket.

Listagem 1.3: Cliente piada

```
1  import java.io.*;
2  import java.net.*;
3
4  public class ClientePiada {
5
6    public static void main(String args[]) throws IOException {
7
8      DatagramSocket dgSocket = new DatagramSocket();
9      byte[] mensagem = new byte[128];
10
11     /******* ENVIO ******/
12     String msg = "Quero uma piada.";
13     mensagem = msg.getBytes();
14
15     InetAddress endereco = InetAddress.getByName("localhost")
           ;
16
17     DatagramPacket dgPacket = new DatagramPacket(mensagem,
           mensagem.length, endereco, 7777);
18
19     dgSocket.send(dgPacket);
20
21     /******* RECEBIMENTO *******/
22     mensagem = new byte[128];
23     dgPacket = new DatagramPacket(mensagem, mensagem.length);
24
25     dgSocket.receive(dgPacket);
26
27     String piada = new String(dgPacket.getData());
28     System.out.println(piada);
29
30   }
31 }
```

Na linha 8 não é passada nenhuma porta de comunicação como parâmetro ao construtor, pois o cliente enviará um datagrama primeiramente ao servidor de piadas. O endereço e porta do servidor serão definidos diretamente no construtor do datagrama.

Também foi necessário adicionar um vetor de *bytes* de preferência de mesma capacidade ao do servidor (linha 9). Foi atribuido ao vetor `mensagem` o conteúdo da String `msg` (linha 13). Em seguida, obteve-se o endereço da máquina local (*localhost*) através do método estático `getByName()` da classe `InetAddress`(linha 15). Com essas informações, é possível construir o datagrama `dgPacket` atribuindo ao seu construtor os dados necessários (linha 17).

Para o recebimento de um datagrama (piada), o vetor de *bytes* foi redeclarado a fim de eliminar todo o conteúdo previamente a ele atribuído (linha 22). Caso contrário, a mensagem recebida no datagrama teria apenas o espaço restante desse vetor, correndo o risco de talvez não ser recebida completamente.

Além disso, o objeto `dgPacket` é reinstanciado para poder receber um datagrama. Lembre-se que para o recebimento é utilizado o construtor em que se passa a variável do vetor de *bytes* e o seu tamanho (linha 23).

Ao final de toda a preparação, o datagrama é recebido ao invocar o método `receive()` do objeto `dgSocket`(linha 25). A mensagem é convertida para String e exibida na saída padrão (linha 27 e 28).

2 Multicast

O multicast é a classe D de endereços para que um processo possa enviar uma mensagem pra um determinado grupo de processos. Permite que um único pacote seja enviado para um conjunto de processos que façam parte de um grupo multicast conforme ilustra a Figura 1.

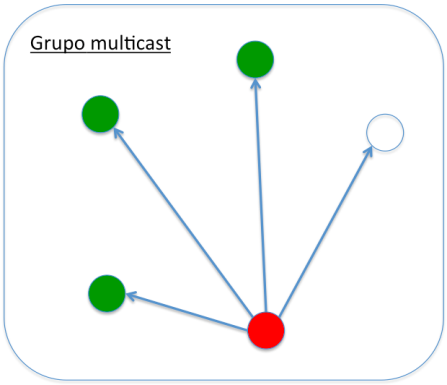

Figura 1: Comunicação multicast

A representação de um grupo multicast é realizado através de um único endereço de classe D. Endereços de classe D compreendem endereços a partir de 224.0.0.0 a 239.255.255.255. Um emissor de uma transmissão multicast não conhece a identidade dos destinatários, muito menos o tamanho do grupo.

Qualquer aplicativo que queira fazer parte de um grupo de multicast, precisa se juntar (**join**) ao grupo multicast. Nesse sentido, todos os processos que estiverem vinculados a um determinado endereço de multicast e uma porta de comunicação em comum receberão qualquer pacote enviado por um membro do grupo, conforme apresentado na Figura 2.

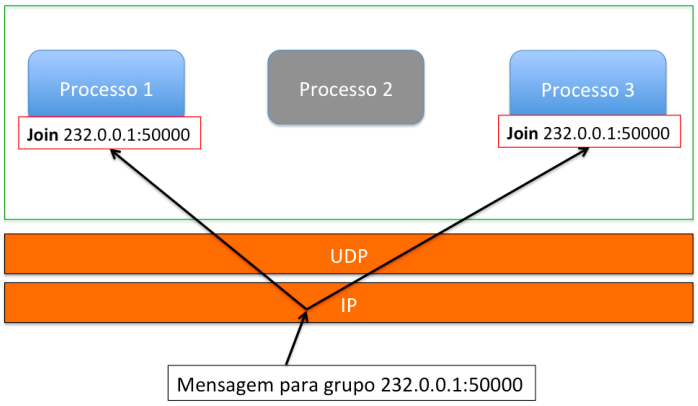

Figura 2: Mensagem para grupo multicast

Para a implementação de uma comunicação multicast, o Java oferece a classe MulticastSocket com seus principais métodos:

- MulticastSocket mSocket = new MulticastSocket(porta);
- s.joinGroup(ipMulticastGrupo)
- s.leaveGroup(ipMulticastGrupo)

O exemplo da Listagem 1.4 apresenta uma aplicação de bate-papo utilizando threads.

Listagem 1.4: Exemplo - bate-papo com multicast

```
1 import java.io.*;
2 import java.net.*;
3
4 public class MulticastChat extends Thread {
5
6   private static String user = null;
7   private static InetAddress endereco;
8   private static int porta;
9
```

```
10  public MulticastChat() {
11  }
12
13  @Override
14  public void run() {
15
16    try {
17
18      byte[] buffer = new byte[64];
19
20      MulticastSocket socket = new MulticastSocket(porta);
21
22      socket.joinGroup(endereco);
23
24      while(true) {
25
26        DatagramPacket receberPacote = new DatagramPacket(
              buffer, buffer.length);
27        socket.receive(receberPacote);
28
29        String mensagem = new String(receberPacote.getData())
              ;
30
31        if(!mensagem.contains(user)) {
32                System.out.print("\r                    ");
33          System.out.println("\n" + mensagem + "\n");
34          System.out.print("Digite a mensagem: ");
35        }
36        buffer = new byte[64];
37      }
38    }catch(Exception e) {
39      System.out.println("Exception: " + e.getMessage());
40    }
41  }
42
43  public static void main(String args[]) {
44
45    if(args.length != 2) {
46            System.out.println("Parametros incorretos: java
                  MulticastChat <multicast> <porta>");
47      System.exit(1);
48    }
49
50    try {
51      porta = Integer.parseInt(args[1]);
52      endereco = InetAddress.getByName(args[0]);
53
54      Thread t = new MulticastChat();
55      t.start();
56
```

```
57        BufferedReader entrada = new BufferedReader(new
              InputStreamReader(System.in));
58        byte[] buffer = new byte[64];
59
60        System.out.print("Digite o seu nome: ");
61        user = entrada.readLine();
62
63
64        MulticastSocket socket = new MulticastSocket();
65        socket.joinGroup(endereco);
66
67        while(true) {
68
69          System.out.print("Digite a mensagem: ");
70          String mensagem = entrada.readLine();
71
72          if(mensagem.equals("sair")) System.exit(0);
73
74          mensagem = user + " diz: " +  mensagem;
75
76          buffer = mensagem.getBytes();
77
78          DatagramPacket enviarPacote = new DatagramPacket(
                buffer, buffer.length, endereco, porta);
79
80          socket.send(enviarPacote);
81        }
82      }catch(Exception e) {
83        System.out.println("Exception: " + e.getMessage());
84      }
85    }
86 }
```

Esta aplicação pode ser abstraída em dois fluxos simultâneoes de execução: um para cuidar do envio de mensagens e outro para receber as mensagens de outros clientes. Neste contexto, é preciso fazer uso de *threads*. Essa aplicação herda a classe Thread (linha 4). O método `main()` verifica se o endereço multicast e uma porta de comunicação são passados como argumentos via linha de comando (linhas 45 a 48). Em seguida, os argumentos são atribuídos às variáveis `porta` e `endereco` (linhas 51 e 52). Uma *thread* é instanciada e inicializada nas linhas 54 e 55, assim, um fluxo que será responsável pelo recebimento de mensagens é iniciado, porém vamos discutí-lo após o analisar as linhas desta função `main()`. O código das linhas 57 a 61 preocupa-se diretamente com a entrada do nome do usuário. Um objeto `MulticastSocket` é instanciado para que o processo possa se juntar ao grupo representado pela variável `endereco` (linha 65).

Em um laço de repetição infinito, uma mensagem é solicitada e tratada (linhas 69 a 72). Um `DatagramPacket` é criado e enviado através do objeto `socket` ao qual diversos processos estão conectados.

Já a *thread* executada pelo método `run()` abre-se uma conexão multicast na porta e conecta-se ao endereço multicast passados via linha de comando (linha 20 e 22). Um laço infinito é executado preparando um datagrama (linha 26) para o recebimento de mensagens através do método `receive()` na linha 27. Após o datagrama recebido, este é convertido para String em que se não houver o nome do usuário local na mensagem, a mensagem é exibida na saída padrão (linhas 29 a 24). Ao final do laço o vetor de *bytes* é redeclarado esvaziando o seu conteúdo, ficando pronto para o recebimento de um novo datagrama.