Capítulo

1

Desenvolvendo Aplicações para Comunicação Multimídia em Tempo Real

Daniel G. Costa

Abstract

In real-time multimedia communications, time among media acquisition, its transmission over network and media reproduction at the peers must be low and constant, in order to keep real-time sense and received media quality. Complete multimedia architecture, as SIP, and standards related with the sending of data with time restrictions, as RTP, are available in Internet to support those communications. Developing applications to real-time multimedia communication requires specific programming support, as supplied by Java programming language and some especial libraries, all of them free for use.

Resumo

As comunicações multimídia em tempo real correspondem àquelas onde o tempo entre a captura da mídia a ser utilizada, sua transmissão na rede e a posterior reprodução da mídia nos participantes da comunicação deve ser o menor e mais constante possível, para manter tanto a noção de tempo real quanto a qualidade na reprodução das mídias recebidas. Para suportar essas comunicações, diversos padrões estão disponíveis na Internet, envolvendo desde arquiteturas completas de comunicação, como SIP, a protocolos especializados para carregar mídias codificadas, como RTP. Visando o desenvolvimento de aplicações para comunicação multimídia em tempo, são apresentados recursos da linguagem Java e de algumas bibliotecas especializadas, atendendo assim a demanda por soluções gratuitas de programação.

1.1. Fundamentos das comunicações multimídia na Internet

As comunicações multimídia em tempo real correspondem àquelas onde há a transmissão de mídias numa rede de comunicação, com forte restrição de atraso na recepção dos dados. Inúmeras aplicações são fundamentadas em comunicações em tempo real, tais como áudio e videoconferências, telemedicina, telefonia IP, rádios web, ensino a distância, entre outras [Hersent 2002]. O que há em comum entre essas

aplicações é que o tempo entre a captura da mídia a ser utilizada, sua transmissão na rede e a posterior reprodução da mídia nos participantes da comunicação deve ser o menor e mais constante possível, para manter tanto a noção de tempo real quanto a qualidade da reprodução das mídias recebidas através da infra-estrutura de comunicação disponível.

Nessas comunicações, o princípio básico de operação é a transmissão e a recepção, em tempo real, de informações de mídias codificadas. Para a transmissão de mídias, como áudio e vídeo, que são originalmente analógicas, a mídia deve ser "capturada", digitalizada e codificada seguindo determinado padrão. Após a codificação, o envio da informação pode ser realizado. Na recepção dos pacotes transmitidos, que contém os dados anteriormente codificados, o processo de decodificação e reprodução da informação original pode ser adotado. Para os procedimentos de captura e reprodução da mídia, hardwares especializados são utilizados [Costa 2007].

A digitalização refere-se à representação digital das informações analógicas das mídias, enquanto a codificação está relacionada ao ganho de qualidade global da comunicação com o mínimo de consumo de recursos da rede, utilizando para tanto algoritmos especializados. Esses algoritmos definem o formato em que as informações de áudio e vídeo serão codificadas e, opcionalmente, comprimidas para transmissão na rede, sendo chamados de *codecs* (*enCOder - DECoder*). A banda de rede utilizada por uma comunicação, bem como a qualidade final da mídia reproduzida no *host* destino, são diretamente relacionadas aos *codecs* empregados.

Diversas características interessantes podem estar presentes nos *codecs* para comunicações multimídia em tempo real. Uma dessas características, talvez a mais importante, é a sensibilidade à perda de pacotes. Espera-se que esses algoritmos codifiquem a informação de forma que, mesmo que pacotes sejam perdidos, a informação original possa ser recuperada ainda que parcialmente. O grau de resistência a essas perdas varia entre os *codecs*. O que é certo é que mais complexidade e maior tempo de processamento são necessários para diminuir a sensibilidade à perda de pacotes.

O áudio é uma mídia menos complexa que o vídeo, e, por isso, mais fácil de ser processada. Além disso, as comunicações de áudio utilizam menos banda que as comunicações de vídeo. A codificação de áudio está baseada na amostragem de um sinal de voz ou na segmentação "completa" de um sinal de áudio, como ocorre com a música.

Um vídeo é uma seqüência de imagens que são exibidas rapidamente. Nosso cérebro faz o encadeamento natural dessas imagens, dando a idéia de movimento. Uma característica importante dos *codecs* de vídeo é, portanto, o número de imagens (quadros) que um *codec* "apresenta" por unidade de tempo. Essa medida é geralmente feita em quadros por segundo, ou fps (frames per second). No sistema de televisão, por exemplo, utilizam-se frequentemente vídeos a uma taxa de 30fps. Já no cinema costumam-se utilizar vídeos a uma taxa de 24fps. Para as comunicações multimídia em tempo real, cada *codec* geralmente permite que o número de quadros por segundo seja configurável, tipicamente variando entre 15fps e 30fps. De uma maneira geral, quanto mais quadros por segundo são utilizados por um *codec* numa comunicação, melhor é a qualidade do vídeo recebido, porém maior é a banda de transmissão que ele utiliza. Os

codecs de vídeo utilizam mais banda de comunicação que os *codecs* de áudio, devido, sobretudo, à própria natureza da mídia de vídeo, que contem mais informação que a mídia de áudio.

Em média, numa comunicação multimídia em tempo real, os atrasos devem ser inferiores a 150ms para que a comunicação tenha uma boa qualidade, e entre esse valor e 400ms para que a qualidade seja aceitável. Da mesma forma, a variação do atraso, também chamada de *jitter*, deve ser em média inferior a 50ms para não prejudicar consideravelmente a comunicação. Para o cálculo desses valores, além do atraso de transmissão dos pacotes deve-se considerar o atraso de processamento dos *codecs*. Cada *codec* possui um atraso de processamento característico, que deve ser levado em consideração na escolha da melhor solução para o cenário de comunicação considerado. Além disso, possuem também uma taxa de transmissão característica, atraso de processamento, tamanho médio de pacotes e sensibilidade a perdas, atrasos e *jitter*.

A característica de tempo real deve ser preservada, ou seja, as mídias devem ser transmitidas tão logo elas tenham sido capturadas e codificadas, e o receptor deve recebê-las o quanto antes. Caso haja muito atraso para recebimento da mídia, ela perde seu sentido, e o usuário perde a noção de comunicação "ao vivo". Por exemplo, numa videoconferência, se as pessoas se vêem e se ouvem com atrasos consideráveis, a comunicação como um todo é prejudicada, e talvez seja até impraticável. Essas comunicações, portanto, são ditas como sensíveis ao atraso, uma vez que os dados das mídias devem ser recebidos pelos usuários o mais próximo possível do momento em que foram capturadas e codificadas. Ou seja, esperam-se atrasos reduzidos e constantes. Atrasos grandes prejudicam a natureza de tempo real da comunicação, enquanto que atrasos variáveis prejudicam a continuidade da reprodução da mídia. Nesse ponto, verificamos outra característica marcante das comunicações multimídia, que é a continuidade na transmissão. A operação "padrão" da Internet é dita ser em rajada, uma vez que uma grande quantidade de informações é transmitida e recebida em um instante de tempo, e ainda a utilização da rede nesse momento frequentemente é variável. Por outro lado, as comunicações multimídia são formadas por transmissões contínuas de informações, que geralmente ocupam bastante tempo e, principalmente, utilizam recursos da rede de forma constante.

A pilha de protocolos TCP/IP é voltada à transmissão de dados segundo a idéia do "melhor esforço". Assim, não há, nessa rede, qualquer cumprimento de requisitos de tempo, uma vez que os pacotes possuem todos a mesma importância na Internet [Comer 1998]. Para permitir o atendimento das necessidades operacionais da transmissão de dados isócronos, novos protocolos tiveram de ser desenvolvidos para esse fim. Um desses protocolos, o RTP (Real Time Protocol), constitui-se hoje na base das transmissões multimídias em tempo real na Internet, sendo o padrão adotado em praticamente todas as soluções multimídia baseadas em IP. Trazendo informações de número de seqüência e marcas de tempo, além de não realizar qualquer mecanismo de retransmissão de informações perdidas, o RTP oferece um serviço não encontrado em protocolos de transporte da Internet, como o UDP, o TCP e o SCTP.

Para gerenciar essas comunicações, permitindo a criação, encerramento e controle de conferências multimídia, como áudio e videoconferência, outros tipos de padrões foram desenvolvidos. Entre esses padrões, uma arquitetura que vem tendo bastante destaque, principalmente em aplicações multimídia em tempo real na Internet,

tem como ponto central o protocolo SIP. O SIP oferece controle de comunicações multimídia, podendo ser utilizado em conferências e telefonia IP, apenas para citar alguns exemplos.

Há muitos tipos de comunicação multimídia em tempo real, sobretudo quando consideramos o ambiente formado pela Internet. Com elas, é possível que pessoas em locais geograficamente distantes possam realizar reuniões e encontros como se estivessem no mesmo local. Os benefícios podem ser vistos nas áreas econômica, acadêmica, científica, cultural, entre tantas outras. Para que os cenários de comunicação multimídia em tempo real sejam possíveis, aplicações específicas devem ser utilizadas. As aplicações podem ser baseadas em software ou hardware. O desenvolvimento de aplicações para comunicação multimídia em tempo real é importante, e a demanda é alta e crescente. Existem diversas soluções para essa área. As seções seguintes abordam o desenvolvimento de aplicações para comunicação multimídia em tempo real, através da linguagem Java.

1.2. Java e Multimídia

Java é uma linguagem de programação de uso geral e gratuita desenvolvida pela Sun Microsystens, empresa americana fundada em 1982. Extremamente rica e poderosa, a linguagem Java alcançou grande expressividade no cenário de programação, se tornando uma das principais soluções nessa área. Atualmente, Java se apresenta como uma plataforma de desenvolvimento completa, sendo utilizada na produção de programas corporativos, de jogos on-line, de processamento científico, de programas educativos, de aplicações multimídia e de programas em rede, apenas para citar alguns exemplos [Junior 2007].

A primeira versão oficial da linguagem foi lançada em 1995. Desde então, houve muitos aperfeiçoamentos e adaptações, o que vem culminando em uma linguagem de programação madura e adaptada aos novos problemas da programação moderna [Deitel 2005].

Java é uma linguagem simples e de fácil utilização. Juntamente com sua estrutura principal é oferecido um grande número de pacotes adicionais para diversas funcionalidades. Há pacotes para utilização de janelas gráficas, para operações de entrada e saída, para interação com bando de dados, para programação em rede, para processamento multimídia, entre outras possibilidades de programação [Arnold 2007].

Uma das características marcantes dessa linguagem é a portabilidade. Um programa portável é aquele que necessita de muito pouca ou nenhuma alteração no seu código para poder ser executado em plataformas diferentes. As plataformas atuais, compostas basicamente por hardware e sistema operacional, podem ser bastante diferentes entre si, o que impossibilita, na maioria das vezes, a execução de um mesmo programa em plataformas distintas. Para oferecer portabilidade, Java utiliza processos de compilação e interpretação para a execução de programas. Um programa para ser executado deve ser compilado pelo compilador Java e posteriormente interpretado. O interpretador da linguagem está embutido em um software chamado JVM (Java Virtual Machine). Como há uma JVM específica para cada plataforma, um mesmo programa compilado em Java pode ser executado em plataformas distintas, com resultados praticamente iguais [Horstamann 2004].

Java possui diversos recursos na área de multimídia, considerando tanto o processamento de informações quanto comunicações multimídia em tempo real. Esse suporte é obtido através de um conjunto de API, composto por soluções muitas vezes pouco documentadas [Costa 2008]. Os assuntos abordados nas próximas seções irão facilitar o entendimento das API JMF e JAIN SIP, o que virá a colaborar no aperfeiçoamento das habilidades dos programadores no desenvolvimento de aplicações para comunicação multimídia em tempo real.

Muitos dos exemplos apresentados irão enviar, através da rede, informações seguindo algum protocolo particular. Para facilitar o acompanhamento dessas informações, sugere-se a utilização de programas de tipo *sniffer*, a exemplo do Ethereal e do Analyzer. Outro comentário importante sobre os exemplos é em relação ao objetivo de cada código apresentado, que tem com foco questões didáticas, e não requisitos de desempenho e completude.

1.3. Transmissão multimídia em tempo real

Os requisitos das comunicações multimídia em tempo real requerem soluções específicas para atender a essas exigências. Na Internet, a transmissão de dados desse tipo exige protocolos específicos, como o RTP. Os dados enviados através do RTP formam a base para todas as comunicações multimídia em tempo real na Internet.

1.3.1. Protocolo RTP

O RTP (Real Time Protocol) é um protocolo da Internet utilizado imediatamente acima do protocolo UDP, considerando a modularização através do conceito de camadas presente na arquitetura Internet. Esse é um protocolo simples, destinado a atender a demanda das comunicações multimídia em tempo real. O RTP fornece marcadores de tempo, identificadores de mensagens e informações para sincronização na reprodução de mídias diferentes, como áudio e vídeo.

Um pacote RTP é uma unidade de transmissão consistindo de um cabeçalho fixo, uma lista de fontes de contribuição opcional (para um fluxo "misturado"), um cabeçalho de extensão também opcional e uma área de dados, chamada de *payload*. O cabeçalho RTP contém as informações de tempo (*timestamp*) e as identificações de fluxo (SSRC) e de cada pacote. Ele é seguido opcionalmente por uma lista de fontes contribuintes, ou fontes CSRC. Logo após, pode estar presente ou não um cabeçalho de extensão contendo dados de controle opcionais ou experimentais. O *payload*, a última parte do pacote RTP, contém os dados da comunicação, geralmente áudio e vídeo digitalmente codificado. Como as mídias podem ser codificadas utilizando *codecs* diferentes, um campo no cabeçalho RTP especifica o tipo do *codec* que foi utilizado para codificar aqueles dados, sendo essa informação utilizada na reprodução da mídia no *host* destino. Cada um desses pacotes RTP estará na área de dados de um datagrama UDP, que por sua vez será transmitido através da rede usando diretamente o serviço do protocolo IP.

Para auxiliar o RTP nas comunicações multimídia na Internet foi desenvolvido o protocolo RTCP (Real Time Control Protocol). O RTCP foi especificado para ser um protocolo de controle que auxiliasse o RTP na sua tarefa de transmissão de dados em tempo real. A operação desse protocolo está baseada em diversas mensagens de controle com funções específicas. Essas mensagens são enviadas periodicamente para todos os

participantes de uma sessão RTP usando os mesmos mecanismos de comunicação utilizados para o envio de pacotes RTP.

A RFC 3550 é a especificação oficial relacionada aos protocolos RTP e RTCP. Nesse documento podem ser obtidas mais informações a respeito da estrutura e operação desses protocolos.

Por que o RTP e o RTCP operam obrigatoriamente sobre o UDP?

O TCP é o protocolo de transporte mais usado na Internet. As aplicações utilizadas hoje que necessitam de transmissões confiáveis foram projetadas para usufruir dos recursos oferecidos por esse protocolo. Entretanto, as características de operação do TCP o tornam inadequado para as aplicações usuárias de comunicação multimídia em tempo real. A garantia do serviço de comunicação confiável do TCP, fruto de um mecanismo de retransmissão de informações perdidas, prejudica os dados sensíveis ao tempo [Tanembaum 2003]. Para as comunicações multimídia em tempo real, o tempo é mais importante do que a confiabilidade na transmissão dos dados. Como o UDP é um protocolo não confiável, ele se apresenta como solução mais favorável para esse tipo de comunicação.

O UDP sozinho não resolve os problemas das comunicações multimídia em tempo real, porque esse protocolo não possui informações de tempo nem identificadores de mensagens. As informações de tempo são necessárias para a reprodução correta das mídias nos destinos da comunicação. Já os identificadores de mensagem são úteis para reordenação de pacotes recebidos fora de ordem. Para oferecer essas informações de controle adicionais é utilizado o protocolo RTP em conjunto com o protocolo UDP [Peterson 2004].

1.3.2. Escrevendo programas com JMF

Há dois grandes tipos de comunicações multimídia em tempo real. O primeiro tipo refere-se às comunicações com pouco ou nenhum controle. Nessas comunicações, apresentadas nessa seção, o foco está na transmissão dos dados, que tipicamente ocorre entre um emissor e um ou mais receptor. No segundo tipo de aplicações são usados controles especiais, como mecanismos de abertura e encerramento de comunicações. A seção seguinte traz o protocolo SIP, uma das principais soluções para controle de sessões multimídia. Independente do controle, essas aplicações utilizam diretamente o serviço de transmissão de dados multimídia em tempo real utilizado no primeiro tipo apresentado, ou seja, transmissão de mídias codificadas através do RTP.

Para a transmissão, em tempo real, de mídias codificadas, a API Java Media Framework (JMF) pode se utilizada. A finalidade dessa API está ligada diretamente a utilização das mídia de áudio e vídeo. Captura, processamento, armazenamento e transmissão em tempo real dessas mídias são os principais serviços oferecidos pela JMF [Costa 2008]. Serão apresentadas nessa seção as principais classes e interfaces dessa API que oferecem suporte às comunicações multimídia em tempo real.

Para utilizar os pacotes da JMF é necessário que essa API seja corretamente instalada e configurada. No repositório oficial da JMF, localizado no endereço http://java.sun.com/products/java-media/jmf/, estão disponíveis os arquivos dessa API para as principais plataformas utilizadas atualmente, como Windows e Linux. Nesse repositório podem ser obtidos gratuitamente instaladores que de forma automática

configuram o ambiente Java para utilização da JMF. Para essa seção serão consideradas implementações da JMF com versão igual ou superior a 2.0.

Codecs para comunicação multimídia em tempo real são suportados por essa API. JMF suporta os *codecs* de áudio AIFF, Sun Audio (extensão au), AVI, GSM, MIDI, MP2, MP3, QT (extensão mov), RMF e WAV. Já os *codecs* de vídeo suportados são AVI, MPEG-1, QT, H.261 e H.263.

Uma ferramenta importante é o JMF Registry. Dispositivos de áudio e vídeo, como microfones e webcams, só ficam disponíveis para a JMF se estiverem devidamente registrados nesse software. A Figura 1.1 apresenta uma janela do programa JMF Registry indicando o registro de microfone e webcam.

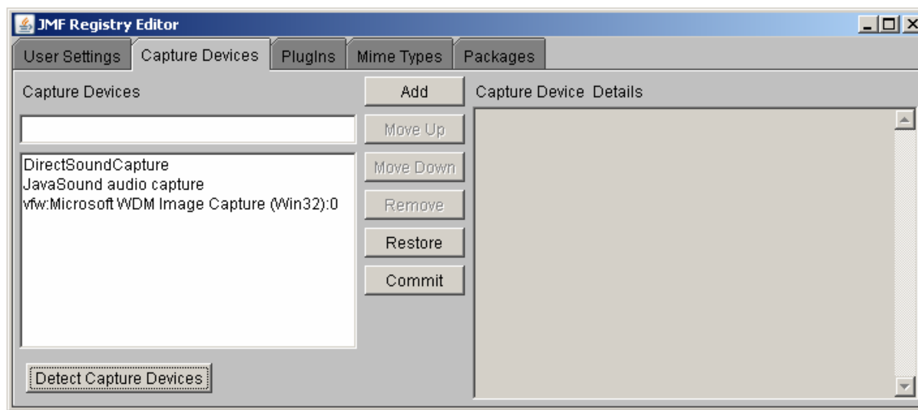

Figura 1.1. JMF Registry.

A JMF especifica uma arquitetura complexa, ligada a várias funções da programação multimídia. Para as comunicações multimídia em tempo real apresentadas nesse capítulo, apenas parte das funcionalidades dessa API será utilizada.

A Tabela 1.1 apresenta as principais classes dessa API. Os pacotes mais utilizados da JMF são javax.media, javax.media.control, javax.media.format e javax.media.protocol.

Tabela 1.1. Principais classes da API JMF.

Classe	Descrição
DataSink	Utilizada para a transmissão de dados codificados.
DataSource	Representa uma fonte de dados.
Manager	Utilizada para a criação de objetos de diversos elementos centrais da JMF.
Player	Responsável por “tocar” a mídia associada.
Processor	Realiza o processamento de uma fonte, formatando-a como configurado.

A Figura 1.2 apresenta um programa que lê um arquivo de áudio e reproduz o seu conteúdo. Para executar esse exemplo, utilize um arquivo de áudio com formato wav. Esse exemplo não considera envio e/ou recebimento de dados pela rede, mas apenas mostra alguns detalhes da API.

```
1.  import javax.media.*;
2.  import java.io.*;
3.  import javax.swing.JOptionPane*;
```

```
4.
5.  public class TocadorDeAudio
6.  {
7.       Player audioPlayer = null;
8.       public TocadorDeAudio (File arquivo)
9.       {
10.           try
11.           {
12.                audioPlayer = Manager.createRealizedPlayer (arquivo.toURI().toURL());
13.           }
14.           catch (Exception exc)
15.           {
16.                System.err.println (exc.toString());
17.           }
18.      }
19.      static public void main (String args[])
20.      {
21.           String entrada = JOptionPane.showInputDialog(null, "Digite o nome do
                   arquivo.");
22.           File arquivo = new File(entrada);
23.           TocadorDeAudio tocar = new TocadorDeAudio (arquivo);
24.           tocar.play();
25.      }
26.      public void play()
27.      {
28.           audioPlayer.start();
29.      }
30.  }
```

Figura 1.2.

Na linha 1 do exemplo é indicado que o pacote javax.media será utilizado pelo código. Esse é o pacote que contém a estrutura básica da API JMF. Na linha 21 é utilizado o método estático showInputDialog() da classe javax.swing.JOptionPane para obter do usuário o nome do arquivo de áudio que será “tocado”. O nome pode ser um caminho completo ou um caminho relativo, considerando para tanto o diretório onde está localizado o arquivo com extensão .class referente ao código compilado do exemplo.

A String digitada pelo usuário é utilizada para criar um objeto File na linha 22. Esse objeto é passado como parâmetro para o construtor da classe TocarAudio. Nesse construtor, o objeto File é utilizado para a criação de um Player, na linha 12. A chamada ao método estático createRealizedPlayer() da classe Manager retorna uma referência do tipo Player. Esse método recebe como parâmetro a URL que indica o arquivo que será reproduzido.

A classe Manager é utilizada para a criação de objetos de diversos elementos centrais da operação da JMF, como Player, DataSource e Processor.

Na linha 28 é chamado o método start() da classe Player. Esse método inicia a reprodução do arquivo de áudio. Para encerrar a reprodução desse arquivo deve-se utilizar o método stop() a partir da mesma referência.

Para transmitir áudio e/ou vídeo em tempo real utilizando a API JMF é necessário que três etapas gerais sejam seguidas. Essas três etapas são a captura, o processamento e o envio dos pacotes.

A captura de mídia está ligada a classe DataSource. Um DataSource representa uma fonte de mídia, que pode ser um arquivo ou um periférico especializado, como um microfone ou uma câmera.

Quando a fonte dos dados a serem transmitidos for um arquivo, pode-se utilizar um objeto da classe MediaLocator para indicar a localização desse arquivo de áudio e/ou de vídeo. A estrutura básica do endereçamento utilizado pela classe MediaLocator é: “protocolo://endereço:porta/tipo”. Nesse tipo de endereçamento, a parte “tipo” geralmente irá referenciar a mídia de áudio ou a mídia de vídeo.

A fonte de dados disponibilizada por um objeto DataSource deve ser processada por um objeto da classe Processor. Esse processamento está ligado principalmente à escolha dos formatos de codificação das mídias e ao encapsulamento RTP.

A saída de um processamento feito por um Processor é um outro objeto DataSource. Esse DataSource representa agora a mídia já codificada e pronta para ser transmitida. O próximo passo é escolher a forma de transmissão desses dados já codificados e formatados. A maneira mais simples de realizar essa transmissão é através de um objeto da classe DataSink. Um objeto dessa classe é retornado numa chamada ao método estático createDataSink() da classe Manager. A esse DataSink são associados um objeto DataSource resultante do processamento feito por um Processor e um endereço para onde os pacotes RTP devem ser enviados. Esse endereço é informado através de um objeto MediaLocator, onde são indicados o endereço IP e a porta UDP de destino.

Outra forma de realizar transmissão de dados através do RTP é utilizando objetos RTPManager. Nessa solução são criadas sessões RTP, o que inclui o envio e recebimento de relatórios RTCP. Essa forma de transmissão, mais completa, porém mais difícil de programar, pode ser encontrada em bibliografia complementar.

A Figura 1.3 apresenta um programa que envia pacotes de áudio em tempo real. Esses pacotes são construídos a partir do arquivo “musica.wav” contido no mesmo diretório que o arquivo .class do exemplo. Apenas como lembrete, alguns formatos de áudio podem não ser bem processados dependendo da versão da API JMF sendo utilizada.

```
1.  import javax.media.*;
2.  import javax.media.control.*;
3.  import javax.media.format.*;
4.  import javax.media.protocol.*;
5.
6.  public class EmissorRTP
7.  {
8.      static String destino = "200.159.32.168:8888";
9.      public static void main(String[] args)
10.     {
11.         new EmissorRTP().iniciarTransmissao (destino, AudioFormat.MPEG_RTP);
12.     }
```

```
13.      public void iniciarTransmissao (String destino, String codec)
14.      {
15.          try
16.          {
17.              String arquivo = "file:musica.wav";
18.              MediaLocator localizacao = null;
19.              DataSource dataSource = null;
20.              Processor processor = null;
21.
22.              localizacao = new MediaLocator (arquivo);
23.              dataSource = Manager.createDataSource(localizacao);
24.              processor = Manager.createProcessor(dataSource);
25.
26.              processor.configure();
27.              while (processor.getState() != Processor.Configured)
28.              {
29.                  Thread.sleep(100);
30.              }
31.
32.              processor.setContentDescriptor(new ContentDescriptor
                     (ContentDescriptor.RAW_RTP));
33.
34.              TrackControl track[] = processor.getTrackControls();
35.              track[0].setFormat(new AudioFormat(codec));
36.
37.              processor.realize();
38.              while (processor.getState() != Processor.Realized)
39.              {
40.                  Thread.sleep(100);
41.              }
42.
43.              DataSource sourceFinal = processor.getDataOutput();
44.
45.              String url = "rtp://" + destino + "/audio";
46.              MediaLocator receptor = new MediaLocator(url);
47.
48.              DataSink sink = Manager.createDataSink(sourceFinal, receptor);
49.              sink.open();
50.              sink.start();
51.              processor.start();
52.          }
53.          catch (Exception exc)
54.          {
55.              System.err.println (exc.toString());
56.              System.exit(1);
57.          }
58.      }
59.  }
```

Figura 1.3.

Nas linhas 1, 2, 3 e 4 são importados pacotes da API JMF necessários para as operações sobre a mídia de áudio nesse exemplo. Na linha 17 é informado o arquivo de

áudio que será a fonte para os pacotes RTP a serem enviados pela rede. Um objeto MediaLocator que irá representar o arquivo musica.wav é criado na linha 22. Em seguida, na linha 23, é utilizado o método createDataSource() da classe Manager para criar um objeto DataSource referente ao arquivo apontado pelo MediaLocator. A partir do método estático createProcessor() da classe Manager é criado um objeto Processor, passando como parâmetro para esse método o objeto DataSource criado anteriormente.

Na linha 26 é iniciada a configuração do objeto Processor. O método configure() pode não ter resultado imediato, de forma que a estrutura de repetição definida entre as linhas 27 e 30 é utilizada para garantir o resultado esperado para a execução desse método. A idéia é que a execução do código irá continuar apenas após a "configuração" correta do Processor. O mesmo é válido para o método realize() dessa classe, chamado na linha 37. Ambos os métodos devem ser chamados seguindo essa ordem e nos dois casos deve-se ter certeza que os métodos configure() e realize() colocaram o Processor no estado correto.

Na linha 32 é utilizado o método setContentDescriptor() da classe Processor para estabelecer o tipo do conteúdo processado. Quatro constantes da classe ContentDescriptor estão disponíveis: RAW, RAW_RTP, MIXED e CONTENT_UNKNOWN. Os dois primeiros formatos estão relacionados à forma de utilização de *buffers*. O terceiro formato corresponde às mídias "misturadas", proveniente, por exemplo, de mais de um DataSource. Já o quarto formato apresentado refere-se a um conteúdo desconhecido. O formato RAW é o mais comum, sendo o padrão para quando não for especificado qualquer formato através do método setContentDescriptor() do Processor. Nesse exemplo, por outro lado, foi especificado o formato RAW_RTP.

O método getTrackControls() retorna um array do tipo TrackControl. Esse array contém a indicação das mídias (trilhas) presentes na fonte. Um arquivo de vídeo típico, como um videoclipe, por exemplo, irá ser formado geralmente por duas mídias: áudio e vídeo. Já um arquivo de áudio "puro", como o associado ao DataSource no exemplo, será formado apenas por uma mídia.

Na linha 35 é estabelecido que a única mídia do arquivo será codificada em MPEG_RTP, um *codec* para comunicações multimídia em tempo real disponível na API JMF. Outros exemplos de *codecs* de áudio são AudioFormat.DVI_RTP, AudioFormat.G723_RTP, AudioFormat.GSM_RTP e AudioFormat.ULAW_RTP.

Na linha 45 é especificado em um objeto String o endereço e a porta para onde o fluxo de áudio será enviado, baseado no endereço fictício especificado na linha 8. É especificado também um descritor do tipo da mídia sendo transmitida (*audio*), que será considerado pelo receptor desses pacotes. Esse objeto String é utilizado para criar um MediaLocator na linha 46.

Para transmitir o áudio já processado e configurado é utilizado nesse exemplo um objeto DataSink. Esse objeto é criado a partir do método estático createDataSink() da classe Manager, tendo como parâmetros um DataSource e um MediaLocator, esse último contendo o endereço e a porta do receptor do fluxo de áudio. O DataSource passado ao método corresponde a fonte da mídia já "configurada" e pronta para ser transmitida.

Para iniciar a transmissão é necessário que os métodos open() e start() da classe DataSink sejam chamados, nessa ordem. Com o objeto DataSink em estado de transmissão é preciso agora que o Processor também seja iniciado. Isso ocorre com a chamada ao método start() na linha 51. Nesse ponto, pacotes RTP contendo o áudio codificado serão enviados para o endereço especificado pelo MediaLocator criado na linha 46. Para encerrar a transmissão a partir de um objeto DataSink, pode-se utilizar o método stop().

A Figura 1.4 apresenta um código para receber o fluxo RTP enviado a partir do programa do exemplo anterior.

```
1.  import javax.media.Manager;
2.  import javax.media.MediaLocator;
3.  import javax.media.Player;
4.
5.  public class ReceptorRTP
6.  {
7.      public static void main(String[] args)
8.      {
9.          try
10.         {
11.             String url = "rtp://200.159.32.168:8888/audio";
12.             MediaLocator localizacao = new MediaLocator(url);
13.
14.             Player player = Manager.createPlayer(localizacao);
15.
16.             player.realize();
17.             while (player.getState() != Player.Realized)
18.             {
19.                 Thread.sleep(50);
20.             }
21.
22.             player.start();
23.         }
24.         catch (Exception exc)
25.         {
26.             System.err.println(exc.toString());
27.             System.exit(1);
28.         }
29.     }
30. }
```

Figura 1.4.

Na linha 11 é especificada uma String que representa um endereço RTP fictício para onde os pacotes contendo áudio codificado estão sendo enviados, no caso o endereço do *host* onde está sendo executado esse exemplo. Esse objeto String é utilizado para criação de um MediaLocator na linha seguinte. Na linha 14, o método createPlayer() da classe Manager cria um objeto Player recebendo como parâmetro um objeto MediaLocator.. Na linha 22 é iniciada a reprodução do áudio recebido através da chamada ao método start() da classe Player.

A Figura 1.5 apresenta um programa que transmite pacotes RTP contendo áudio de maneira semelhante ao exemplo da Figura 1.3. A diferença entre esses exemplos é que agora a fonte dos dados será um periférico de microfone, e não um arquivo de

áudio. Além disso, os pacotes serão enviados agora para um grupo multicast. Com isso, mais de um cliente poderá receber o mesmo áudio de um único emissor.

```
1.  import java.util.Vector;
2.  import javax.media.*;
3.  import javax.media.control.*;
4.  import javax.media.format.*;
5.  import javax.media.protocol.*;
6.
7.  public class EmissorRTP2
8.  {
9.      public static void main(String[] args)
10.     {
11.         try
12.         {
13.             AudioFormat format = new AudioFormat(AudioFormat.LINEAR, 8000, 8,
                    1);
14.
15.             Vector dispositivos = CaptureDeviceManager.getDeviceList(format);
16.
17.             CaptureDeviceInfo info = null;
18.
19.              if (dispositivos.size() > 0)
20.                 info = (CaptureDeviceInfo) dispositivos.elementAt (0);
21.
22.             Processor processor = null;
23.             processor = Manager.createProcessor(info.getLocator());
24.
25.             processor.configure();
26.             while (processor.getState() != Processor.Configured)
27.             {
28.                 Thread.sleep(100);
29.             }
30.
31.             TrackControl track[] = processor.getTrackControls();
32.             track[0].setFormat( new AudioFormat(AudioFormat.GSM_RTP, 8000, 8,1));
33.
34.             processor.realize();
35.             while (processor.getState() != Processor.Realized)
36.              {
37.                 Thread.sleep(100);
38.              }
39.
40.             DataSource sourceFinal = processor.getDataOutput();
41.
42.             String url = "rtp:// 224.200.123.147:5555/audio";
43.             MediaLocator destino = new MediaLocator(url);
44.
45.             DataSink sink = Manager.createDataSink(sourceFinal, destino);
46.             sink.open();
47.             sink.start();
48.             processor.start();
```

```
49.             }
50.             catch (Exception exc)
51.             {
52.                 System.err.println (exc.toString());
53.             }
54.         }
55. }
```

Figura 1.5.

Um objeto AudioFormat é criado na linha 13. Os quatro parâmetros passados ao construtor dessa classe estão relacionados a uma codificação de áudio linear, com amostragem de voz utilizando o algoritmo PCM (freqüência de 8000Hz com 8 bits por amostragem). Esse formato é utilizado na chamada ao método getDeviceList(), da classe CaptureDeviceManager. O método irá retornar um objeto Vector contendo os dispositivos que utilizam dados seguindo a formatação passada como parâmetro. Na maior parte dos casos esse objeto Vector conterá apenas um elemento, correspondente ao microfone utilizado em computadores (geralmente apenas um). Na linha 20 é feita uma conversão explícita entre o tipo desse único elemento e a classe CaptureDeviceInfo. O método getLocator() do objeto dessa classe é utilizado para retornar um objeto MediaLocator, que na linha 23 é usado para a criação de um Processor.

Os outros procedimentos nesse exemplo já foram vistos anteriormente. O que é interessante notar é a especificação de um endereço multicast na String definida na linha 42. As comunicações multimídia em tempo real são diretamente beneficiadas pela utilização da tecnologia multicast. Considerando a natureza dessas comunicações, que são altas consumidoras de banda em relação a outros serviços da Internet, as transmissões multicast podem reduzir potencialmente o uso dos recursos de rede e ainda permitir que vários *hosts* participem de uma comunicação.

Para reproduzir o áudio contido nos pacotes recebidos, basta fazer como no exemplo da Figura 1.4. A diferença aqui está na especificação de um endereço multicast, que deve ser considerado na definição do MediaLocator no receptor.

A Figura 1.6 apresenta um programa que transmite em tempo real pacotes contendo vídeo codificado.

```
1.  import java.util.Vector;
2.  import javax.media.format.*;
3.  import javax.media.protocol.*;
4.  import javax.media.rtp.*;
5.  import javax.media.*;
6.  import javax.media.control.*;
7.
8.  public class EmissorRTPVideo
9.  {
10.     public EmissorRTPVideo()
11.     {
12.         try
13.         {
14.             Vector deviceList = CaptureDeviceManager.getDeviceList(new
```

```
                    VideoFormat(VideoFormat.YUV));
15.             CaptureDeviceInfo device = (CaptureDeviceInfo)deviceList.get(0);
16.             Processor processor = Manager.createProcessor(device.getLocator());
17.
18.             processor.configure();
19.             while (processor.getState() != Processor.Configured)
20.             {
21.                 Thread.sleep(100);
22.             }
23.
24.             processor.setContentDescriptor(new ContentDescriptor
                    (ContentDescriptor.RAW));
25.
26.             TrackControl track[] = processor.getTrackControls();
27.             track[0].setFormat(new VideoFormat (VideoFormat.JPEG_RTP));
28.
29.             processor.realize();
30.             while (processor.getState() != Processor.Realized)
31.             {
32.                 Thread.sleep(100);
33.             }
34.
35.             DataSource sourceFinal = processor.getDataOutput();
36.
37.             String url= "rtp://10.65.97.94:6666/video";
38.             MediaLocator destino = new MediaLocator(url);
39.
40.             DataSink sink = Manager.createDataSink(sourceFinal, destino);
41.             sink.open();
42.             sink.start();
43.             processor.start();
44.         }
45.         catch (Exception exc)
46.         {
47.             exc.printStackTrace();
48.         }
49.     }
50.     public static void main (String args[])
51.     {
52.         new EmissorRTPVideo();
53.     }
54. }
```

Figura 1.6.

Esse programa é semelhante aos emissores RTP anteriormente apresentados As diferenças estão na forma de captura e formatação do vídeo.

Na linha 14 é obtida uma lista com os dispositivos de captura de vídeo. Para um computador com uma webcam comum (USB) instalada, essa lista conterá apenas um elemento. A configuração do formato do vídeo a ser transmitido é feita na linha 27.

A Figura 1.7 apresenta o receptor dos pacotes de vídeo enviados pelo programa do exemplo anterior.

```
1.   import javax.media.*;
2.   import javax.swing.JFrame;
3.
4.   public class ReceptorRTPVideo extends JFrame
5.   {
6.        public static void main (String entrada[])
7.        {
8.             new ReceptorRTPVideo();
9.        }
10.       public ReceptorRTPVideo()
11.       {
12.            try
13.            {
14.                 String url= "rtp://10.65.97.94:5530/video";
15.
16.                 MediaLocator dadosRecebidos = new MediaLocator(url);
17.
18.                 Player player = Manager.createRealizedPlayer(dadosRecebidos);
19.
20.                 player.start();
21.
22.                 setSize (300, 300);
23.                 getContentPane().add (player.getVisualComponent());
24.                 setVisible(true);
25.            }
26.            catch (Exception exc)
27.            {
28.                 exc.printStackTrace();
29.            }
30.       }
31.  }
```

Figura 1.7.

As informações codificadas do vídeo serão exibidas através de um objeto Player associado a uma janela gráfica. A definição da classe ReceptorRTPVideo especifica a utilização da classe JFrame como uma superclasse. Os métodos dessa classe são utilizados para a criação de uma janela gráfica. Essa janela irá conter apenas um elemento, que é especificado na linha 23. Como já foi visto, o método getVisualComponent() retorna um objeto de tipo Component, que pode ser associado, por exemplo, a um JPanel ou a um JFrame. Com isso, a janela gráfica irá apresentar apenas o vídeo recebido. Para aplicações que recebem dados multimídia em tempo real não é interessante que um painel de controle (com stop/pause/play) seja fornecido, como ocorreu nesse exemplo.

Como foi visto nessa seção, para transmitir um fluxo de áudio e/ou vídeo em tempo real é necessário utilizar um objeto Processor para gerar um DataSource codificado para transmissão RTP. A partir desse objeto DataSource é possível iniciar uma transmissão utilizando um objeto DataSink. Outra opção é enviar os pacotes através de um objeto RTPManager. A transmissão de pacotes RTP é a base para todas as comunicações multimídia em tempo real na Internet. Na seção seguinte será

apresentada a arquitetura SIP, a solução de controle de sessões multimídia mais popular da Internet.

1.4. Arquitetura SIP

As comunicações multimídia em tempo real podem ter mais controle do que o simples envio de dados multimídia para um ou mais destino. Por exemplo, pode ser interessante que algum mecanismo para iniciar e encerrar comunicações seja utilizado. É o caso das aplicações de videoconferência e telefonia IP, que precisam explicitamente iniciar e encerrar uma sessão multimídia. Esse controle é uma parte adicional às comunicações multimídia em tempo real, que já utilizam a estrutura RTP/UDP para envio dos dados codificados.

Existem diversas soluções para controle de comunicações multimídia em tempo real. Nessa seção será apresentado o desenvolvimento de aplicações seguindo a arquitetura SIP, considerando para tanto a API JAIN SIP.

1.4.1.Protocolo SIP

O SIP (Session Initiation Protocol) foi especificado pelo grupo de trabalho MMUSIC do IETF, inicialmente com a RFC 2543, publicada em 1999. A versão atual do protocolo SIP, contida na RFC 3261, foi publicada no ano de 2002. Ele é destinado à abertura, controle e encerramento de chamadas em comunicações multimídia em tempo real.

Para o desenvolvimento do protocolo SIP, dois protocolos de aplicação da Internet foram considerados: o HTTP (Hypertext Transfer Protocol) e o SMTP (Simple Mail Transfer Protocol). O SIP utiliza endereços URL, como o protocolo HTTP, e uma estrutura de cabeçalho semelhante a presente em mensagens SMTP. Além disso, o protocolo SIP é estruturado textualmente, como esses dois protocolos.

O protocolo SIP utiliza portas de comunicação pré-definidas para a abertura de comunicações. As portas utilizadas são 5060 e 5061, que podem ser, por padrão, UDP ou TCP. Essa característica do protocolo SIP garante mais flexibilidade à comunicação.

Existem dois tipos de mensagem SIP. O primeiro tipo corresponde às mensagens de requisição. A Tabela 1.2 apresenta as principais mensagens desse tipo, juntamente com suas finalidades.

Tabela 1.2. Principais mensagens SIP.

Mensagem	Descrição
ACK	Confirmar recebimento de mensagem.
BYE	Indicar o encerramento de uma comunicação SIP.
CANCEL	Indicar cancelamento de pedidos.
INVITE	Indicar um pedido de abertura de comunicação SIP.
MESSAGE	Mensagem genérica.
NOTIFY	Informar sobre a ocorrência de algum evento previamente solicitado.
OPTIONS	Requisitar informações de servidores SIP.
REGISTER	Realizar registros e consultas a usuários em servidores Registrar.
SUBSCRIBE	Registrar o usuário junto a elementos SIP especiais.

Além das mensagens de requisição, o SIP especifica também mensagens de resposta. As mensagens SIP de resposta utilizam um código numérico que identifica a situação do pedido feito anteriormente. Cada código está contido em uma faixa geral de

descrição, como mostra a Tabela 1.3. Esse esquema de informação por códigos é semelhante àquele utilizado pelo protocolo HTTP.

Tabela 1.3. Códigos de resposta SIP.

Faixa	Descrição
100 – 199	Ação ou situação temporária.
200 – 299	Indicação de sucesso.
300 – 399	Informação de redirecionamento.
400 – 499	Erro do lado cliente da comunicação.
500 – 599	Erro do lado servidor da comunicação.
600 – 699	Indicação de falha global.

Para iniciar uma comunicação seguindo a arquitetura SIP, uma conexão deve ser estabelecida com o protocolo SIP, sobre TCP ou UDP. As mensagens de requisição utilizadas nesse procedimento são INVITE e ACK. Para as mensagens de resposta, mais de um tipo é possível, como 180 e 200. O pedido de abertura de conexão pode ser originado por qualquer terminal SIP que implemente essa funcionalidade.

Logo após a abertura de uma conexão SIP, os participantes já podem iniciar a troca de pacotes RTP contendo as mídias codificadas.

Para encerramento de uma conexão SIP, mensagens específicas são utilizadas. Uma associação pode ser encerrada a qualquer momento da comunicação. Caso elementos especiais da arquitetura SIP estejam sendo utilizados, trocas de mensagens adicionais podem ser necessárias para o encerramento de uma comunicação SIP.

Uma arquitetura SIP típica é formada por terminais e opcionalmente por servidores *proxies*, de registro e de redirecionamento, além de agentes de notificação e *gateways*. O uso de nenhum desses elementos é obrigatório, com exceção dos terminais SIP. Um terminal é composto por duas partes, que podem ou não estar presentes: a parte UAC (User Agent Client), que realiza solicitações; e a parte UAS (User Agent Server), que responde às solicitações. Nos exemplos nessa seção, estaremos sempre considerando comunicações originadas de um terminal que só implementa a parte UAC para um terminal que só implementa a parte UAS, por questão de simplicidade.

1.4.2. Desenvolvendo aplicações com JAIN SIP

JAIN SIP é uma API Java destinada ao desenvolvimento de aplicações SIP. Essa API oferece uma interface de baixo nível para acesso a estruturas do protocolo, como mensagens. O termo "baixo nível" está ligado ao fato de que, nessa API, detalhes específicos da comunicação SIP devem ser trabalhos pelo programador, tornando grande e potencialmente complexos os códigos escritos através dessa API. Apesar disso, o programador adquire maior controle sobre a comunicação.

Sendo uma API de baixo nível, o programador deve conhecer o máximo possível o protocolo SIP. Por exemplo, para enviar uma mensagem de requisição, é interessante que o programador conheça a estruturas das mensagens SIP desse tipo. Como exemplo, observe a mensagem SIP apresentada a seguir.

INVITE sip: daniel@uefs.br SIP/2.0
Via: SIP/2.0/UDP uefs.br:5060
Max-Forwards: 70
To: Daniel <sip:daniel@uefs.br>
From: Michael Phelps <sip: phelps@usa.com>

Call-ID: 5412365@ uefs.br
CSeq: 1 INVITE
Subject: Comunicações multimídia em tempo real
Contact: <sip:phelps@usa.com>
Content-Type: application/sdp
Content-Length: 160

Para utilizar os pacotes da JAIN SIP é necessário que essa API seja corretamente instalada e configurada. No repositório oficial dessa API, localizado no endereço https://jain-sip.dev.java.net/, estão disponíveis os arquivos dessa API para as principais plataformas utilizadas atualmente, como Windows e Linux. Os pacotes dessa API que serão utilizados são javax.sip, javax.sip.message, javax.sip.address e javax.sip.header.

As principais classes dessa API são apresentadas na Tabela 1.4.

Tabela 1.4. Principais classes da API JAIN SIP.

Classe	Descrição
AddressFactory	Classe utilizada para criação de endereços especiais da API.
HeaderFactory	Classe utilizada para criação de cabeçalhos das mensagens SIP.
ListeningPoint	Cria um ponto de recebimentos de pacote, na porta especificada.
MessageFactory	Classe utilizada para a criação de mensagens.
SipProvider	Utilizada para criação de objetos da API.
SipStack	Representa uma implementação de uma pilha de protocolos SIP.

Os próximos dois exemplos apresentam um cenário muito simples de comunicação SIP. Nesse cenário, um UAC irá enviar uma mensagem SIP do tipo MESSAGE para um UAS. Apenas isso. Dentro dessa mensagem estará contida uma mensagem de texto que será exibida no receptor. A mensagem SIP é montada na Figura 1.8.

```
1.  import javax.sip.*;
2.  import javax.sip.message.*;
3.  import javax.sip.address.*;
4.  import javax.sip.header.*;
5.  import java.util.*;
6.  import javax.swing.JOptionPane;
7.
8.  public class ClienteSIPTexto implements SipListener
9.  {
10.     String endLocal = null;
11.     String endDestino = null;
12.
13.     int portaLocal = 5060;
14.     int portaDestino = null;
15.
16.     String remetente = "Daniel G. Costa";
17.     String remetenteSIP = "danielgcosta@uefs.br";
18.
19.     SipFactory sf;
20.     SipProvider sp;
21.     HeaderFactory headerFactory;
22.     AddressFactory addressFactory;
23.     MessageFactory messageFactory;
```

```
24.
25.    public static void main (String args[])
26.    {
27.         new ClienteSIPTexto();
28.    }
29.    public ClienteSIPTexto ()
30.    {
31.         try
32.         {
33.              endLocal = java.net.InetAddress.getLocalHost().getHostAddress();
34.
35.              sf = SipFactory.getInstance();
36.              sf.setPathName ("gov.nist");
37.
38.              Properties prop = new Properties ();
39.
40.              prop.setProperty ("javax.sip.STACK_NAME", "gov.nist");
41.              prop.setProperty ("javax.sip.IP_ADDRESS", endLocal);
42.
43.              SipStack pilhaSip = sf.createSipStack (prop);
44.
45.              ListeningPoint esperarUdp =  pilhaSip.createListeningPoint (endLocal,
                      portaLocal, "udp");
46.              sp = pilhaSip.createSipProvider (esperarUdp);
47.              sp.addSipListener (this);
48.
49.              headerFactory = sf.createHeaderFactory();
50.              addressFactory = sf.createAddressFactory();
51.              messageFactory = sf.createMessageFactory();
52.
53.              enviarMensagem ();
54.         }
55.         catch (Exception exc)
56.         {
57.              System.out.println (exc.toString());
58.              exc.printStackTrace();
59.              System.exit (-1);
60.         }
61.    }
62.    public void enviarMensagem () throws Exception
63.    {
64.         endDestino = JOptionPane.showInputDialog(null, "Digite o endereço de
                 destino");
65.         portaDestino = Integer.parseInt (JOptionPane.showInputDialog(null, "Digite a
                 porta de destino"));
66.         String mensagem = JOptionPane.showInputDialog(null, "Digite a mensagem a
                 ser enviada");
67.
68.         SipURI fromAddress = addressFactory.createSipURI(remetente, remetenteSIP);
69.         Address fromNameAddress = addressFactory.createAddress(fromAddress);
70.         fromNameAddress.setDisplayName(remetente);
71.         FromHeader fromHeader = headerFactory.createFromHeader
```

```
                    (fromNameAddress, null);
72.
73.         SipURI toAddress = addressFactory.createSipURI(endDestino, endDestino);
74.         Address toNameAddress = addressFactory.createAddress(toAddress);
75.         toNameAddress.setDisplayName(endDestino);
76.         ToHeader toHeader = headerFactory.createToHeader(toNameAddress, null);
77.
78.         SipURI enderecoDestino = addressFactory.createSipURI("destino", endDestino
                + ":" + portaDestino);
79.
80.         ArrayList viaHeaders = new ArrayList();
81.         ViaHeader viaHeader = headerFactory.createViaHeader
                    (endDestino,portaDestino,"udp",null);
82.         viaHeaders.add(viaHeader);
83.
84.         ContentTypeHeader contentTypeHeader =
                headerFactory.createContentTypeHeader("text", "plain");
85.
86.         Request request =  messageFactory.createRequest
                    (enderecoDestino,Request.MESSAGE,
                     sp.getNewCallId(), headerFactory.createCSeqHeader((long)1,
                     Request.MESSAGE),  fromHeader, toHeader, viaHeaders,
                     headerFactory.createMaxForwardsHeader(70));
87.
88.         SipURI contato = addressFactory.createSipURI(remetente, endLocal);
89.
90.         Address contactAddress = addressFactory.createAddress(contato);
91.
92.         ContactHeader contactHeader = headerFactory.createContactHeader
                    (contactAddress);
93.         request.addHeader(contactHeader);
94.         request.setContent(mensagem, contentTypeHeader);
95.
96.         sp.sendRequest(request);
97.
98.         System.out.println ("Mensagem enviada.");
99.      }
100.    public void processDialogTerminated(DialogTerminatedEvent evento)
101.    {
102.    }
103.    public void processTransactionTerminated(TransactionTerminatedEvent evento)
104.    {
105.    }
106.    public void processIOException(IOExceptionEvent evento)
107.    {
108.    }
109.    public void processTimeout(TimeoutEvent evento)
110.    {
111.    }
112.    public void processResponse(ResponseEvent evento)
113.    {
114.    }
```

```
115.    public void processRequest(RequestEvent evento)
116.    {
117.    }
120.}
```

Figura. 1.8.

Nas quatro primeiras linhas são importados quatro pacotes para operação seguindo o protocolo SIP. A implementação seguindo a especificação JAIN SIP começa na linha 35, com a criação de um SipFactory. Um objeto desse tipo é responsável pela criação de outros elementos ligados direta ou indiretamente à comunicação, como ocorre nas linhas 49, 50 e 51. Na linha 36 é estabelecido o diretório base para a API, de acordo com a implementação utilizada. A implementação desenvolvida pelo NIST (National Institute of Standards and Technology) é a mais utilizada.

Para a comunicação poder ocorrer corretamente, uma pilha SIP deve ser utilizada. Essa pilha será a base para toda a comunicação. Um objeto Properties é utilizado para configurar corretamente essa pilha. A partir dessas configurações, um objeto SipStack pode ser criado, como ocorre na linha 43.

Após criar a pilha, devem-se configurar os pontos de recepção de mensagens SIP. Nesses pontos, a aplicação irá "escutar" pelo recebimento de mensagens. Na linha 45, esse ponto é criado, de acordo com o endereço, a porta e o protocolo de transporte especificado. Esse ponto de recepção criado deve ser associado a um objeto SipProvider, que é criado na linha 46.

O objeto HeaderFactory criado na linha 49 será utilizado para a construção de cabeçalhos das mensagens SIP. Já o objeto do tipo AddressFactory estará responsável pela criação de endereços SIP. Por fim, a mensagem como um todo será montada com auxílio de um objeto do tipo MessageFactory.

Após as configurações iniciais, vamos partir para a construção de uma mensagem SIP de tipo MESSAGE.

Um objeto SipURI é criado na linha 68 para representar um endereço SIP de origem, utilizando para tanto um objeto AddressFactory como base. Esse endereço é utilizado para a criação de um endereço de tipo Address, na linha 69, que por sua vez é utilizado para a criação de um objeto FromHeader. Um procedimento semelhante ocorre entre as linhas 73 e 76, mas com o objetivo de criar um objeto de tipo ToHeader. Esses objetos farão parte da mensagem SIP que será criada.

O destino da mensagem SIP é especificado na linha 78. Na criação desse endereço são especificados o endereço IP de origem e a porta.

Entre as linhas 80 e 81 é criado a parte Via da mensagem SIP, que, inclusive, pode indicar mais de um caminho. Em seguida, na linha 84, é especificado o tipo da informação carregada pela mensagem SIP. Uma mensagem SIP pode carregar qualquer informação, ou mesmo nenhuma. Como nesse exemplo o objetivo é que uma mensagem de texto seja enviada para o UAS, o tipo text/plain é especificado.

A mensagem SIP é criada na linha 86, através do método createMessage() do objeto MessageFactory. Esse método recebe oito parâmetros, sendo eles: o endereço de destino em formato SipURI, o tipo da mensagem SIP sendo criado, um valor para o callID (identificação da mensagem), o número de seqüência da mensagem, a indicação

da origem da mensagem (informação de cabeçalho), a indicação do destino da mensagem (informação de cabeçalho) e a parte Via da mensagem.

Depois de criada a mensagem é necessário ainda adicionar um campo de contato. Esse campo é criado entre as linhas 88 e 92, e adicionado à mensagem na linha 93. Para completar, o conteúdo da mensagem, bem como seu tipo, é adicionado na linha 94.

A partir do SipProvider criado anteriormente, a mensagem é enviada para seu destino com a instrução na linha 96.

Na definição da classe ClienteSIPTexto é utilizada a interface SipListener. Todo terminal SIP deve utilizar essa interface, que obriga a classe ClienteSIPTexto a implementar seis métodos, como apresentado na Tabela 1.5.

Tabela 1.5. Métodos da interface SipListener.

Método	Descrição
public void processRequest (RequestEvent)	Chamado com o recebimento de uma mensagem de requisição SIP.
public void processResponse (ResponseEvent)	Chamado com o recebimento de uma mensagem de resposta SIP.
public void processTimeout (TimeoutEvent)	Chamado com notificações de *timeout*.
public void processIOException (IOExceptionEvent)	Chamado quando ocorrerem notificações de IO.
public void processTransactionTerminated (TransactionTerminatedEvent)	Chamado com o encerramento de um Transaction (relacionado com a comunicação SIP).
public void processDialogTerminated (DialogTerminatedEvent)	Chamado com o encerramento de um Dialog (relacionado com a comunicação SIP).

A Figura 1.9 apresenta o UAS que irá receber a mensagem enviada pelo exemplo anterior.

```
1.  import javax.sip.*;
2.  import javax.sip.message.*;
3.  import javax.sip.address.*;
4.  import javax.sip.header.*;
5.  import java.util.*;
6.  public class ServidorSIPTexto implements SipListener
7.  {
8.      SipStack pilhaSip;
9.      SipFactory sf;
10.
11.     static String endLocal;
12.     final static int portaLocal = 5060;
13.
14.     public static void main (String args[])
15.     {
16.         new ServidorSIPTexto();
17.     }
18.     public ServidorSIPTexto ()
19.     {
```

```
20.         try
21.         {
22.             sf = SipFactory.getInstance();
23.             sf.setPathName ("gov.nist");
24.
25.             Properties prop = new Properties ();
26.
27.             endLocal = java.net.InetAddress.getLocalHost().getHostAddress();
28.             prop.setProperty ("javax.sip.STACK_NAME", "gov.nist");
29.
30.             pilhaSip = sf.createSipStack (prop);
31.             ListeningPoint esperarUdp =  pilhaSip.createListeningPoint (endLocal,
                    portaLocal, "udp");
32.
33.             SipProvider sp = pilhaSip.createSipProvider (esperarUdp);
34.             sp.addSipListener (this);
35.             System.out.println ("Aguardando solicitacoes SIP.");
36.         }
37.         catch (Exception exc)
38.         {
39.             System.out.println (exc.toString());
40.             exc.printStackTrace();
41.             System.exit (-1);
42.         }
43.     }
44.     public void processRequest(RequestEvent evento)
45.     {
46.         Request requisicao = evento.getRequest();
47.
48.         System.out.println ("\nTexto recebido: " + new String
                (requisicao.getRawContent()));
49.
50.         System.out.println ("\nTipo da mensagem SIP: " + requisicao.getMethod());
51.
52.         System.out.println ("\nMensagem SIP recebida:\n" + requisicao.toString());
53.     }
54.     public void processDialogTerminated(DialogTerminatedEvent evento)
55.     {
56.     }
57.     public void processTransactionTerminated(TransactionTerminatedEvent evento)
58.     {
59.     }
60.     public void processIOException(IOExceptionEvent evento)
61.     {
62.     }
63.     public void processTimeout(TimeoutEvent evento)
64.     {
65.     }
66.     public void processResponse(ResponseEvent evento)
67.     {
68.     }
69. }
```

Figura 1.9.

A estrutura desse exemplo é semelhante à utilizada no exemplo anterior. Tanto o UAC quanto o UAS devem utilizar a interface SipListener. A diferença aqui é que a parte UAS da comunicação deve implementar o método processRequest(RequestEvent), já que uma solicitação SIP será recebida.

Entre as linhas 22 e 34 são feitas as configurações inicias, de modo semelhante ao exemplo da Figura 1.8. O tratamento da solicitação ocorre ente as linhas 46 e 52. Na linha 46, a requisição é capturada. A partir desse objeto é possível verificar o corpo da mensagem SIP (linha 48), o tipo da mensagem (linha 50) e a mensagem SIP como um todo (linha 52).

A Figura 1.10 apresenta um programa que envia uma mensagem SIP INVITE e apresenta ao usuário a mensagem de resposta. Esse exemplo é para ser usado em conjunto com um terminal SIP completo, que implemente pelo menos a parte UAS.

```
1.  import javax.sip.*;
2.  import javax.sip.message.*;
3.  import javax.sip.address.*;
4.  import javax.sip.header.*;
5.  import java.util.*;
6.
7.  public class TesteSIP implements SipListener
8.  {
9.      String endLocal = null;
10.     String endDestino = null;
11.
12.     int portaLocal = 5060;
13.     int portaDestino = 5060;
14.
15.     String fromName = "Daniel";
16.     String fromDisplayName = "Daniel G. Costa";
17.     String toUser = "Michael Phelps";
18.     String toDisplayName = Michael Phelps";
19.
20.     SipFactory sf;
21.
22.     SipProvider udpP, tcpP, sp;
23.
24.     public static void main (String args[])
25.     {
26.         new TesteSIP();
27.     }
28.     public TesteSIP ()
29.     {
30.         try
31.         {
32.             endLocal = java.net.InetAddress.getLocalHost().getHostAddress();
33.             endDestino = javax.swing.JOptionPane.showInputDialog ("Digite o
                    endereço.", null);
34.
35.             sf = SipFactory.getInstance();
```

```
36.             sf.setPathName ("gov.nist");
37.
38.             Properties prop = new Properties ();
39.
40.             prop.setProperty ("javax.sip.STACK_NAME", "cliente");
41.
42.             SipStack pilhaSip = sf.createSipStack (prop);
43.
44.             ListeningPoint esperarTcp = pilhaSip.createListeningPoint (endLocal,
                    portaLocal, "tcp");
45.             tcpP = pilhaSip.createSipProvider (esperarTcp);
46.             tcpP.addSipListener (this);
47.
48.             ListeningPoint esperarUdp =  pilhaSip.createListeningPoint (endLocal,
                    portaLocal, "udp");
49.             udpP = pilhaSip.createSipProvider (esperarUdp);
50.             udpP.addSipListener (this);
51.
52.             sp = udpP;
53.
54.             enviarMensagem ();
55.         }
56.         catch (Exception exc)
57.         {
58.             System.out.println (exc.toString());
59.             exc.printStackTrace();
60.             System.exit (-1);
61.         }
62.     }
63.     public void enviarMensagem () throws Exception
64.     {
65.         HeaderFactory headerFactory = sf.createHeaderFactory();
66.         AddressFactory addressFactory = sf.createAddressFactory();
67.         MessageFactory messageFactory = sf.createMessageFactory();
68.
69.         SipURI fromAddress = addressFactory.createSipURI(fromName, endLocal + ":"
                + portaLocal);
70.         Address fromNameAddress = addressFactory.createAddress(fromAddress);
71.         fromNameAddress.setDisplayName(fromDisplayName);
72.         FromHeader fromHeader = headerFactory.createFromHeader
                 (fromNameAddress, "clienteteste");

73.         SipURI toAddress = addressFactory.createSipURI(toUser, endDestino + ":" +
                portaDestino);
74.         Address toNameAddress = addressFactory.createAddress(toAddress);
75.         toNameAddress.setDisplayName(toDisplayName);
76.         ToHeader toHeader = headerFactory.createToHeader(toNameAddress, null);
77.
78.         SipURI enderecoDestino = addressFactory.createSipURI(toUser, endDestino +
                ":" + portaDestino);
79.         enderecoDestino.setTransportParam("udp");
80.
```

```
81.         ArrayList viaHeaders = new ArrayList();
82.         ViaHeader viaHeader = headerFactory.createViaHeader
                (endDestino,portaDestino,"udp",null);
83.         viaHeaders.add(viaHeader);
84.
85.         Request request = messageFactory.createRequest
                (enderecoDestino,Request.INVITE, sp.getNewCallId(),
                 headerFactory.createCSeqHeader((long)1, Request.INVITE),
                 fromHeader, toHeader, viaHeaders,
                 headerFactory.createMaxForwardsHeader(70));
86.
87.         SipURI contato = addressFactory.createSipURI(fromName, endLocal);
88.         contato.setPort(portaLocal);
89.
90.         Address contactAddress = addressFactory.createAddress(contato);
91.         contactAddress.setDisplayName(fromName);
92.         ContactHeader contactHeader = headerFactory.createContactHeader
                (contactAddress);
93.
94.         request.addHeader(contactHeader);
95.
96.         String texto = "Esse eh apenas um texto de exemplo.";
97.         ContentTypeHeader contentTypeHeader =
              headerFactory.createContentTypeHeader("text", "plain");
98.
99.         request.setContent(texto.getBytes(), contentTypeHeader);
100.
101.        ClientTransaction con = sp.getNewClientTransaction(request);
102.        con.sendRequest();
103.
104.        System.out.println ("INVITE enviado.");
105.    }
106.    public void processDialogTerminated(DialogTerminatedEvent evento)
107.    {
108.    }
109.    public void processTransactionTerminated(TransactionTerminatedEvent evento)
110.    {
111.    }
112.    public void processIOException(IOExceptionEvent evento)
113.    {
114.    }
115.    public void processTimeout(TimeoutEvent evento)
116.    {
117.    }
118.    public void processResponse(ResponseEvent evento)
119.    {
120.        try
121.        {
122.            Response resposta = evento.getResponse();
123.            System.out.println ("Mensagem recebida: " + resposta.getStatusCode());
124.
125.            byte[] conteudo = resposta.getRawContent();
```

```
126.                if (conteudo != null)
127.                    System.out.println ("Conteudo da mensagem:\n" + new String
                            (conteudo));
128.            }
129.            catch (Exception exc)
130.            {
131.                System.err.println(exc.toString());
132.                exc.printStackTrace();
133.            }
134.    }
135.    public void processRequest(RequestEvent evento)
136.    {
137.    }
137.}
```

Figura 1.10.

Esse exemplo é semelhante ao exemplo 1.7. Uma mensagem é criada e enviada a um destino especificado. A diferença aqui está no tipo da mensagem, que é agora é INVITE. Essa mensagem é enviada para um terminal SIP, que deve responder com alguma mensagem seguindo a especificação do protocolo.

Entre as linhas 119 e 132 é especifico um código para tratar mensagens recebidas. Esse código apresenta para o usuário o código da mensagem de resposta juntamente com o conteúdo dessa mensagem.

Os dois próximos exemplos apresentam um estabelecimento de uma sessão SIP, baseada sobre o protocolo UDP. Esse estabelecimento segue o diagrama especificado na Figura 1.11.

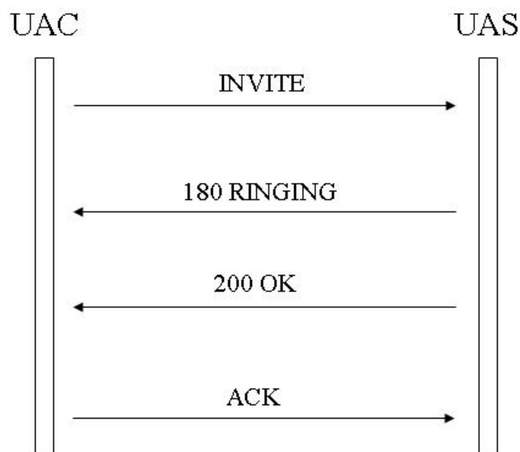

Figura 1.11.

A mensagem de resposta com código 180 corresponde a indicação de que o terminal remoto está “tocando”. Essa é uma mensagem opcional.

O UAC é apresentado na Figura 1.12, enquanto o UAS é apresentado no exemplo da Figura 1.13.

```
1.  import javax.sip.*;
2.  import javax.sip.message.*;
```

```
3.   import javax.sip.address.*;
4.   import javax.sip.header.*;
5.   import java.util.*;
6.
7.   public class ClienteSIPConexao implements SipListener
8.   {
9.        String endLocal = null;
10.       String endDestino = null;
11.
12.       int portaLocal = 5060;
13.       int portaDestino = 5060;
14.
15.       String fromName = "Daniel";
16.       String fromDisplayName = "Daniel G. Costa";
17.       String toUser = "Usain Bolt";
18.       String toDisplayName = "The Flash";
19.
20.       SipFactory sf;
21.
22.       SipProvider udpP, tcpP, sp;
23.
24.       Dialog dialog;
25.
26.       public static void main (String args[])
27.       {
28.            new ClienteSIPConexao();
29.       }
30.       public ClienteSIPConexao ()
31.       {
32.            try
33.            {
34.                 endLocal = java.net.InetAddress.getLocalHost().getHostAddress();
35.                 endDestino = "200.25.36.147";
36.
37.                 sf = SipFactory.getInstance();
38.                 sf.setPathName ("gov.nist");
39.
40.                 Properties prop = new Properties ();
41.
42.                 prop.setProperty ("javax.sip.STACK_NAME", "cliente");
43.
44.                 SipStack pilhaSip = sf.createSipStack (prop);
45.
46.                 ListeningPoint esperarTcp = pilhaSip.createListeningPoint (endLocal,
                         portaLocal, "tcp");
47.                 tcpP = pilhaSip.createSipProvider (esperarTcp);
48.                 tcpP.addSipListener (this);
49.
50.                 ListeningPoint esperarUdp =  pilhaSip.createListeningPoint (endLocal,
                         portaLocal, "udp");
51.                 udpP = pilhaSip.createSipProvider (esperarUdp);
52.                 udpP.addSipListener (this);
```

```
53.
54.                 sp = udpP;
55.
56.                 enviarMensagem ();
57.             }
58.             catch (Exception exc)
59.             {
60.                 System.out.println (exc.toString());
61.                 exc.printStackTrace();
62.                 System.exit (-1);
63.             }
64.         }
65.         public void enviarMensagem () throws Exception
66.         {
67.             HeaderFactory headerFactory = sf.createHeaderFactory();
68.             AddressFactory addressFactory = sf.createAddressFactory();
69.             MessageFactory messageFactory = sf.createMessageFactory();
70.
71.             SipURI fromAddress = addressFactory.createSipURI(fromName, endLocal + ":"
                    + portaLocal);
72.             Address fromNameAddress = addressFactory.createAddress(fromAddress);
73.             fromNameAddress.setDisplayName(fromDisplayName);
74.             FromHeader fromHeader =  headerFactory.createFromHeader
                     (fromNameAddress, "clienteteste");
75.
76.             SipURI toAddress = addressFactory.createSipURI(toUser, endDestino + ":" +
                    portaDestino);
77.             Address toNameAddress = addressFactory.createAddress(toAddress);
78.             toNameAddress.setDisplayName(toDisplayName);
79.             ToHeader toHeader = headerFactory.createToHeader(toNameAddress, null);
80.
81.             SipURI enderecoDestino = addressFactory.createSipURI(toUser, endDestino +
                    ":" + portaDestino);
82.             enderecoDestino.setTransportParam("udp");
83.
84.             ArrayList viaHeaders = new ArrayList();
85.             ViaHeader viaHeader = headerFactory.createViaHeader
                    (endDestino,portaDestino,"udp",null);
86.             viaHeaders.add(viaHeader);
87.
88.             Request request = messageFactory.createRequest
                     (enderecoDestino,Request.INVITE, sp.getNewCallId(),
                    headerFactory.createCSeqHeader((long)1, Request.INVITE),
                    fromHeader, toHeader, viaHeaders,
                    headerFactory.createMaxForwardsHeader(70));
89.
90.             SipURI contato = addressFactory.createSipURI(fromName, endLocal);
91.             contato.setPort(portaLocal);
92.
93.             Address contactAddress = addressFactory.createAddress(contato);
94.             contactAddress.setDisplayName(fromName);
95.             ContactHeader contactHeader = headerFactory.createContactHeader
```

```
                 (contactAddress);
96.
97.          request.addHeader(contactHeader);
98.
99.          String texto = "Esse eh apenas um texto de exemplo.";
100.              ContentTypeHeader contentTypeHeader =
                  headerFactory.createContentTypeHeader("text", "plain");
101.
102.         request.setContent(texto.getBytes(), contentTypeHeader);
103.         ClientTransaction con = sp.getNewClientTransaction(request);
104.         dialog = con.getDialog();
105.         con.sendRequest();
106.         System.out.println ("INVITE enviado.");
107.     }
108.     public void processDialogTerminated(DialogTerminatedEvent evento)
109.     {
110.     }
111.     public void processTransactionTerminated(TransactionTerminatedEvent evento)
112.     {
113.     }
114.     public void processIOException(IOExceptionEvent evento)
115.     {
116.     }
117.     public void processTimeout(TimeoutEvent evento)
118.     {
119.     }
120.     public void processResponse(ResponseEvent evento)
121.     {
122.         try
123.         {
124.             Response resposta = evento.getResponse();
125.             System.out.println ("Mensagem recebida: " + resposta.getStatusCode());
126.
127.             MessageFactory mF = sf.createMessageFactory();
128.             AddressFactory addressFactory = sf.createAddressFactory();
129.             HeaderFactory headerFactory = sf.createHeaderFactory();
130.             if (resposta.getStatusCode() == Response.OK)
131.             {
132.                 Dialog dl = evento.getClientTransaction().getDialog();
133.                 Request ackRequest = dl.createAck(1);
134.                 dl.sendAck(ackRequest);
135.                 System.out.println ("Enviei ACK");
136.             }
137.         }
138.         catch (Exception exc)
139.         {
140.             System.err.println("E agora?: " + exc.toString());
141.             exc.printStackTrace();
142.         }
143.     }
144.     public void processRequest(RequestEvent evento)
145.     {
```

```
146.    }
146.}
```

Figura 1.12.

Esse exemplo aguarda por mensagens SIP na porta 5060, tanto TCP quanto UDP. Porém, para o estabelecimento da comunicação, o protocolo UDP é utilizado (linhas 82 e 85).

Após o envio de uma mensagem INVITE, o programa espera pelo recebimento de mensagens de resposta. Caso uma mensagem de código 200 (sucesso) seja recebida, as instruções entre as linhas 132 e 135 são executadas. Nessas instruções é criada uma mensagem de requisição de tipo ACK, que é retornada para o *host* remoto. Nesse ponto, a comunicação já está estabelecida para o UAC.

O UAS para esse esquema de comunicação é apresentado na Figura 1.13.

```
1.  import javax.sip.*;
2.  import javax.sip.message.*;
3.  import javax.sip.address.*;
4.  import javax.sip.header.*;
5.  import java.util.*;
6.
7.  public class ServidorSIPConexao implements SipListener
8.  {
9.      Dialog dialog;
10.     SipFactory sf;
11.
12.     SipProvider udpP, tcpP, sp;
13.
14.     String endLocal;
15.     final static int portaLocal = 5060;
16.
17.     public static void main (String args[])
18.     {
19.         new ServidorSIPConexao();
20.     }
21.     public ServidorSIPConexao ()
22.     {
23.         try
24.         {
25.             sf = SipFactory.getInstance();
26.             sf.setPathName ("gov.nist");
27.
28.             Properties prop = new Properties ();
29.
30.             endLocal = java.net.InetAddress.getLocalHost().getHostAddress();
31.
32.             prop.setProperty ("javax.sip.STACK_NAME", "gov.nist");
33.             prop.setProperty ("javax.sip.IP_ADDRESS", endLocal);
34.
35.             SipStack pilhaSip = sf.createSipStack (prop);
36.
```

```
37.             ListeningPoint esperarTcp=  pilhaSip.createListeningPoint (endLocal,
                    portaLocal, "tcp");
38.             tcpP = pilhaSip.createSipProvider (esperarTcp);
39.             tcpP.addSipListener (this);
40.
41.             ListeningPoint esperarUdp=  pilhaSip.createListeningPoint (endLocal,
                    portaLocal, "udp");
42.             udpP = pilhaSip.createSipProvider (esperarUdp);
43.             udpP.addSipListener (this);
44.
45.             sp = udpP;
46.
47.             System.out.println ("Estou aguardando conexÃµes.");
48.         }
49.         catch (Exception exc)
50.         {
51.             System.out.println (exc.toString());
52.             exc.printStackTrace();
53.             System.exit (-1);
54.         }
55.     }
56.     public void processDialogTerminated(DialogTerminatedEvent evento)
57.     {
58.     }
59.     public void processTransactionTerminated(TransactionTerminatedEvent evento)
60.     {
61.     }
62.     public void processIOException(IOExceptionEvent evento)
63.     {
64.     }
65.     public void processTimeout(TimeoutEvent evento)
66.     {
67.     }
68.     public void processResponse(ResponseEvent evento)
69.     {
70.     }
71.     public void processRequest(RequestEvent evento)
72.     {
73.         try
74.         {
75.             if (evento.getRequest().getMethod().equals(Request.INVITE))
76.                 processarInvite(evento);
77.             else if (evento.getRequest().getMethod().equals(Request.ACK))
78.                 processarAck(evento);
79.         }
80.         catch (Exception exc)
81.         {
82.             System.err.println ("Erro resposta: " + exc.toString());
83.         }
84.     }
85.     public void processarInvite (RequestEvent evento) throws Exception
86.     {
```

```
87.         Request requisicao = evento.getRequest();
88.         System.out.println ("Mensagem recebida: " + requisicao.getMethod() + " - " +
                requisicao.getSIPVersion());
89.
90.         MessageFactory mF = sf.createMessageFactory();
91.         AddressFactory addressFactory = sf.createAddressFactory();
92.         HeaderFactory headerFactory = sf.createHeaderFactory();
93.
94.         Response resposta = mF.createResponse(180, requisicao);
95.
96.         ToHeader toHeader = (ToHeader)resposta.getHeader(ToHeader.NAME);
97.         toHeader.setTag("9999");
98.
99.         Address address = addressFactory.createAddress("Server <sip:" + endLocal +
                ":" + portaLocal + ">");
100.        ContactHeader contactHeader =
                headerFactory.createContactHeader(address);
101.        resposta.addHeader(contactHeader);
102.
103.        SipProvider provedor = (SipProvider) evento.getSource();
104.        ServerTransaction st = evento.getServerTransaction();
105.        if (st == null)
106.        {
107.            st = provedor.getNewServerTransaction(requisicao);
108.        }
109.        st.sendResponse(resposta);
110.        System.out.println ("180 enviado.");
111.
112.        resposta = mF.createResponse(200, requisicao);
113.
114.        toHeader = (ToHeader)resposta.getHeader(ToHeader.NAME);
115.        toHeader.setTag("9999");
116.
117.        address = addressFactory.createAddress("Server <sip:" + endLocal + ":" +
                portaLocal + ">");
118.        contactHeader = headerFactory.createContactHeader(address);
119.        resposta.addHeader(contactHeader);
120.
121.        st.sendResponse(resposta);
122.        System.out.println ("Resposta enviada.");
123.    }
124.    public void processarAck (RequestEvent evento)
125.    {
126.        System.out.println ("Recebi um ACK");
127.    }
127.}
```

Figura 1.13.

Quando uma mensagem INVITE é recebida, o UAS responde com uma mensagem 180 (opcional), indicando que o terminal está tocando, e, em seguida, com uma mensagem de código 200, indicando aceitação da solicitação de início de comunicação. A partir daí é esperada uma mensagem ACK. Como apenas requisições

são recebidas por esse exemplo, apenas o método processRequest (RequestEvent) é implementado. Nesse método, o código entre as linhas 75 e 78 identifica o tipo da requisição, tratando-a corretamente através de métodos auxiliares.

Foi apresentando nos dois exemplos anteriores um mecanismo de iniciação de sessões multimídia através do SIP. A abertura de comunicação é parte importante, porém de nada adianta esse processo se a transmissão de dados multimídia em tempo real não ocorrer. Para isso, um protocolo especial é utilizado, o SDP.

O protocolo SDP (RFC 2327), obrigatório para a arquitetura SIP, é utilizado para descrever sessões multimídia. Com o SDP, aplicações participantes de comunicações com suporte multimídia podem trocar informações sobre suas capacidades de processamento de mídia, permitindo assim compatibilizar os *codecs* a serem empregados nas comunicações. As informações transmitidas pelo protocolo SDP trafegam encapsuladas em mensagens SIP.

Algumas das informações presentes nas mensagens SDP são endereços IP, portas UDP ou TCP utilizadas, *codecs* suportados, título e objetivo da sessão multimídia, informações de contatos, entre outras. Todas essas informações são disponibilizadas textualmente seguindo a especificação desse protocolo.

Uma mensagem SDP é formada por um conjunto de campos. A Tabela 1.6 apresenta alguns desses campos, juntamente com sua finalidade.

Tabela 1.6. Principais campos das mensagens SDP.

Parâmetro	Descrição
a	Atributos da mídia.
c	Informações de controle da comunicação.
m	Descrições da mídia.
o	Identificador de quem está iniciando a comunicação.
s	Assunto/nome da sessão.
t	Tempo de início e fim da sessão.
v	Versão do protocolo SDP.

Os próximos exemplos apresentam um UAC que abre uma conexão com um UAS, a fim de enviar pacotes de áudio em tempo real provenientes de um microfone. Nesses exemplos, a mensagem de requisição INVITE e a mensagem de resposta 200 carregam informações SDP, cada uma referente ao seu respectivo emissor.

```
1.  import javax.sip.*;
2.  import javax.sip.message.*;
3.  import javax.sip.address.*;
4.  import javax.sip.header.*;
5.  import java.util.*;
6.  import javax.sdp.*;
7.  import javax.swing.JOptionPane;
8.
9.  public class ClienteSIPMedia implements SipListener
10. {
11.     String endLocal = null;
12.     String endDestino = null;
13.
14.     int portaLocal = 5060;
```

```
15.     int portaDestino = 5060;
16.     int  portaPacotesAudio = 9999;
17.
18.     String fromName = "Daniel";
19.     String fromDisplayName = "Daniel G. Costa";
20.     String toUser = "Cesar Cielo";
21.     String toDisplayName = "Cesao";
22.
23.     SipFactory sf;
24.
25.     SipProvider udpP, tcpP, sp;
26.
27.     HeaderFactory headerFactory;
28.     AddressFactory addressFactory;
29.     MessageFactory messageFactory;
30.
31.     public static void main (String args[])
32.     {
33.         new ClienteSIPMedia();
34.     }
35.     public ClienteSIPMedia ()
36.     {
37.         try
38.         {
39.             endLocal = java.net.InetAddress.getLocalHost().getHostAddress();
40.             endDestino = JOptionPane.showInputDialog ("Digite o endereço do UAS",
                    null);
41.
42.             sf = SipFactory.getInstance();
43.             sf.setPathName ("gov.nist");
44.
45.             Properties prop = new Properties ();
46.
47.             prop.setProperty ("javax.sip.STACK_NAME", "cliente");
48.
49.             SipStack pilhaSip = sf.createSipStack (prop);
50.
51.             ListeningPoint esperarTcp = pilhaSip.createListeningPoint (endLocal,
                    portaLocal, "tcp");
52.             tcpP = pilhaSip.createSipProvider (esperarTcp);
53.             tcpP.addSipListener (this);
54.
55.             ListeningPoint esperarUdp =  pilhaSip.createListeningPoint (endLocal,
                    portaLocal, "udp");
56.             udpP = pilhaSip.createSipProvider (esperarUdp);
57.             udpP.addSipListener (this);
58.
59.             headerFactory = sf.createHeaderFactory();
60.             addressFactory = sf.createAddressFactory();
61.             messageFactory = sf.createMessageFactory();
62.
63.             sp = udpP;
```

```
64.
65.             enviarMensagem ();
66.         }
67.         catch (Exception exc)
68.         {
69.             System.out.println (exc.toString());
70.             exc.printStackTrace();
71.             System.exit (-1);
72.         }
73.     }
74.     public void enviarMensagem () throws Exception
75.     {
76.         SipURI fromAddress = addressFactory.createSipURI(fromName, endLocal + ":"
                + portaLocal);
77.         Address fromNameAddress = addressFactory.createAddress(fromAddress);
78.         fromNameAddress.setDisplayName(fromDisplayName);
79.         FromHeader fromHeader = headerFactory.createFromHeader
                 (fromNameAddress, "clientemedia");
80.
81.         SipURI toAddress = addressFactory.createSipURI(toUser, endDestino + ":" +
                portaDestino);
82.         Address toNameAddress = addressFactory.createAddress(toAddress);
83.         toNameAddress.setDisplayName(toDisplayName);
84.         ToHeader toHeader = headerFactory.createToHeader(toNameAddress, null);
85.
86.         SipURI enderecoDestino = addressFactory.createSipURI(toUser, endDestino +
                ":" + portaDestino);
87.         enderecoDestino.setTransportParam("udp");
88.
89.         ArrayList viaHeaders = new ArrayList();
90.         ViaHeader viaHeader = headerFactory.createViaHeader
                 (endDestino,portaDestino,"udp",null);
91.         viaHeaders.add(viaHeader);
92.
93.         Request request = messageFactory.createRequest
                 (enderecoDestino,Request.INVITE,
                sp.getNewCallId(), headerFactory.createCSeqHeader((long)1,
                Request.INVITE), fromHeader, toHeader, viaHeaders,
                headerFactory.createMaxForwardsHeader(70));
94.
95.         SipURI contato = addressFactory.createSipURI(fromName, endLocal);
96.         contato.setPort(portaLocal);
97.
98.         Address contactAddress = addressFactory.createAddress(contato);
99.         contactAddress.setDisplayName(fromName);
100.        ContactHeader contactHeader = headerFactory.createContactHeader
                 (contactAddress);
101.
102.        request.addHeader(contactHeader);
103.        ContentTypeHeader contentTypeHeader =
                headerFactory.createContentTypeHeader("application", "sdp");
104.
```

```
105.        SessionDescription sdp = criarSDP ();
106.        request.setContent(sdp, contentTypeHeader);
107.
108.        ClientTransaction con = sp.getNewClientTransaction(request);
109.        con.sendRequest();
110.
111.        System.out.println ("INVITE enviado.");
112.    }
113.    public SessionDescription criarSDP () throws Exception
114.    {
115.        SdpFactory sdpF = SdpFactory.getInstance();
116.        SessionDescription sessionDesc = sdpF.createSessionDescription();
117.        Vector medias = sessionDesc.getMediaDescriptions(true);
118.        int codecs [] = {0};
119.
120.        MediaDescription mDesc = sdpF.createMediaDescription("audio",9999, 1,
                "RTP/AVP", codecs);
121.        Vector atributos = mDesc.getAttributes(true);
122.        atributos.add (sdpF.createAttribute("rtpmap", "0 pcmu/8000/1"));
123.        mDesc.setAttributes(atributos);
124.
125.        medias.add(mDesc);
126.        sessionDesc.setMediaDescriptions(medias);
127.
128.        return sessionDesc;
129.    }
130.    public void processDialogTerminated(DialogTerminatedEvent evento)
131.    {
132.    }
133.    public void processTransactionTerminated(TransactionTerminatedEvent evento)
134.    {
135.    }
136.    public void processIOException(IOExceptionEvent evento)
137.    {
138.    }
139.    public void processTimeout(TimeoutEvent evento)
140.    {
141.    }
142.    public void processResponse(ResponseEvent evento)
143.    {
144.        try
145.        {
146.            Response resposta = evento.getResponse();
147.
148.            if (resposta.getStatusCode() == Response.OK)
149.            {
150.                Dialog dl = evento.getClientTransaction().getDialog();
151.                Request ackRequest = dl.createAck(1);
152.                dl.sendAck(ackRequest);
153.                System.out.println ("Enviei ACK");
154.
155.                new EmissorRTPSIP().iniciarTransmissao(endDestino + ":" +
```

```
                    portaPacotesAudio, "audio");
156.            }
157.        }
158.        catch (Exception exc)
159.        {
160.            System.err.println(exc.toString());
161.            exc.printStackTrace();
162.        }
163.    }
164.    public void processRequest(RequestEvent evento)
165.    {
166.    }
167.}
```

Figura 1.14.

A classe ClienteSIPMedia representa um SIP UAC, que irá enviar uma mensagem INVITE e uma mensagem ACK, e irá aguardar por uma mensagem de resposta 200. Dentro da mensagem INVITE estará presente uma mensagem SDP, construída com auxílio de classes do pacote javax.sdp. Essa não é uma restrição, uma vez que, assim como o SIP, o protocolo SDP utiliza mensagens de texto puro. Dessa forma, a mensagem pode ser montada diretamente pelo programador através dos objetos String, caso desejado.

A mensagem SDP vai especificar, para o *host* remoto, o tipo de informação que pode ser recebida pelo emissor dessa mensagem. Entre as informações tipicamente contidas nessa mensagem estão a porta de recepção, os *codecs* aceitos e o tipo da mídia (geralmente áudio ou vídeo).

O método definido entre as linhas 113 e 129 é utilizado para criar uma mensagem SDP. Assim como a classe SipFactory é utilizada para criação de objetos auxiliares para a construção de mensagens SIP, a classe SdpFactory tem funcionalidade semelhante, porém ligada a criação de mensagens SDP. Entre os objetos que podem ser criados, SessionDescription está relacionado diretamente com o conceito de mensagem SDP. Na linha 120 são especificados o tipo da mídia, a porta que espera-se receber dados em tempo real e o único *codec* suportado. Essa informação será "anexada" à mensagem SDP.

Após estabelecer a comunicação SIP, o programa inicia a transmissão, utilizando para tanto a classe definida na Figura 1.15.

```
1.  import java.util.Vector;
2.  import javax.media.*;
3.  import javax.media.control.*;
4.  import javax.media.format.*;
5.  import javax.media.protocol.*;
6.
7.  public class EmissorRTPSIP
8.  {
9.      public void iniciarTransmissao (String endereco, String tipo)
10.     {
11.         try
12.         {
```

```
13.             AudioFormat format = new AudioFormat(AudioFormat.LINEAR, 8000, 8,
                    1);
14.             Vector dispositivos = CaptureDeviceManager.getDeviceList(format);
15.
16.             CaptureDeviceInfo info = null;
17.
18.             if (dispositivos.size() > 0)
19.                 info = (CaptureDeviceInfo) dispositivos.elementAt (0);
20.
21.             Processor processor = null;
22.             processor = Manager.createProcessor(info.getLocator());
23.
24.             processor.configure();
25.             while (processor.getState() != Processor.Configured)
26.             {
27.                 Thread.sleep(100);
28.             }
29.
30.             TrackControl track[] = processor.getTrackControls();
31.             track[0].setFormat( new AudioFormat(AudioFormat.ULAW_RTP, 8000,
                    8,1));
32.
33.             processor.realize();
34.             while (processor.getState() != Processor.Realized)
35.             {
36.                 Thread.sleep(100);
37.             }
38.
39.             DataSource sourceFinal = processor.getDataOutput();
40.
41.             String url = "rtp://" + endereco + "/" + tipo;
42.             MediaLocator destino = new MediaLocator(url);
43.
44.             DataSink sink = Manager.createDataSink(sourceFinal, destino);
45.             sink.open();
46.             sink.start();
47.             processor.start();
48.         }
49.         catch (Exception exc)
50.         {
51.             System.err.println (exc.toString());
52.             System.exit(1);
53.         }
54.     }
55. }
```

Figura 1.15.

Esse é um esquema simples de comunicação. Apenas o programa da Figura 1.14 irá enviar informações, ficando a parte remota responsável apenas pala recepção desses dados. Sendo assim, a informação SDP enviada no INVITE, nesse exemplo, não tem utilidade, já que o UAC não receberá informações. Essa característica é válida, é claro, apenas nesse exemplo!

O UAS irá enviar também sua informação SDP, através da mensagem de resposta SIP de código 200. Essa mensagem conterá a informação da porta de recepção que será utilizada e do *codec* suportado. A partir dessas informações, o programa da Figura 1.14 poderá enviar suas informações de forma correta.

Há uma curiosidade interessante nesses exemplos. A classe apresentada na Figura 1.15 especifica o *codec* como sendo ULAW_RTP, que é equivalente à informação SDP recebida. A questão é que essa informação não é considerada na hora da definição do *codec*. Isso porque, no receptor, a identificação do *codec* é automática: objetos da classe Player automaticamente identificam o *codec* utilizado. Para tornar o exemplo mais completo, uma possibilidade é verificar os *codecs* suportados antes da escolha do codificador para os dados que serão enviados.

A Figura 1.16 apresenta a parte SIP UAS desse esquema de comunicação. Esse programa recebe uma mensagem INVITE, responde com uma mensagem 200 OK e espera pelo recebimento de uma mensagem ACK. Após o estabelecimento da comunicação SIP, esse programa inicia o processo de recebimento das mídias codificadas.

```
1.  import javax.sip.*;
2.  import javax.sip.message.*;
3.  import javax.sip.address.*;
4.  import javax.sip.header.*;
5.  import java.util.*;
6.  import javax.sdp.*;
7.
8.  public class ServidorSIPMedia implements SipListener
9.  {
10.     SipFactory sf;
11.     SipProvider udpP, tcpP, sp;
12.
13.     String endLocal;
14.     final static int portaLocal = 5060;
15.
16.     MessageFactory messageFactory;
17.     AddressFactory addressFactory;
18.     HeaderFactory headerFactory;
19.
20.     SessionDescription sdpRemoto;
21.
22.     public static void main (String args[])
23.     {
24.         new ServidorSIPMedia();
25.     }
26.     public ServidorSIPMedia ()
27.     {
28.         try
29.         {
30.             sf = SipFactory.getInstance();
31.             sf.setPathName ("gov.nist");
32.
33.             Properties prop = new Properties ();
```

```
34.                 endLocal = java.net.InetAddress.getLocalHost().getHostAddress();
35.
36.                 prop.setProperty ("javax.sip.STACK_NAME", "gov.nist");
37.                 prop.setProperty ("javax.sip.IP_ADDRESS", endLocal);
38.
39.                 SipStack pilhaSip = sf.createSipStack (prop);
40.
41.                 ListeningPoint esperarTcp=  pilhaSip.createListeningPoint (endLocal,
                        portaLocal, "tcp");
42.                 tcpP = pilhaSip.createSipProvider (esperarTcp);
43.                 tcpP.addSipListener (this);
44.
45.                 ListeningPoint esperarUdp=  pilhaSip.createListeningPoint (endLocal,
                        portaLocal, "udp");
46.                 udpP = pilhaSip.createSipProvider (esperarUdp);
47.                 udpP.addSipListener (this);
48.
49.                 sp = udpP;
50.
51.                 messageFactory = sf.createMessageFactory();
52.                 addressFactory = sf.createAddressFactory();
53.                 headerFactory = sf.createHeaderFactory();
54.
55.                 System.out.println ("Estou aguardando conexões.");
56.             }
57.             catch (Exception exc)
58.             {
59.                 System.out.println (exc.toString());
60.                 exc.printStackTrace();
61.                 System.exit (-1);
62.             }
63.         }
64.         public void processDialogTerminated(DialogTerminatedEvent evento)
65.         {
66.         }
67.         public void processTransactionTerminated(TransactionTerminatedEvent evento)
68.         {
69.         }
70.         public void processIOException(IOExceptionEvent evento)
71.         {
72.         }
73.         public void processTimeout(TimeoutEvent evento)
74.         {
75.         }
76.         public void processResponse(ResponseEvent evento)
77.         {
78.         }
79.         public void processRequest(RequestEvent evento)
80.         {
81.             try
82.             {
83.                 if (evento.getRequest().getMethod().equals(Request.INVITE))
```

```
84.                     processarInvite(evento);
85.                 else if (evento.getRequest().getMethod().equals(Request.ACK))
86.                     processarAck(evento);
87.             }
88.             catch (Exception exc)
89.             {
90.                 System.err.println (exc.toString());
91.                 exc.printStackTrace();
92.             }
93.         }
94.         public void processarInvite (RequestEvent evento) throws Exception
95.         {
96.             Request requisicao = evento.getRequest();
97.             System.out.println ("Mensagem recebida: " + requisicao.getMethod() + " - " +
                    requisicao.getSIPVersion());
98.
99.             //Retirar e processar a mensagem SDP
100.            byte [] recebido = requisicao.getRawContent();
101.            SdpFactory sdpF = SdpFactory.getInstance();
102.            sdpRemoto = sdpF.createSessionDescription(new String (recebido));
103.
104.            Response resposta = messageFactory.createResponse(200, requisicao);
105.
106.            ToHeader toHeader = (ToHeader)resposta.getHeader(ToHeader.NAME);
107.            toHeader.setTag("9999");
108.
109.            Address address = addressFactory.createAddress("Server <sip:" + endLocal +
                    ":" + portaLocal + ">");
110.            ContactHeader contactHeader = headerFactory.createContactHeader(address);
111.            resposta.addHeader(contactHeader);
112.
113.            ContentTypeHeader content = headerFactory.createContentTypeHeader
                    ("application", "sdp");
114.            SessionDescription sdpEnviar = (SessionDescription) sdpRemoto.clone()
115.            resposta.setContent(sdpEnviar, content);
116.
117.            SipProvider provedor = (SipProvider) evento.getSource();
118.            ServerTransaction st = evento.getServerTransaction();
119.            if (st == null)
120.            {
121.                st = provedor.getNewServerTransaction(requisicao);
122.            }
123.            st.sendResponse(resposta);
124.            System.out.println ("200 OK enviado.");
125.        }
126.        public void processarAck (RequestEvent evento) throws Exception
127.        {
128.            System.out.println ("Recebi um ACK");
129.            String endereco = null;
130.            int porta = 0;
131.            String tipo = null;
132.
```

```
133.         endereco = sdpRemoto.getOrigin().getAddress();
134.
135.         Vector medias = sdpRemoto.getMediaDescriptions(true);
136.         Enumeration enu = medias.elements();
137.         while (enu.hasMoreElements())
138.         {
139.             MediaDescription md = (MediaDescription)enu.nextElement();
140.             porta = md.getMedia().getMediaPort();
141.             tipo = md.getMedia().getMediaType();
142.         }
143.         new ReceptorRTPSIP().iniciarRecepcao(endereco + ":" + porta, tipo);
144.     }
145.}
```

Figura 1.16.

Na linha 102 é criado um objeto do tipo SessionDescription. Esse objeto contém as informações SDP que estavam dentro da mensagem INVITE recebida. Informações desse objeto são recuperadas nas 133 e 135. A partir dessas informações é possível iniciar o processo de recebimento e reprodução dos pacotes de áudio recebidos.

A Figura 1.17 apresenta uma classe especialmente definida para o recebimento de pacotes de áudio em tempo real. O método iniciarRecepcao(String, String) recebe o endereço de origem (IP + porta) e o tipo da mídia a ser recebida. Essas informações são utilizadas na criação de um objeto DataSink.

```
1.  import javax.media.Manager;
2.  import javax.media.MediaLocator;
3.  import javax.media.Player;
4.
5.  public class ReceptorRTPSIP
6.  {
7.      public void iniciarRecepcao (String origem, String tipo)
8.      {
9.          try
10.         {
11.             String url = "rtp://" + origem + "/" + tipo;
12.             MediaLocator localizacao = new MediaLocator(url);
13.
14.             Player player = Manager.createPlayer(localizacao);
15.
16.             player.realize();
17.             while (player.getState() != Player.Realized)
18.             {
19.                 Thread.sleep(50);
20.             }
21.
22.             System.out.println ("Iniciando recepção de pacotes de aúdio");
23.             player.start();
24.         }
25.         catch (Exception exc)
26.         {
```

```
27.                 System.err.println(exc.toString());
28.                 System.exit(1);
29.             }
30.         }
31. }
```

Figura 1.17.

1.5. Referências bibliográficas

Arnold, Ken. e Gosling, James. A Linguagem de Programação Java. 4ª ed. Editora Bookman. 2007. 799p.

Bodoff, Stephanie, e Green, Dale. Tutorial do J2EE. 1ª ed. Editora Campus.2002. 464p.

Comer, D. E. Interligação em Redes com TCP/IP, Volume 1. 2ªed. Editora Campus. 1998. 672p.

Costa, Daniel G. Comunicações Multimídia na Internet: da Teoria à Prática. 1ª ed. Editora Ciência Moderna. 2007. 256p.

Costa, Daniel G. Java em Rede – Programação Distribuída na Internet. 1ª ed. Editora Brasport. 2008. 312p.

Deitel, H. M. e DEITEL, P. J. Java Como Programar. 6ª ed. Editora Prentice-Hall. 2005. 1152p.

Flanagan, David. Java: O Guia Essencial. 5ª ed. Ediora Bookman. 2006. 1099p.

Hersent, O., Guide, D. e Petit, J-P. Telefonia IP: Comunicação Multimídia Baseada em Pacotes. 2ª ed. Editora Makron Books. 2002. 451p.

Horstmann, Cay. Big Java. 1ª ed. Editora Bookman. 2004. 1128p.

Junior, Peter. Java: Guia do Programador. 1ª ed. Editora Novatec. 2007. 68p.

Kurose James F. e Ross, Keith W. Redes de Computadores e a Internet. 3ª ed. Editora Addison-Wesley. 2006. 656p.

Mathews, Jeanna. Redes de Computadores: Protocolos de Internet em Ação. 1ª ed. Editora LTC. 2006. 224p.

Peterson, Larry L. e Davie, Bruce S. Redes de Computadores: Uma Abordagem de Sistemas. 3ª ed. Editora Campus. 2004. 700p.

Sampaio, Cleuton. SOA e Web Services em Java. 1ª ed. Editora Brasport. 2006. 168p.

Sierra, Kathy. e Bates, Bert. SCJP: Certificação SUN para Programador Java 5 – Guia de Estudo. 2ª ed. Editora Alta Books. 2006. 452p.

Tanembaum, A. S. Redes de Computadores. 4ª ed. Editora Campus. 2003, 946p.