

Comunicação de Computadores utilizando Sockets

Francisco Assis da Silva

Mestrando FACCAR/UFRGS

FIPP – Faculdade de Informática de Presidente Prudente/UNOESTE

Rua: José Bongiovani, 700, Cidade Universitária

CEP: 19050-900, Presidente Prudente-SP

E-mail: chico@unoeste.br

1 – Introdução

Os computadores da Internet são conectados entre si pelo protocolo TCP/IP. Na década de 1980, a ARPA (Advanced Research Projects Agency) do governo norte-americano forneceu recursos financeiros à Universidade da Califórnia em Berkeley com a finalidade de oferecer uma implementação UNIX do pacote de protocolos TCP/IP. O que foi desenvolvido então ficou conhecido como interface de sockets. Hoje, a interface de sockets é o método mais utilizado para acesso a uma rede TCP/IP [Hopson 97].

A idéia de um socket faz parte do TCP/IP, o conjunto de protocolos usado pela Internet. Um socket essencialmente é uma conexão de dados transparente entre dois computadores numa rede. Ele é identificado pelo endereço de rede dos computadores e por seus pontos finais e uma porta em cada computador. Os computadores em rede direcionam os streams de dados recebidos da rede para programas receptores específicos, associando cada programa a um número diferente, a porta do programa. Da mesma forma, quando o tráfego de saída é gerado, o programa de origem recebe um número de porta para a transação. Caso contrário, o computador remoto poderia não responder à entrada. Determinados números de porta são reservados no TCP/IP para protocolos específicos – por exemplo, 25 para SMTP e 80 para HTTP [Thomas 97].

Um socket não é nada além de uma abstração conveniente. Ele representa um ponto de conexão para uma rede TCP/IP. Quando dois computadores querem manter uma conversação, cada um deles utiliza um socket. Um computador é chamado servidor, ele abre um socket e presta atenção às conexões. O outro computador denomina-se cliente, ele chama o socket servidor para iniciar a conexão. Para estabelecer uma conexão, é necessário apenas um endereço de destino e um número de porta [Hopson 97].

Cada computador em uma rede TCP/IP possui um endereço exclusivo. As portas representam conexões individuais dentro desse endereço. Cada porta de um computador compartilha o mesmo endereço, mas os dados são roteados dentro de cada computador pelo número da porta. Quando um socket é criado, ele tem de estar associado a uma porta específica – o processo é conhecido como acoplamento a uma porta [Hopson 97].

As próximas seções são organizadas como segue: A Seção 2 apresenta os modos de transmissão de sockets. As classes do Java baseadas em conexões para clientes e servidores são apresentadas na Seção 3, juntamente com uma aplicação desenvolvida para este trabalho, um MiniChat utilizando conexões. As classes de datagramas do Java para recepção e transmissão são apresentadas na Seção 4, juntamente com uma aplicação utilizando datagramas, um servidor de eco de mensagens. As conclusões principais deste trabalho são apresentadas na Seção 5.

2 – Modos de Transmissão de Sockets

Os sockets têm dois modos principais de operação: o modo *baseado em conexões* e o modo *sem conexão*. Os sockets baseados em conexões operam como um telefone; eles têm de estabelecer uma conexão e suspender a ligação. Tudo que flui entre esses dois eventos chega na mesma ordem em que foi transmitido. Os sockets sem conexão operam como o correio, a entrega não é garantida, e os diferentes itens da correspondência podem chegar em uma ordem diferente daquela em que foram enviados [Hopson 97].

O modo a ser utilizado é determinado pelas necessidades de um aplicativo. Se a conformidade é

importante, então a operação baseada em conexões é a melhor opção. Os servidores de arquivos precisam fazer todos os seus dados chegarem corretamente e em sequência. Se alguma parte dos dados se perdesse, a utilidade do servidor seria invalidada. Quando precisar de confiabilidade, a aplicação terá que pagar um preço. Garantir a sequência e a correção dos dados exige processamento extra e utilização de mais memória; esse overhead adicional pode reduzir o tempo de resposta de um servidor [Hopson 97].

A operação sem conexão utiliza o UDP (*User Datagram Protocol*). Um datagrama é uma unidade autônoma que tem todas as informações necessárias para tentar fazer sua entrega. Similar a um envelope, o datagrama tem um endereço do destinatário e do remetente e contém em seu interior os dados a serem enviados. Um socket nesse modo de operação não precisa se conectar a um socket de destino; ele simplesmente envia o datagrama. O protocolo UDP só promete fazer o melhor esforço possível para tentar entregar o datagrama. A operação sem conexão é rápida e eficiente, mas não é garantida [Hopson 97].

A operação baseada em conexões emprega o TCP (*Transport Control Protocol*). Um socket nesse modo de operação precisa se conectar ao destino antes de transmitir os dados. Uma vez conectados, os sockets são acessados pelo uso de uma interface de fluxos: abertura-leitura-escrita-fechamento. Tudo que é enviado por um socket é recebido pela outra extremidade da conexão, exatamente na mesma ordem em que foi transmitido. A operação baseada em conexões é menos eficiente do que a operação sem conexão, mas é garantida [Hopson 97].

3 – Classes do Java Baseadas em Conexões

As classes baseadas em conexões do Java têm uma versão cliente e outra servidora. O socket cliente emite uma conexão para um socket servidor que está na escuta. Os sockets clientes são criados e conectados pelo uso de um construtor da classe Socket. A tabela 1 mostra os construtores da classe Socket.

Tabela 1. Construtores da classe Socket [Thomas 97]

Construtor	Descrição
Socket(String, int)	O nome do computador e a porta de conexão.
Socket(String, int, boolean)	O nome do computador, a porta e um booleano indicando se o socket é para streams (true) ou datagramas (false).
Socket(InetAddress, int)	O endereço Internet e a porta para conexão.
Socket(InetAddress, int, boolean)	O endereço Internet, a porta e um booleano, indicando se o socket é para streams (true) ou datagramas (false).

A linha a seguir cria um socket cliente e o conecta a um host:

```
Socket clientSocket = new Socket("microchico", 4321);
```

O primeiro parâmetro é o nome do host a se conectar, que especifica o computador de destino; o segundo parâmetro é o número da porta. É necessário o número da porta para completar a transação e permitir que um aplicativo individual receba a chamada.

A tabela 2 relaciona os métodos para Socket, `getInputStream` e `getOutputStream` são importantes para comunicar com computadores remotos. Streams são geralmente usados para tratar da transferência de dados entre os computadores.

Tabela 2. Métodos da classe Socket [Thomas 97]

Métodos	Descrição
<code>close()</code>	Fecha o socket.
<code>InetAddress getInetAddress()</code>	Retorna o <code>InetAddress</code> do computador no outro lado do socket.
<code>int getLocalPort()</code>	Retorna o número da porta local à qual este socket está ligado.
<code>InputStream getInputStream()</code>	Retorna um <code>InputStream</code> anexado a este socket.
<code>OutputStream getOutputStream()</code>	Retorna um <code>OutputStream</code> anexado a este socket.
<code>SetSocketImplFactory(SocketImplFactory)</code>	Define o socket implementation factory para o sistema.

Pelo fato de ser baseada em conexões, a classe `Socket` fornece uma interface de fluxos para operações de leitura e escrita (gravação). As classes do pacote `java.io` devem ser usadas para acesso a um socket conectado:

```
DataOutputStream outbound = new DataOutputStream(clienteSocket.getOutputStream());
DataInputStream inbound = new DataInputStream(clienteSocket.getInputStream());
```

Quando o programa termina de usar o socket, a conexão precisa ser fechada:

```
outbound.close();
inbound.close();
clienteSocket.close();
```

Todos os fluxos de socket devem ser fechados antes do fechamento do próprio socket.

Sockets Clientes

Topos os programas cliente seguem o mesmo script básico:

1. Criam a conexão de socket cliente.
2. Adquirem fluxos de leitura e escrita para o socket.
3. Utilizam os fluxos de acordo com o protocolo do servidor.
4. Fecham os fluxos.
5. Fecham o socket.

Sockets Servidores

Os servidores não criam conexões de forma ativa. Em vez disso, eles permanecem passivamente aguardando um pedido de conexão de cliente, e depois fornecem seus serviços. Os servidores são criados com um construtor da classe `ServerSocket`. A linha a seguir cria um socket servidor e acopla esse socket à porta 4321:

```
ServerSocket ssocket = new ServerSocket(4321,5);
```

O primeiro parâmetro é o número da porta na qual o servidor deve fazer a escuta. O segundo parâmetro é opcional. O segundo parâmetro indica a profundidade da pilha de escuta. Um servidor pode receber pedidos de conexão de vários clientes ao mesmo tempo, mas cada chamada tem de ser processada isoladamente. A *pilha de escuta* é uma fila de pedidos de conexão não respondidos. O código anterior instrui o driver do socket a manter os

cinco últimos pedidos de conexão. Se o construtor omitir a profundidade da pilha de escuta, será utilizado o valor padrão 50.

Depois que o socket é criado e passa a ficar atendo a pedidos de conexões, cada conexão que chega é criada e colocada na pilha de escuta. O método `accept()` retorna um socket cliente conectado que é utilizado para se comunicar com o chamador:

```
Socket cliSocket = new ServerSocket.accept();
```

Nenhuma conversação é conduzida sobre o próprio socket servidor. Em vez disso, o socket servidor irá gerar um novo socket no método `accept()`. O socket servidor ainda permanece aberto e enfileirando novos pedidos de conexões. Como no caso do socket cliente, o próximo passo é criar um fluxo de entrada e saída:

```
DataInputStream inbound = new
DataInputStream(cliSocket.getInputStream());
DataOutputStream outbound = new
DataOutputStream(cliSocket.getOutputStream());
```

As operações normais de I/O podem agora ser executadas pela utilização dos fluxos recém-criados. Esse servidor espera que o cliente envie uma linha em branco antes de transmitir sua resposta. Quando a conversação se completa, o servidor fecha os fluxos e o socket cliente. Nesse ponto, o servidor tenta aceitar outras chamadas. Quando não há nenhuma chamada na fila, o método irá aguardar a chegada de uma chamada. Esse comportamento é conhecido como *bloqueio*. O método `accept()` irá bloquear o thread do servidor, impedindo que este execute quaisquer outras tarefas até a chegada de uma nova chamada.

Todos os servidores seguem o mesmo script básico:

1. Criam o socket servidor e iniciam a escuta.
2. Chamam o método `accept()` para obter novas conexões.
3. Criam fluxos de entrada e saída para o socket retornado.
4. Conduzem a conversação com base no protocolo combinado.
5. Fecham os fluxos e o socket do cliente.
6. Voltam ao Passo 2 ou continuam até o Passo 7.
7. Fecham o socket servidor.

A figura 1 resume os passos necessários para os aplicativos cliente/servidor baseados em conexões.

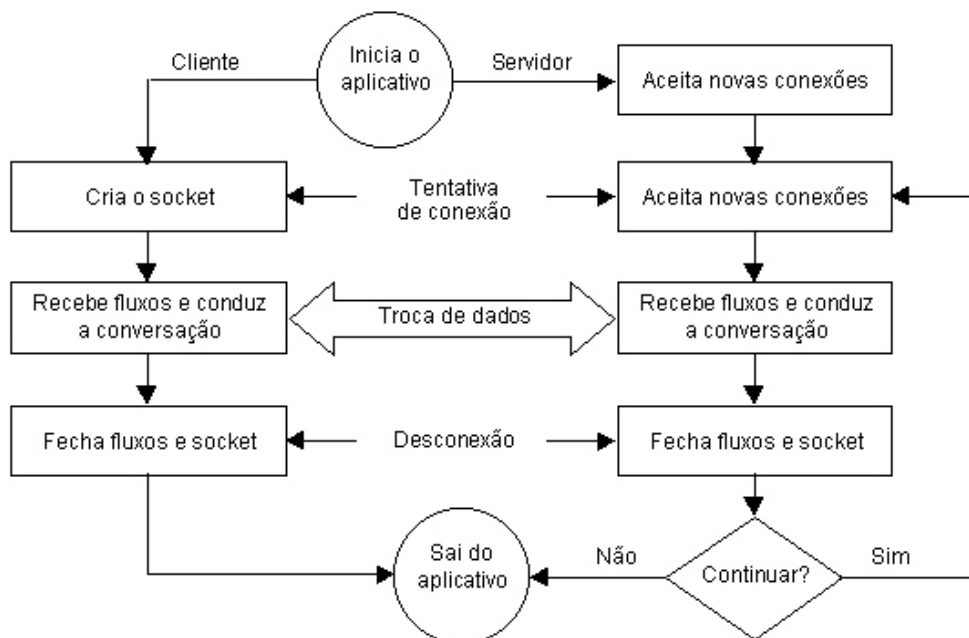


Figura 1. Aplicativo cliente e servidor baseados em conexões [Hopson 97].

3.1 – Uma Aplicação baseada em Conexões

Foi desenvolvida uma Aplicação de Comunicação de Computadores utilizando o protocolo TCP (*Transport Control Protocol*). Dois computadores podem se comunicar por meio do MiniChat implementado utilizando Conexão de Sockets. O socket servidor do MiniChat é ativado e permanece passivo aguardando um pedido de conexão de cliente (fica ouvindo na porta especificada no desenvolvimento (4321)). Os sockets clientes emitem uma conexão para o socket servidor que está na escuta. Quando um socket cliente envia uma mensagem para o socket servidor, o cliente anexa o nome do usuário do MiniChat junto com a mensagem. Daí então o socket servidor repassa para todos os sockets clientes conectados a mensagem recebida.

Para ver os arquivos fontes do Minichat, clique no link a seguir:

[Fonte do servidor socket "chatserver.java", fonte do cliente socket "ClienteCall.java" e fonte HTML para executar a applet do socket cliente "ClienteCall.html".](#)

Para obter os arquivos fontes do MiniChat: servidor socket "chatserver.java", cliente socket "ClienteCall.java" e HTML do socket cliente "ClienteCall.html": [\[chat.zip\]](#).

3.1.1 – Executando o MiniChat

Passo 1:

Executando o servidor do MiniChat (socket servidor):

- Abra uma janela do Prompt do MS-DOS.
- Direcione o Path do Sistema Operacional para C:\jdk1.2.2\bin ou para outro diretório em que o Java 1.2.2 estiver instalado.
- Compile o arquivo chatserver.java para gerar o bytecode (interpretável pela JVM - Java Virtual Machine) usando a seguinte linha de comando: **C:\...>javac -deprecation chatserver.java**
- Para executar o servidor do MiniChat use a seguinte linha de comando: **C:\...>java chatserver**

A figura 2 mostra o servidor socket do MiniChat em uso com duas conexões de clientes realizadas.

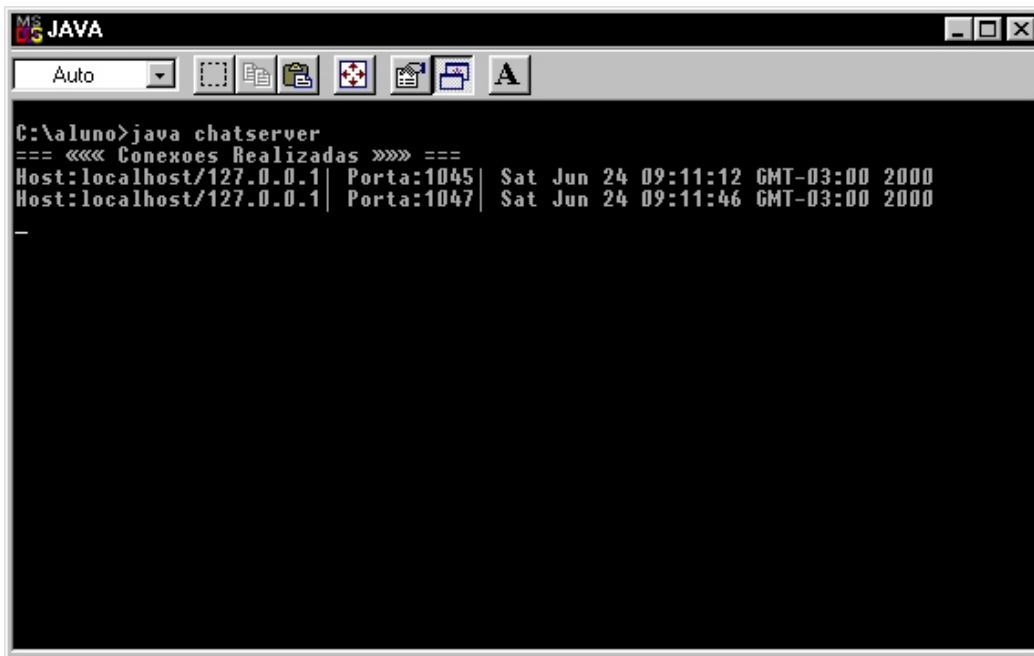


Figura 2. Servidor do MiniChat em uso.

Passo 2:

Executando o cliente do MiniChat (socket cliente):

- Abra uma janela do Prompt do MS-DOS.
- Direcione o Path do Sistema Operacional para C:\jdk1.2.2\bin ou para outro diretório em que o Java 1.2.2 estiver instalado.
- Compile o arquivo ClienteCall.java para gerar o bytecode (interpretável pela JVM - Java Virtual Machine) usando a seguinte linha de comando: **C:\...>javac -deprecation ClienteCall.java**
- Para executar o cliente do MiniChat, será necessário utilizar o *appletviewer* do java, pelo fato do cliente ser escrito na forma de applet. Para isso use a seguinte linha de comando: **C:\...>appletviewer ClienteCall.html**
- Repita todas as etapas do **Passo 2** para executar outro cliente do MiniChat.

A figura 3 mostra dois clientes sendo executados e se comunicando através da tela do socket cliente do MiniChat.

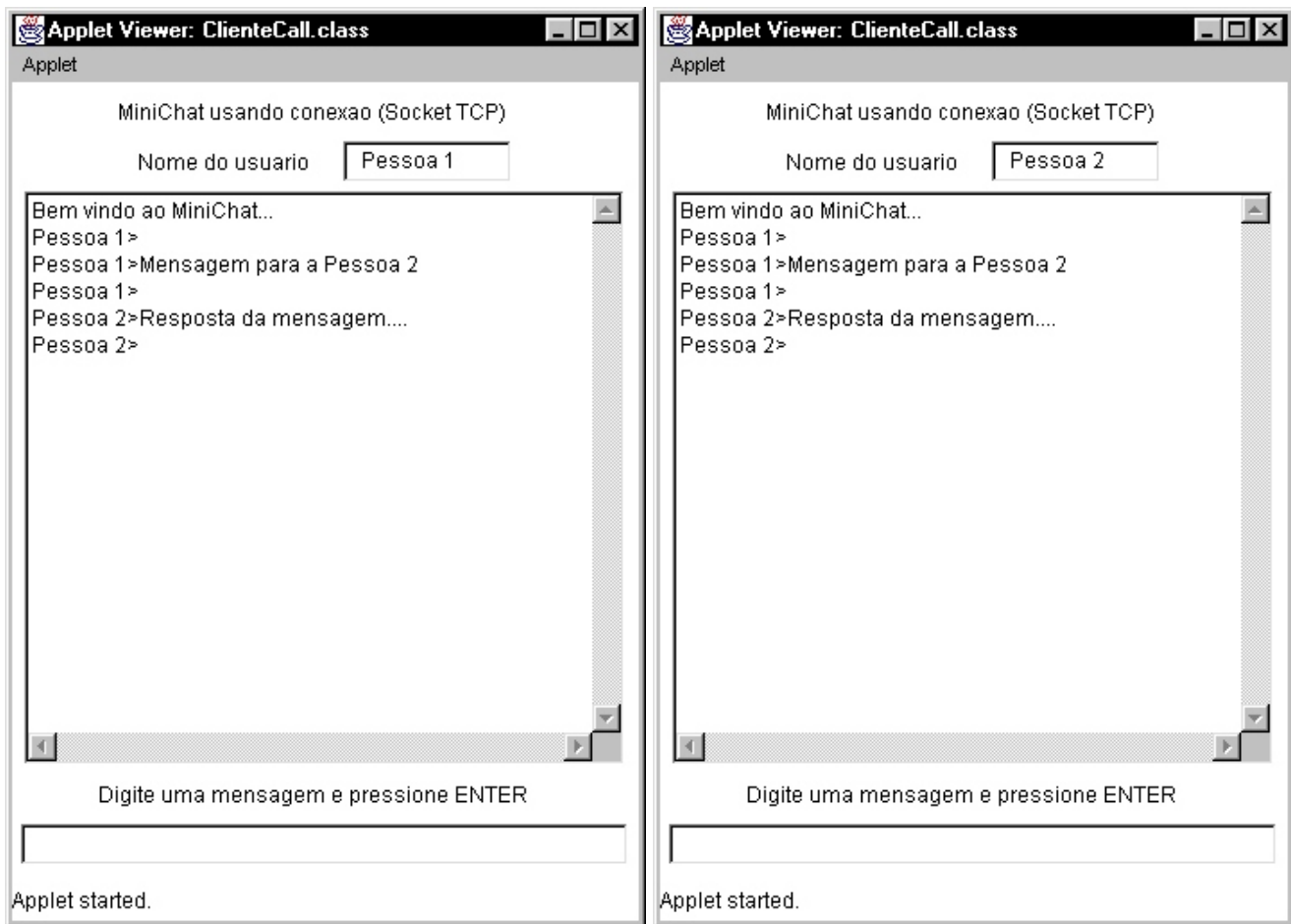


Figura 3. Sockets clientes do MiniChat se comunicando.

4 – Classes de Datagramas do Java

Diferentes das classes baseadas em conexões, as versões de datagramas do cliente e do servidor se comportam de maneira praticamente idêntica, a única diferença ocorre na implementação. A mesma classe é utilizada na metade cliente e na metade servidora. As linhas a seguir criam sockets de datagramas cliente e servidor:

```
DatagramSocket serverSocket = new DatagramSocket(4545);
DatagramSocket clientSocket = new DatagramSocket();
```

O servidor especifica sua porta (4545) no parâmetro do construtor. O cliente pode utilizar qualquer porta disponível para chamar o servidor. O parâmetro omitido do construtor na segunda chamada instrui o sistema operacional a atribuir o número da próxima porta disponível. O cliente poderia ter solicitado uma porta específica, mas a chamada iria falhar se algum outro socket já estivesse acoplado a essa porta.

Recepção de Datagramas

A classe `DatagramPacket` é usada para receber e enviar dados sobre classes `DatagramSocket`. A classe de pacotes contém informações sobre a conexão, bem como os dados. Os datagramas são unidades autônomas de transmissão. A classe `DatagramPacket` encapsula essas unidades. A tabela 3 mostra os construtores e métodos da

classe DatagramPacket.

Tabela 3. Construtores e Métodos da classe DatagramPacket [Thomas 97]

Construtores e Métodos	Descrição
DatagramPacket(byte[], int)	Constrói um pacote a ser usado para receber um datagrama. O conteúdo do datagrama será copiado para o array de bytes quando o datagrama for recebido. O inteiro especifica o número de bytes a ser copiado para o array.
DatagramPacket(byte[], int, InetAddress, int)	Constrói um pacote a ser usado para enviar um datagrama. O conteúdo do array de bytes será enviado para a porta do computador remoto especificada pelo segundo inteiro e pelo InetAddress. O primeiro inteiro especifica o número de bytes a ser enviado.
InetAddress getAddress()	Retorna o InetAddress do pacote.
int getPort()	Retorna a porta do pacote.
byte[] getData()	Retorna os dados do pacote.
int getLength()	Retorna o comprimento do pacote.

As linhas a seguir recebem dados de um socket de datagramas:

```
DatagramPacket packet = new DatagramPacket(new byte[512], 512);
clienteSocket.receive(packet);
```

O construtor correspondente ao pacote precisa saber onde deve colocar os dados recebidos. Um buffer de 512 bytes foi criado e repassado ao construtor como primeiro parâmetro. O segundo parâmetro do construtor foi o tamanho do buffer. O método receive() ficará bloqueado até haver dados disponíveis.

Transmissão de Datagramas

Para transmissão de datagramas é necessário apenas um endereço completo. Os endereços são criados e controlados pelo uso da classe InetAddress. Essa classe não tem construtores públicos, mas tem vários métodos estáticos que podem ser usados com a finalidade de criar uma instância da classe. A lista a seguir mostra os métodos públicos que criam instâncias da classe InetAddress:

```
InetAddress.getByName(String host);
InetAddress.getAllByName(String host);
InetAddress.getLocalHost();
```

Tanto o getByName() quanto o getAllByName() exigem o nome do host de destino. O primeiro apenas retorna a primeira coincidência que encontra. O segundo método é necessário, porque um computador pode ter mais de um endereço. O computador tem um único nome, mas existem várias maneiras de alcançá-lo.

Todos os métodos de criação são identificados como estáticos:

```
InetAddress addr1 = InetAddress.getByName("microchico");
```



```
InetAddress addr2[] = InetAddress.getAllByName("microchico");
InetAddress addr3 = InetAddress.getLocalHost();
```

Qualquer uma dessas chamadas pode emitir uma *UnknownHostException*. Se um computador não estiver conectado a um DNS (*Domain Name Server*) ou se o host realmente não for encontrado, será emitida uma exceção. Se um computador não tiver uma configuração TCP/IP ativa, então provavelmente `getLocalHost()` também irá falhar com essa exceção.

Depois que um endereço é determinado, podem ser enviados datagramas. As linhas a seguir transmitem um String para um socket de destino:

```
String toSend = "Esses são os dados a serem enviados!";
byte[] sendbuf = new byte[toSend.length()];
toSend.getBytes(0, toSend.length(), sendbuf, 0);
DatagramPacket sendPacket = new DatagramPacket(sendbuf, sendbuf.length, addr, port);
clienteSocket.send(sendPacket);
```

Primeiramente, o string tem de ser convertido em um array de bytes. O método `getBytes()` cuida dessa conversão. Em seguida, tem de ser criada uma nova instância de `DatagramPacket`. O endereço e a porta de destino devem ser incluídos no pacote. Um datagrama é semelhante a um envelope, ele tem um endereço de retorno que é incluído automaticamente no pacote enviado. O endereço de retorno pode ser extraído do pacote pelo uso de `getAddress()` e `getPort()`. A seguir uma maneira como um servidor responderia a um pacote do cliente:

```
DatagramPacket sendPacket = new DatagramPacket(sendbuf,
sendbuf.length, recvPacket.getAddress(),
recv.getPort());
ServerSocket.send(sendPacket);
```

Os construtores e métodos da classe `DatagramSocket` usados para construir um servidor de datagramas são mostrados na tabela 4.

Tabela 4. Construtores e Métodos da classe `DatagramSocket` [Thomas 97]

Construtores e Métodos	Descrição
<code>DatagramSocket()</code>	Cria um socket UDP numa porta livre não especificada.
<code>DatagramSocket(int)</code>	Cria um socket UDP na porta especificada.
<code>Receive(DatagramPacket)</code>	Espera para receber um datagrama e copia os dados para o <code>DatagramPacket</code> especificado.
<code>send(DatagramPacket)</code>	Envia um <code>DatagramPacket</code> .
<code>getLocalPort()</code>	Retorna a porta local à qual o socket está ligado.
<code>close()</code>	Fecha o socket.

Servidores de Datagramas

Script básico para servidores de datagramas:

- 1. Criar o socket de datagrama em uma porta específica.
- 2. Chamar o receive, a fim de esperar pacotes que estão chegando.
- 3. Responder aos pacotes recebidos, de acordo com o protocolo combinado.
- 4. Voltar ao Passo 2 ou continuar até o Passo 5.
- 5. Fechar o socket de datagramas.

Clientes de Datagramas

O cliente que corresponde ao servidor de datagramas utiliza o mesmo processo, com uma única exceção: um cliente tem de iniciar a conversação:

- 1. Criar o socket de datagrama em qualquer porta disponível.
- 2. Criar o endereço para envio.
- 3. Transmitir os dados de acordo com o protocolo do servidor.
- 4. Esperar para receber dados.
- 5. Voltar ao Passo 3 (enviar outros dados), 4 (esperar pela recepção) ou 6 (sair).
- 6. Fechar o socket de datagramas.

A figura 4 resume os passos necessários para aplicativos cliente/servidor de datagramas.

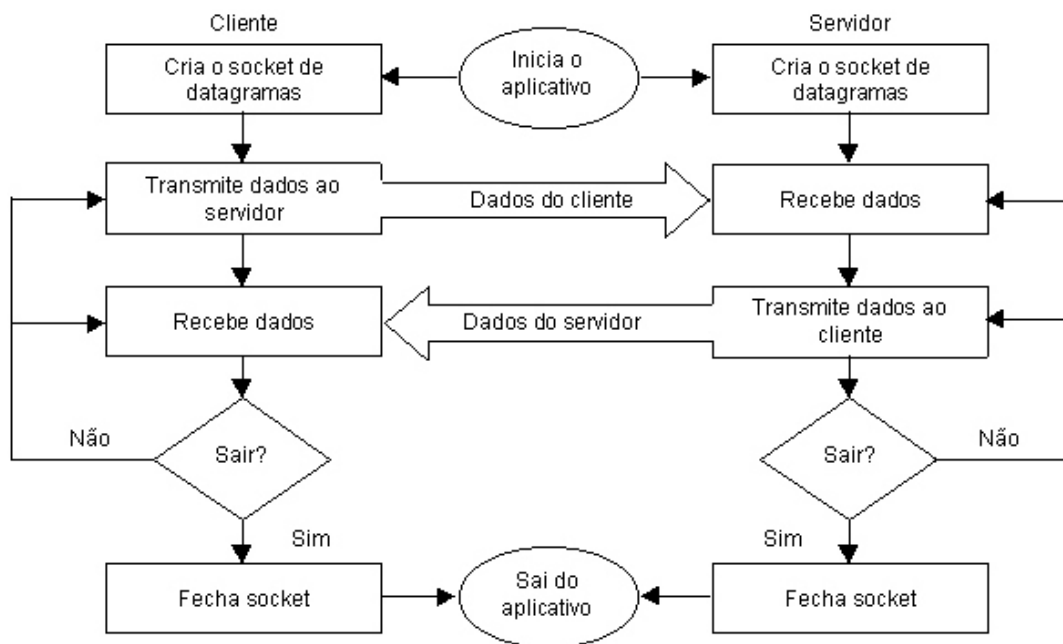


Figura 4. Aplicativo cliente e servidor de datagramas [Hopson 97]

4.1 – Uma Aplicação baseada em Datagramas

Foi desenvolvida uma Aplicação de comunicação de computadores utilizando o protocolo UDP (*User Datagram Protocol*). O servidor datagrama fica ouvindo na porta específica no desenvolvimento da aplicação (4545) e retransmite o datagrama que recebe, ecoando a mensagem. O cliente datagrama envia um datagrama contendo uma mensagem e o recebe de volta do servidor datagrama.

Para ver os arquivos fontes do servidor de eco datagrama e do cliente datagrama, clique no link a seguir:

Fonte do servidor datagrama "ServidorDatagrama.java" e fonte do cliente datagrama "ClienteDatagrama.java".

Para obter os arquivos fontes do servidor de eco datagrama e cliente datagrama: Servidor Datagrama "ServidorDatagrama.java" e cliente datagrama "ClienteDatagrama.java": [\[eco.zip\]](#).

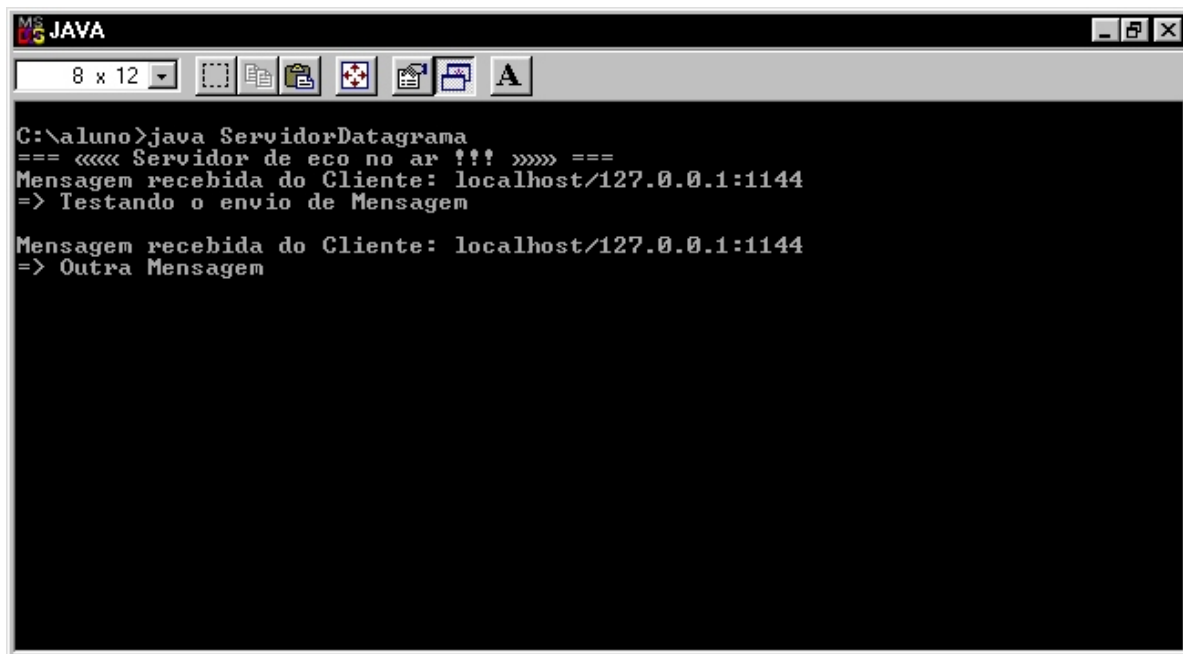
4.1.1 – Executando a Aplicação

Passo 1:

Executando o servidor de eco (servidor datagrama):

- Abra uma janela do Prompt do MS-DOS.
- Direcione o Path do Sistema Operacional para C:\jdk1.2.2\bin ou para outro diretório em que o Java 1.2.2 estiver instalado.
- Compile o arquivo ServidorDatagrama.java para gerar o bytecode (interpretável pela JVM - Java Virtual Machine) usando a seguinte linha de comando: **C:\...>javac -deprecation ServidorDatagrama.java**
- Para executar o servidor de eco datagrama use a seguinte linha de comando: **C:\...>java ServidorDatagrama**

A figura 5 mostra o servidor de eco datagrama no ar recebendo e repassando datagramas de mensagens.



```
MS-DOS JAVA
8 x 12
C:\aluno>java ServidorDatagrama
=== <<<< Servidor de eco no ar !!! >>>> ===
Mensagem recebida do Cliente: localhost/127.0.0.1:1144
=> Testando o envio de Mensagem

Mensagem recebida do Cliente: localhost/127.0.0.1:1144
=> Outra Mensagem
```

Figura 5. Servidor de eco datagrama no ar.

Passo 2:

Executando o cliente datagrama:

- Abra uma janela do Prompt do MS-DOS.
- Direcione o Path do Sistema Operacional para C:\jdk1.2.2\bin ou para outro diretório em que o Java 1.2.2 estiver instalado.
- Compile o arquivo ClienteDatagrama.java para gerar o bytecode (interpretável pela JVM - Java Virtual Machine) usando a seguinte linha de comando: **C:\...>javac -deprecation ClienteDatagrama.java**
- Para executar o cliente datagrama use a seguinte linha de comando: **C:\...>java ClienteDatagrama**
- Para executar outro cliente datagrama repita todas as etapas do **Passo 2**.

A figura 6 mostra um cliente datagrama sendo executado e se comunicando com servidor, enviando e recebendo as mensagens que envia.



```
C:\aluno>java ClienteDatagrama
Mensagem para enviar: Testando o envio de Mensagem
Mensagem recebida (eco): Testando o envio de Mensagem

Mensagem para enviar: Outra Mensagem
Mensagem recebida (eco): Outra Mensagem

Mensagem para enviar:
```

Figura 6. Cliente datagrama se comunicando com o servidor de eco datagrama.

5 – Conclusões

Presentemente existem muitas e diversas aplicações desenvolvidas com a tecnologia de interface de sockets, que é o método mais utilizado para acesso a uma rede TCP/IP. Comunicação de Computadores utilizando Sockets representa um ponto de conexão para uma rede TCP/IP. Para dois computadores manter uma conversação, cada um deles utiliza um socket. Um computador chamado servidor abre um socket e presta atenção às conexões, ouvindo num determinado endereço IP e número de porta. O outro ou os outros computadores chamados de clientes, chamam o socket servidor através do endereço IP e número de porta que foi configurado para iniciar a conexão.

Quando a confiabilidade for necessária, deve-se empregar a operação baseada em conexão utilizando o protocolo TCP, ou seja, tudo que é enviado por um socket é recebido pela outra extremidade da conexão, na mesma ordem que foi transmitido, para isso a aplicação vai ter que pagar um preço. Garantir a sequência e a correção dos dados exige processamento extra e utilização de mais memória, o que pode significar em um overhead adicional reduzindo o tempo de resposta do servidor.

Quando se deseja rapidez e eficiência a operação sem conexão utilizando o protocolo UDP é mais adequado, mas não garantido. Datagramas têm menos overhead do que sockets TCP, pois nenhum controle de fluxo é necessário quando se envia ou recebe um datagrama UDP. Os datagramas UDP também podem ser usados num servidor Java que envie atualizações periódicas para um conjunto de clientes. O servidor teria de fazer menos trabalho e conseqüentemente seria mais rápido, se ele pudesse simplesmente enviar datagramas UDP em vez de ter de iniciar uma conexão TCP para cada cliente [Thomas 97].

Referências Bibliográficas

[Hopson 97] Hopson, K. C e Ingram, Stephen E. *Desenvolvendo Applets com Java*. Editora Campus, 1997. Págs. 335 – 351.

[Thomas 97] Thomas, Michael D., Patel, Pratik R., Hudson, Alan D e Ball, Donald A Jr. *Programando em Java para a Internet*. Makron Books, 1997. Págs. 467 – 489.

[Damasceno 96] Damasceno, Américo Jr. *Aprendendo Java – Programação na Internet*. Editora Érica, 1996. Págs. 217 – 262.