Java Media Framework

プログラミングガイド

2003 年 10 月 23 日

東邦大学理学部情報科学科
古谷　立美

Java Media Framework
プログラミングガイド

目次

2003 年 10 月 23 日 （初版）

東邦大学理学部情報科学科
古谷 立美

I　はじめに

JMFは、Javaで映像や音声を扱うためのAPI群である。1997年ごろにJMF1.0が配布され、その後改良が加えられ（紆余曲折を経て）現在JMF2.1.1が普及している。大変便利な機能を有しているため部分的には便利に利用されているようであるが、メディアサポートが不充分だというような理由から本格的利用は敬遠されているという話もある。残念なことに、全体像を説明する日本語の書物や記事があまり見当たらない。そのためJMFをマスターしようとすると説明が不充分な英語のガイドを頼りに、ソースプログラムを読まねばならないケースが少なくない。筆者はJavaの勉強を兼ね、興味に任せてJMFを勉強し代表的なプログラムを作成してみた。この経験が皆様のお役に立てる機会があればと思い、まとめてみた。

JMFに関する情報は全て以下のサイトからはじまる。

http://java.sun.com/products/java-media/jmf/index.html

またダウンロードは以下から可能である。

http://java.sun.com/aboutJava/communityprocess/maintenance/JMF2.0

ダウンロードやインストールは国内にも紹介サイトがあるので参考にされたい。

ここで紹介するプログラムは筆者がWindoew98SE上でJ2DK1.4.1_03（JMF2.1.1）上で動作確認をしているが、それ以外のOSやJDKでは動作確認をしていない。そのため、他の環境では修正が必要な部分が発生するかもしれない。

ここで紹介するプログラムの多くは、ネットワーク上に公開されていたものを参考に、大胆に単純化し、本質部分だけを抽出した（多くのプログラムはオリジナルが何か分からないと思う）。しかし、オリジナルのプログラム法を引きずっている部分もある。プログラム形態を統一しようかとも考えたが、様々なプログラム法があることが分かるようあえて統一しなかった。多分初心者には参考になると思う。という、筆者自身Java習得途上のため、不適切な部分があれば、ご指摘いただければ幸いである。

II　JMFの概要

2.１ JMFとは何か？
JMFは、Javaで映像や音声といったマルチメディア操作プログラムを簡単にプログラムできる道具である。また、マルチキャスト通信機能を提供し、マルチメディアのストリーミング方式の通信プログラムも簡単に実現できる。JavaのSoundAPIはよく知られているが、JMFはその高レベル版になっている（JMFのF 即ちFrameworkは、低レベルの命令を使って組み立てた高レベルの命令群を指す）。また、JMFはOSごとに、それに適したものが用意されているため、高速処理が可能になっていることも特徴の一つである。

2.2 JMFの基本機能
JMFの目指しているものは次の通りである。
・プログラムが容易
・音声や映像のキャプチャー
・音声や映像の再生
・音声や映像の保存
・Javaによるメディアストリーミングやネットワーク会議システムの開発基盤
・全てJavaによるストリームサーバ、クライアントの開発基盤
・プラグインCODECの導入（独自のCODEC開発も可能）

2.3.　JMFを使うと簡単に書けると言うマルチメディアプログラムとはどんなものか？
JMFで利用できる音声・映像の再生装置はプレーヤとプロセッサである。この２つを使って実現できるシステム例をあげてみる。
（１） プレーヤを使って実現できるシステム
プレーヤは再生専用装置で、図 2.1 のような構成（あるいは、その一部）が考えられる。例えば、
・ハードディスクに保存されている音楽ファイルの再生
・カメラやマイクからキャプチャーされた映像・音声実時間表示
等である。
ここで、注意して欲しい事は、プレーヤは出力をスピーカーとモニタ画面にしか送り出せない（キャプチャーされた音声・画像をファイルに格納できない)、ということである。

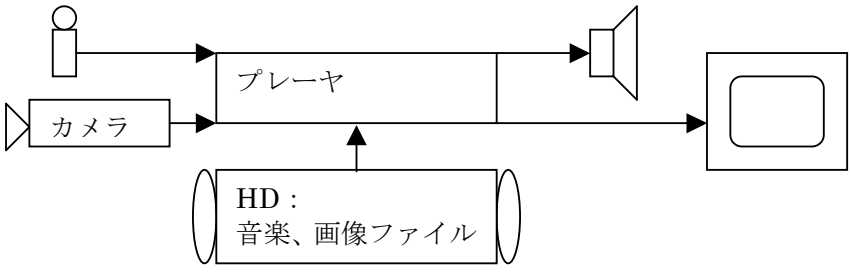


図 2.1　メディアプレーヤ

（2） プロセッサを用いて実現できるシステム
JMF ガイドによればプロセッサはプレーヤの高級版と紹介されている。プレーヤとプロセッサの違いは２つある。
＊第１は、プロセッサはスピーカーとモニタ画面以外の場所に出力ができる点である。これにより、キャプチャーした音声・映像をファイルに保存できたり、キャプチャーした音声・映像、やハードディスクに保存されている音声・映像ファイルの中身をネットワークに送り出すことができる。
図 2.2 は、この機能を使って実現できるインターネットを介した映像音声送受信システムである。特に、JMF では RTP というプロトコルをサポートしており、複雑な通信形態が実現できる。図 2.2 では紙面の都合上、インターネットにプロセッサを２台しか接続していないが、実際には多数のプロセッサを接続した高機能な通信形態が可能となる。例えば、プロセッサ２台ではテレビ電話であるが、複数のプロセッサを付ければビデオ会議が可能となる。また、1 台の送信用プロセッサと多数の受信用プレーヤを接続すればインターネット放送が可能となる。

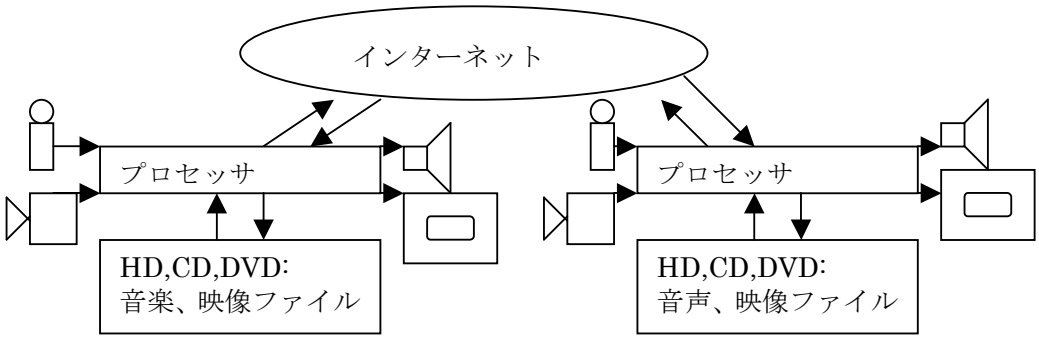


図 2.2　音声・映像通信

＊第２は、プロセッサ内でストリームを分解でき、それぞれに対して信号処理が可能になっている点である。図 2.3 はプロセッサ内部で音声と映像のトラックを分離し、それぞれのトラックに別の信号処理を施している様子を示している。分解されたトラックは出口で再び統合されて出力されている。

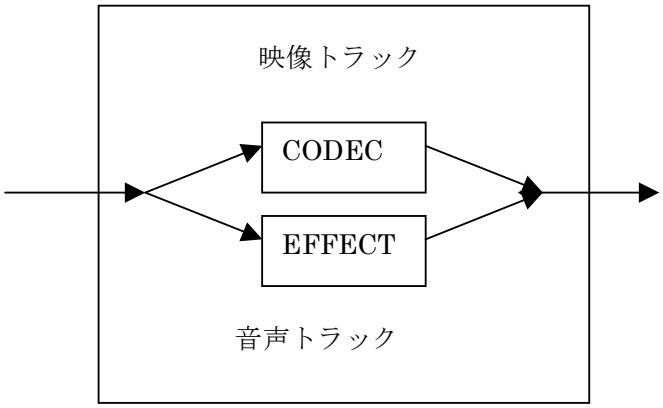


図 2.3　プロセッサ

図 2.3 では上のトラックが映像トラックで CODEC という信号処理が施され、下の音声トラックには EFFECT という信号処理が施されている。CODEC とは符号化-復号化を表し、圧縮・伸張処理などが行われる。EFFECT は音量調整などで、入力と出力のフォーマットが同じものである。

JMF を使うと、プレーヤ（player）とプロセッサ（processor）を簡単に生成（createPlayer という命令と createProcessor という命令を使うことにより）でき、それらを組み合わせて様々なシステムを構築できる。以下、JMF を用いた典型的なマルチメディアシステムを、具体例を使って紹介する。

III. プレーヤによる音声・映像の再生
ここでは、プレーヤを用いた音声・映像の再生を簡単な所から順を追って紹介する。本章は JMF プログラムの基本となる所なので、かなり丁寧に話を進める。

3.1 簡単なプレーヤによる音声ファイルの再生
最初の例は最も単純なもので、q.mp3 という mp3 形式に圧縮された音楽ファイルを再生するものである（図 3.1 参照)。(mp3 フォーマットをサポートしていない JMF（OS によっては）もあるようなので、要確認)

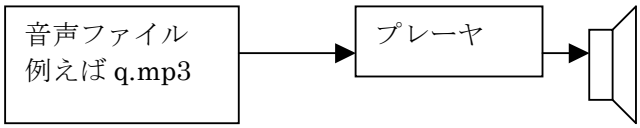


図 3.1 音声ファイルの再生

次に示すプログラム（プログラム１）は、コンソールアプリケーション（MS-DOS プロンプトからプログラムを走らせ、Java のウインドウを開かずに処理を行う）でファイル名（このプログラムでは q.mp3）で指定された音楽を１曲だけ再生する。また、ここでは話の本質を分かりやすくするため、プレーヤ終了はきちんと処理していない(MS-DOS プロンプトで Ctrl-C キーを打たないと終わらない)。

プログラムの説明に入る前に JMF の基本をいくつか紹介する必要がある。
＊ プレーヤの生成法：ここでのプログラムは図 3.1 の構成を作り上げることになる。そのため、最も重要な作業はプレーヤの生成である。JMF でプレーヤの生成には Manager.createRealizedPlayer()という命令を使う。プレーヤやプロセッサは Manager と呼ばれるものの管理下にあり、Manager を最初に付ける。また、Manager.createPlayer という方法もあるが、処理が複雑になるため、VI 章までは Manager.createRealizedPlayer を用いる。

プレーヤ生成命令：Manager.createRealizedPlayer(s); s：プレーヤへの音声・映像入力源（URL|MadeLocator|DataSource）

＊ ファイルの登録とプレーヤへの接続：予め用意した音楽ファイル（以下の例では q.mp3）をオブジェクトとして登録し、プレーヤに接続する。プレーヤへのデータの入力法は①URL、②メディアロケータ③データソースの３つの方法がある。例えて言えば、3 種類の接続ケーブルがある、というようなものである。これについては 3.4 節で紹介するが、当面は URL という方式で接続するものとする。プログラム１は URL という方式（接続ケーブル）でファイルの中身をプレーヤに送っている。
＊ プレーヤの起動：プレーヤへの再生開始指令（メソッド）は start()である。

これから多数のプログラム例が出てくるが、プログラムの本質を見易くするため例外処理はで

きるだけ簡単にしている。
それでは q.mp3 という音楽ファイルを用意し、プログラム１をコンパイル（javac Play.java）して実行（java Play）しよう。

プログラム１：簡単なプレーヤ作成プログラム

```
import javax.media.*;---①
import java.net.*;
import java.io.*;

public class Play{
     public static void main( String[] args ) {
        try {
            File file= new File("q.mp3");---②
            URL url= file.toURL();---③
            Player player = Manager.createRealizedPlayer( url );---④
            player.start();---⑤
       } catch ( Throwable t) {}
    }
}
```

<プログラムの説明>
JMF を利用するには先ず、①の javax を import する必要がある。これに続き２つの import があるが、それぞれ URL 用、File 用である。②では再生ファイルを file という名のオブジェクトにして、③でプレーヤに送り込むための変換（ファイルを接続ケーブルにつなぐ）を行っている。④はプレーヤを生成すると共に、引数としてファイルの出力を入力している（プレーヤへのケーブル接続に対応)。⑤の player.start はプレーヤの再生ボタンを押したことに相当する。

3.2. コントローラ付き音楽再生プレーヤ
MS-DOS プロンプトの下で音楽を再生して終了するのは一寸さびしいので、一般のメディアプレーヤにあるような（図 3.2 参照)、音楽の再生状況をリアルタイムで表示したり、“つまみ”を操作して“再生点を指定したり”というコントロルコンポーネントが付いたプレーヤを次に製作する。これを実現するには Java のウインドウを開く必要があるが、このウインドウは後の映像の表示にも利用される。ウインドウの生成には Java の標準 Frame を用いてもよいが、ちらつきの扱いなどが簡単な Swing の JFrame を用いる。Swing とは Java に OS と関係無く共通のグラフィックユーザインタフェースを提供するものである。

図 3.2 コントロールコンポーネント

プログラムの解説に入る前にいくつかの予備知識を示す。

＊ プレーヤには、再生の状態を制御する図 3.2 のようなコントロルコンポーネント（controlPanelComponent）と、映像を表示するビジュアルコンポーネント（visualComponent）が用意されており、それぞれ、getControlPanelComponent()メソッドと getVisualComponent()で取り出すことができる。JFrame の場合、取り出したコンポーネントを JFrame のパネルに貼りつけることにより、可視化する。
＊ Swing では Frame の代わりに JFrame と呼ばれるフレームが利用され、コントローラ等のコンポーネントは ContentPane と呼ばれる土台の上に配置される。その配置法も Frame とは若干異なる。

プログラム２：コントローラ付き音楽再生プレーヤ

```
import javax.media.*;
import javax.swing.*;---①
import java.awt.*;
import java.io.*;
import java.net.*;

public class Playc extends JFrame {
    public static void main(String[] args) {
       try{ JFrame frame= new JFrame();
            File file= new File("q.mp3");
            URL url= file.toURL();
            Player player = Manager.createRealizedPlayer( url );
            Component cp = player.getControlPanelComponent();---②
            frame.getContentPane().add(cp);---③
            frame.setVisible(true);---④
            player.start();
        } catch (Throwable t) {}
    }
}
```

＜プログラムの解説＞
Swing を使うために新たに①以下 2 行が加わる。②でプレーヤからコントロールコンポーネントを取り出し、③でウィンドウ（フレーム）に貼りつけている。④はフレームを可視化する。

3.3. 音声・映像プレーヤの作成
JMF のプレーヤは優れもので、ファイルの拡張子を見て、自動的にどんな形式のファイルでも再生する。ここでは、音声付きの映像（ファイル名 Q.mpg）を再生するプレーヤを作成する。映像付きでも基本的には、画像表示コンポーネントを追加するだけで済む。プログラムを使用するには、Q.mpg という映像ファイルを用意し、コンパイルし（javac Playm.java）、実行（java Playm）する。

プログラム３：映像音声再生プレーヤ

```
import javax.media.*;
import javax.swing.*;
```

```
import java.awt.*;
import java.io.*;
import java.net.*;

public class Playm extends JFrame {
   public static void main(String[] args) {
      try { JFrame frame= new JFrame();
            File file= new File("Q.mpg");
            URL url= file.toURL();
            Player player = Manager.createRealizedPlayer( url );
            Component vc = player.getVisualComponent();---①
            frame.getContentPane().add( vc, "Center" );---②
            Component cp = player.getControlPanelComponent();
            frame.getContentPane().add( cp, "South" );
            frame.pack();---③
            frame.setVisible(true);
            player.start();
        } catch (Throwable t) {}
    }
}
```

＜プログラムの解説＞音楽再生プレーヤ（プログラム２）との違いは映像表示コンポーネント（VisualComponent）を付け加えているだけ①②である。③は表示サイズの調整用である。

3.4. CCD カメラからのキャプチャー映像の表示

今までの例は、ファイルに保存されている音声や映像を URL 形式でプレーヤの送り込んで再生するというものであった。本節では、カメラやマイクからのキャプチャーした映像・音声をプレーヤで実時間表示する応用である。本節では、２つの新しい事柄を紹介する。先ず第１はプレーヤへの入力形式の話である、第２は再生の終了法である。

3.4.1 プレーヤへの入力形式

プログラム１で q.mp3 という音楽ファイルを再生する場合、ファイルをいったん URL 形式にしてプレーヤに送っていた。いっそ Player player = Manager.createPlayer(“q.mp3”);でよいと思うが、そうはなっていない。これは様々な形態の映像や音楽を統一した方法で扱おうとしているためである。JMF ではプレーヤに入力する３つの形式を用意している。プレーヤへの入力とは、Manager.createRealizedPlayer に与える引数のことで、映像や音声信号をプレーヤに送るための接続（結線）と考えることができる。イメージ的には３種類のケーブル（接続電線）が使える、と言ってもよい。

３種類の接続（線）とは次のものである。

（１） URL 形式: インターネットでおなじみの URL である。すなわち、再生ファイルのありかを URL 形式で指定するもので、プログラム１ではこれが用いられている。プログラム１では②で file という名のファイルを用意し、③の toURL()メソッドで URL 形式に直し、Manager.createPlayer に与えている。筆者のシステムでは、q.mp3 がハードディスク C の中の j2sdk1.4.1_03 の中の jmf フォルダーに入っている。そこで、その URL 形式を出力してみると、file:/C:/j2sdk1.4.1_03/jmf/q.mp3 となっていた。

（２） MediaLocator 形式:これも URL 同様、再生ファイルやキャプチャーデバイスの存在場所を指定するものである。これは URL を含み、さらに広い様々なものが指定できるようになっている。URL 形式を MediaLocator 形式に直すことは可能である。その場合はプログラム１は、

```
File file= newFile( “q.mp3” );
MediaLocator locator= new MediaLocator( file.toURL());
Manager.createRealizedPlayer(locator);
```

となる。
この場合も、（１）同様に “q.mp3” の locator を見てみると URL と同じであった。
次項で改めて示すが、筆者のシステムでのキャプチャーデバイスのメディアロケータは CCD カメラが vfw://0、マイクが dsound://と javasound://44100 であった。

（３） DataSource 形式：これも JMF 専用の形式であるが、上記２つの形式が音声映像の存在場所を示すのに対し、この DataSource はデータの形式を指定する。これも（２）の場合と同様、MediaLocator 形式から Manager.createDataSource()メソッドで DataSource 形式に変換することができる。これは主にプロセッサの方で使われる。

3.4.2 キャプチャーデバイスのメディアロケータ名の調査

筆者は、映像用カメラに IO データ社の CCD カメラを用い（注１）、収音にはマイクをビデオボードに接続している。本節のテーマであるキャプチャー映像・音声の表示には、カメラとマイクの識別名（メディアロケータ名）が必要になる。プログラムで必要になるメディアロケータ名を、プログラム本体の中で調べて利用することもできるが、プログラムの本質が見えにくくなるため、ここではメディアロケータ名の調査を別のプログラムに切り出した。それがプログラム４である。

プログラム４：キャプチャデバイスの MediaLocator 名を求めるプログラム

```
import java.util.*;
import javax.media.*;
import javax.media.format.*;
public class CapDev {
    public static void main( String[] args ) {
        Vector vlist = CaptureDeviceManager.getDeviceList(
                    new VideoFormat( VideoFormat.YUV ) );---①
        for( int i=0; i<vlist.size(); i++ ) {
            CaptureDeviceInfo vinfo
              = (CaptureDeviceInfo)vlist.elementAt( i );
            MediaLocator camera = vinfo.getLocator();
            System.out.println( camera.toString() );---②
        }
        Vector alist = CaptureDeviceManager.getDeviceList(
                   new AudioFormat( AudioFormat.LINEAR ) );---③
        for( int i=0; i<alist.size(); i++ ) {
            CaptureDeviceInfo ainfo
              = (CaptureDeviceInfo)alist.elementAt( i );
```

```
            MediaLocator mic = ainfo.getLocator();
            System.out.println( mic.toString() );---④
        }
    }
}
```

＜プログラムの解説＞①では VideoFormat.YUV という形式のキャプチャーデバイスの一覧表を vlist に取得する。②では取得結果を表示する。③では AudioFormat.LINEAR という形式のキャプチャーデバイスの一覧表を alist に取得し、④で表示する。
筆者のシステムでは、CCD カメラが vfw://0、マイクが dsound://と javasound://44100 と表示された。そこで以下のプログラムでは、これらのメディアロケータ名でデバイスを指定する。

（注１）：CCD カメラを使用する場合、カメラを JMF に登録する作業が必要になる。これには、スタートボタン->プログラム->JavaMediaFramework->JMFRegistry と辿って、表示される画面から Capture Devices タグのページを表示し DetectCaptureDevices ボタンをクリックする（処理完了に少し時間がかかる)。

3.4.3　キャプチャーしたデータの表示
今までのプログラムでは音声再生終了時にコントロールコンポーネントのクローズボタンを押しても DOS プロンプトが返って来ず、Control-C で強制終了を行っていた。ここではクローズボタンを押した時完全に終了するようプログラムを行う。終了処理をきちんと行うことにした理由は CCD カメラからの映像を使った場合、きちんと終了処理をしないと、筆者のシステムではウインドウズがダウンしてしまったためである。

（易しい予備知識）再生用ウインドウを閉じた時に終了処理を行うには、ウィンドウを閉じるイベントを捕らえて、プレーヤの停止とクローズを実施する。具体的にはウインドウを閉じたというイベントを捉えるリッスナーという機能をウインドウに付加することと、そのイベントが発生した時のイベント処理部が必要になる。

プログラムの構造はファイルの再生と大変よく似ている。プレーヤに与える入力が異なるだけである。これが、メディアロケータを導入し入力の扱いを統一した所以である。

プログラム５：CCD カメラ画像の表示

```
import javax.media.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class Caps extends JFrame {
    Player player = null;
    Component vc;
    Component cp;
```

```
    public Caps() {
      try {  ClosingListener CL = new ClosingListener(); ---①
            this.addWindowListener(CL);  ---②
            MediaLocator locator=new MediaLocator("vfw://0");---③
// マイク音声  MediaLocator locator=new MediaLocator("javasound://44100");
// マイク音声  MediaLocator locator=new MediaLocator("dsound://");
            player = Manager.createRealizedPlayer( locator );---④
            vc = player.getVisualComponent();
            if ( vc != null) ---⑤
                 getContentPane().add( vc, "Center" );
            cp = player.getControlPanelComponent();
            if ( cp != null) ---⑥
                 getContentPane().add( cp, "South" );
            pack();
            setVisible(true);
            player.start();
        } catch (Throwable t) {}
    }
   class ClosingListener extends WindowAdapter {---⑦
       public void windowClosing(WindowEvent e) {
        player.stop(); player.close();dispose();System.exit(0); ---⑧
       }
   }
    public static void main(String[] args) {
            Caps frame = new Caps(); ---⑨
    }
}
```

（カメラ映像でなくマイク音声をスピーカーに出したい時は上記コメント文を活かす。但し、カメラと音声を同時に１つのプレーヤでは再生できない）

＜プログラムの解説＞ウインドウが閉じた事を検出するリッスナーは①②でウインドウに付加される。イベントが発生した時働くルーチンは⑦であり、⑧でプレーヤの終了と DOS プロンプトの復活を行っている。また、これらを可能にするため⑨でインスタンスを生成している。③はカメラでのキャプチャーを指定している。④はプレーヤの入力にメディアロケータ形式を用いている。⑤⑥は音声か映像かによって使用するコンポーネントを変えるための分岐である。

＜プログラムの使い方＞
コンパイル（javac Caps.java）し、実行（java Caps)。

IV　音声・映像のファイルへの保存（プロセッサの利用）

図 2.1（ファイルからプレーヤへの入力矢印は有るが、ファイルへの出力矢印はない）から分かる通り、プレーヤはスピーカーとモニタへしか出力できない再生専用装置である。これに対しプロセッサは情報をデータとして他の装置に送り出すことができる。図 4.1 のはプロセッサを用いた音声・映像ファイルの保存を示している。

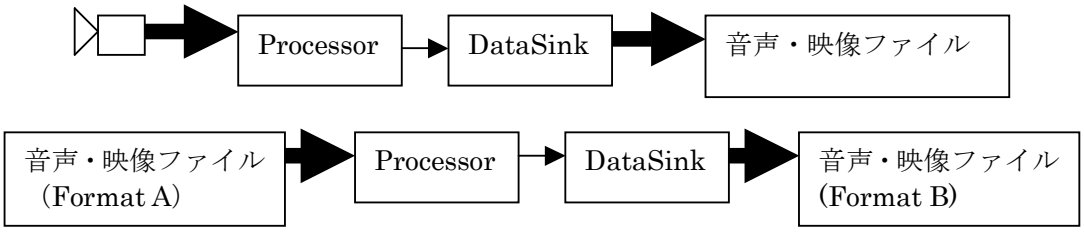


図 4.1　プロセッサを用いた音声・映像ファイルの保存

本章ではファイルへのデータ保存を行う２つのプログラム例を示す。第１のプログラムは図 4.1 の上図に対応するもので、音声・映像キャプチャーデータを音声・映像ファイルに保存する。第２は図 4.1 の下図に対応するもので音声・映像ファイルをフォーマット変換して保存するものである。プログラムの説明に入る前に２つの事項を説明する必要がある。

4.1　プロセッサの生成

プロセッサもプレーヤ同様 Manager の元で Manager.createRealizedProcessor()メソッドにより生成される。しかし、プレーヤに比べて機能が複雑なため、生成法も複雑になり、生成法も色々存在する。代表的な方法としては（１）プロセッサモデルと呼ばれる型紙を準備するタイプの簡単なものと、（２）トラックごとに機能を定義していく方法がある。ここではプロセッサモデルを使用する方法で話しを進める。（２）の方法は VII 章で説明する。

プロセッサモデルを用いる方法は、プロセッサモデルという基本形を用意しておいてプロセッサを作るもので、簡易型生成法である。

先ずは、プロセッサモデルを用いたプロセッサの生成例を示す。

```
processor = Manager.createRealizedProcessor(new ProcessorModel(formats,
        outputType));
```

プロセッサ生成命令：　Manager.createRealizedProcessor(s);
s：プロセッサモデル　ProcessorModel(formats,outputType);
format：プロセッサ内のトラックのデータ形式
outputType：プロセッサの出力タイプ

これを見ると、プロセッサモデル（ProcessorModel）が createRealizedProcessor の引数になっていることがわかる。プロセッサモデルとは典型的なプロセッサの基本構成を示すもので、大方のプロセッサの原型であるため、特別な機能指定をしない普通のプロセッサの生成は、これで済む。プロセッサモデルの基本は図 4.2 のようなもので、各トラックのフォーマット指定と、出力タイプの指定である。プロセッサの出力タイプは CONTENT_DESCRIPTOR と呼ばれるもので、ファイルへの出力の場合 FileTypeDescriptor（例えば FileTypeDescriptor.QUICKTIME）が用いられる。

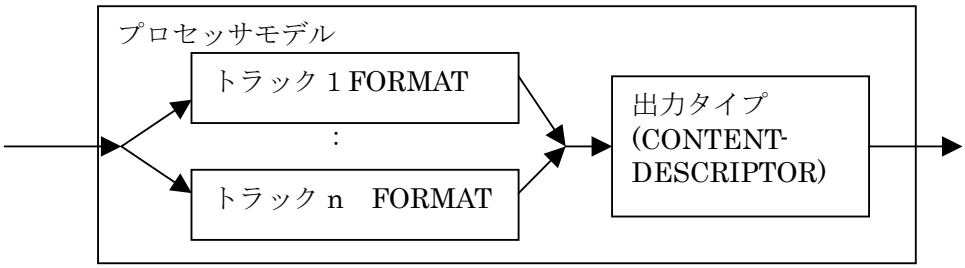


図 4.2 プロセッサモデル

CONTENT__DESCRIPTOR の種類
RAW:Buffer のフォーマットのまま、どんなフォーマットでもよい。
RAW_RTP:RAW の原理に従い RTP 通信に用いる。
この他、主にメディアフォーマットを指定する MPEG, MPEG_AUDIO, QUICKTIME, MIXED, CONTENT_UNKNOWN 等がある。
又、ファイルへの保存の場合 FileTypeDescriptor で指定してもよい(プログラム 7 参照)。

4.2 データシンク（Data Sink）
プロセッサの出力にメディアロケータを指定できれば、出力を直接ファイルやネットワークに送り出すことができるが、実際はそうなっていない。プロセッサの出力をファイルやネットワークに送り出すには図 4.1 のように、プロセッサとファイルの間にデータシンクと呼ばれるものを入れなくてはならない。データシンクも Manager の管理下にあり Manager.create.DataSink()メソッドで生成される。イメージ的に言えばプロセッサの出力は低電圧信号（図 4.1 の細矢印）でファイルにデータを書き込むには高電圧（太矢印）が必要となる。データシンクはこの電圧変換を行う装置と考えるとよい。データシンクへの入力は 3.4.1 節で紹介した DataSource という型が使われる。

データシンクの生成：Manager.create.DataSink(s,d);
s：データシンクへの入力元（DataSource）
d：データシンクの出力先

4.3　音声・映像を圧縮してファイルに保存

ここでは図 4.3 のように音声(IMA4 形式)と映像(CINEPAC 形式)をキャプチャーして QUICKTIME 形式のファイルに保存するプログラムを作成する。

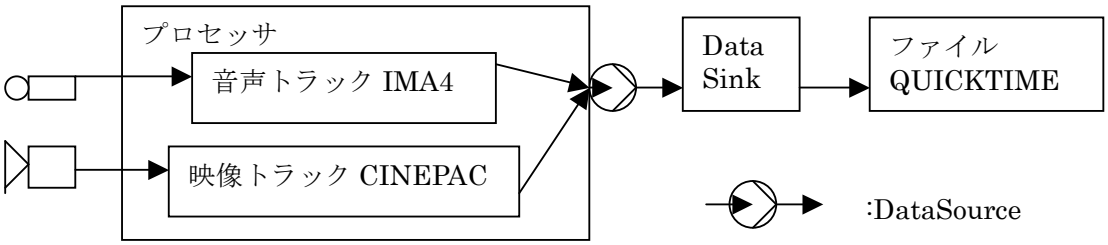


図 4.3　QUICKTIME ファイルの生成

プログラムの構造は図 4.4 である。先ず、2 引数のプロセッサモデル（キャプチャー用モデル）を準備する。このプロセッサモデルでプロセッサのトラックのフォーマットを指定する。次にデータシンクを用意してプロセッサの出力をデータソースとして取りだし、シンクに送る。プロセッサモデルを用いてプロセッサを生成する方法も何種類かあるが、ここで示したプロセッサモデルはキャプチャーデバイスの指定が不要なため大変使いやすい。

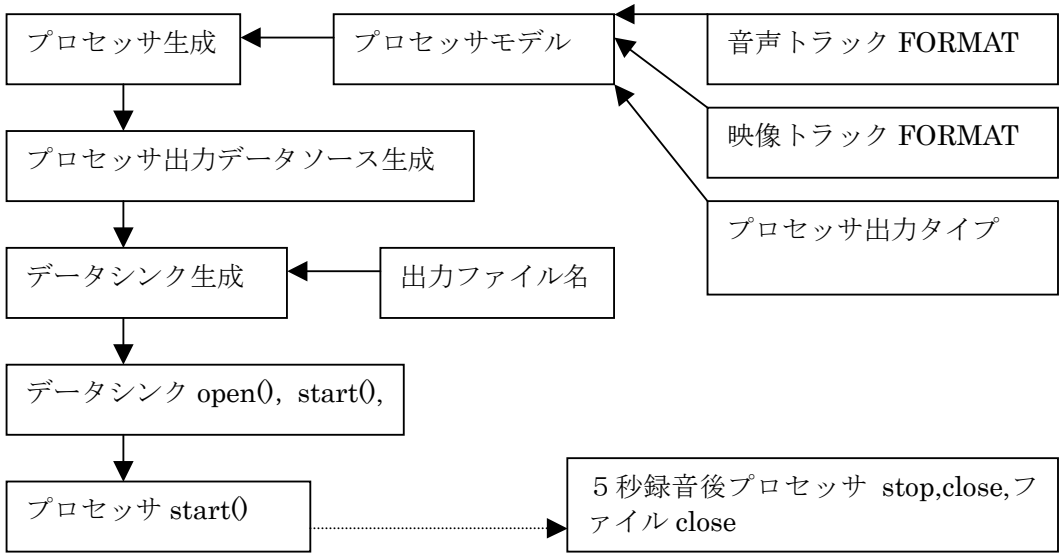


図 4.4　ファイル生成の流れ

プログラム６　ファイル保存

```
import java.io.File;
import java.net.*;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.format.*;

public class Mix {
   public static Processor p= null;
   public static DataSource source= null;
   public static DataSink filewriter= null;

   public Mix() {};

   public void startmix() {
      try {
      Format formats[] = new Format[2];
      formats[0] = new AudioFormat(AudioFormat.IMA4);---①
      formats[1] = new VideoFormat(VideoFormat.CINEPAK);---②
      FileTypeDescriptor outputType =
         new FileTypeDescriptor(FileTypeDescriptor.QUICKTIME);---③
      p = Manager.createRealizedProcessor(new ProcessorModel(formats,
         outputType));---④
      source = p.getDataOutput();---⑤
      MediaLocator dest = new MediaLocator("file://j2sdk1.4.0/jmf/foo.mov");---⑥
      filewriter = Manager.createDataSink(source, dest);---⑦
      filewriter.open();---⑧
      filewriter.start();
      p.start();
      } catch (Throwable th) {}
   }
  public static void main( String[] args ) {
      try {
      Mix mix= new Mix();
      mix.startmix();
      Thread.currentThread().sleep(5000);---⑨
      p.stop();
      p.close();
      filewriter.close();
      System.exit(0);
      } catch (Throwable th) {}
   }
}
```

＜プログラム解説＞①②はプロセッサ内のトラックの音声・映像のフォーマットを指定している。③はプロセッサの出力タイプを決定している。④はプロセッサモデルを利用してプロセッサを生成する。プロセッサの出力をファイルに送る場合、データシンクが必要となることは 4.2 節に示した。データシンクを使うには少し決まりごとがある。プロセッサとデータシンクの接続は、プロセッサ出力は DataSource 型と決まっていて、getDataOutput()で取り出される⑤。データシンクとプロセッサの結合は⑦の createDataSink()の引数として、入力(⑤)と出力(⑥)を指定することによって行われる。⑧以降でデータシンクとプロセッサをスタートさせる。⑨は５秒間録画する仕掛けである。startmix で動き出したプロセッサは動き続けており、５秒後にストップさせている。このプログラムを実行すると、foo.mov というファイルができる。

4.4　ファイルの形式を変換してファイルに保存

本節ではファイルの形式変換、具体的には～.wav 形式で保存されているファイルを～.mp3 の圧縮ファイルに変換するプログラムを示す。このプログラムは前節と似ているが、次の３つの注目点がある。

＊　３つの引数を持ったプロセッサモデルの利用法：前節ではキャプチャーデバイスから直接プロセッサに信号を送る方式だったため、プロセッサへの入力情報を省くことができ、引数２つのプロセッサモデルを用いた。正式には次に示す通り、３種類のプロセッサモデルが使用可能である。本節では３番目の３引数のモデルを利用する。

1. ProcessorModel()：引数なしのモデル
2. ProcessorModel(f,o)：4.1 節参照
3. ProcessorModel(s,f,o)：

s:プロセッサへの入力（注）、f:トラックのフォーマット、o:出力タイプ

（注）プレーヤの場合、引数は URL を指定できたが、プロセッサの引数はメディアロケータかデータソースしか許されていない。

＊イベントの利用したメディア終了処理：ファイル変換終了イベントをプロセッサに付加したリッスナー（ControllerListener）で検出し、検出した際の処理は ControllerUpdate で行われる。このため、インタフェース implements ControllerListener を加える必要がある。

次のプログラムは、ファイル aa.wav を aa.mp3 にタイプ変換を行う。

プログラム７　ファイル変換

```
import java.io.File;
import java.net.*;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.format.*;

public class Conv implements ControllerListener {
   public Processor processor;
```

```
   public DataSource source;
   public DataSink sink;

   public Conv() {
      try {
      File sfile= new File("aa.wav");---①
      MediaLocator slocator= new MediaLocator(sfile.toURL());
      Format formats[] = new Format[1];
      formats[0] = new AudioFormat(AudioFormat.MPEG);
//      FileTypeDescriptor outputType =
//         new FileTypeDescriptor(FileTypeDescriptor.MPEG_AUDIO);---c1
      ContentDescriptor outputType= new ContentDescriptor("audio.mpeg");--c2
//ContentDescriptor outputType=
//          new ContentDescriptor(FileTypeDescriptor.MPEG_AUDIO);---c3
      processor= Manager.createRealizedProcessor(
         new ProcessorModel(slocator, formats, outputType));
      processor.addControllerListener(this);---②
      source= processor.getDataOutput();
      File dfile= new File("aa.mp3");---③
      MediaLocator dlocator= new MediaLocator( dfile.toURL());
      sink= Manager.createDataSink( source, dlocator );
      sink.open();
      sink.start();
      processor.start();
      } catch(Throwable t) {System.err.println("er2");}
   }

   public void controllerUpdate( ControllerEvent e ) {---④
      if ( e instanceof EndOfMediaEvent ) {  //StopEvent ) {
         processor.stop();
         processor.close();
         sink.close();
         System.exit(0);
      }
   }

   public static void main( String[] args ) {
      try {
         Conv conv= new Conv();
      } catch (Throwable th) {};
   }
}
```

＜プログラムの解説＞①は変換前のファイルで、③が変換後のファイルである。②がファイルの終了イベントを検出するリッスナー、④が終了イベント発生時の処理である。

プログラム中の c1, c2, c3 は CONTENT_DESCRIPTOR の異なる 3 種類の与え方を示している。これらは、どれを使っても問題ない。
注意：
（１）processorModel の Format 指定と ContentDescriptor の指定が異なると（例えば Format が Audio.mpeg で ContentDescroptor が MPEG(映像)）エラーになる。
（２）変換後のファイル形式は ProcessorModel で決まり、出力ファイル名（上では dfile）は変換に影響を与えない。例えば ProcessorModel で mpeg を指定し dfile を wav とした場合、~.mp3 ファイルが~.wav という名前で生成される。そのため、音の再生時にプレーヤがエラーを出す。

V　RTP 通信とマルチキャスト

映像配信、ビデオオンデマンド、遠隔ビデオ会議などインターネットを介した実時間ストリーム転送の必要性が高まっている。本章ではこれらの要求に答えて JMF に採り入れられた、ストリーミングとマルチキャスト、そしてそれらをサポートする RTP(Real-Time Transport Protocol)という通信プロトコルを紹介する。

5.1　マルチキャスト

インターネットの通信プロトコル UDP（次節参照）は①ユニキャスト、②マルチキャスト、③ブロードキャストという３つの通信形態をサポートしている。ユニキャストは特定の IP アドレスにデータを転送し、ブロードキャストはネットワークにつながっている全てのコンピュータにデータを送る。これに対しマルチキャストとは特定のマルチキャスト用 IP アドレス（224.0.0.1～239.225.225.225）を用意しておき、そこに送信者がデータパケットを送ると、パケットが複製されて、そのマルチキャスト IP アドレスを共有する全受信者に送られる。マルチキャストパケットの配信形態はネットワーク形態によって異なる。図 5.1　はその基本形で、１送信者が多数の受信者にパケットを送っている。この時、受信者が送信者となり同じマルチキャスト IP アドレスにパケットを送り返すこともできる。この場合もパケットは当然全員に配送される。

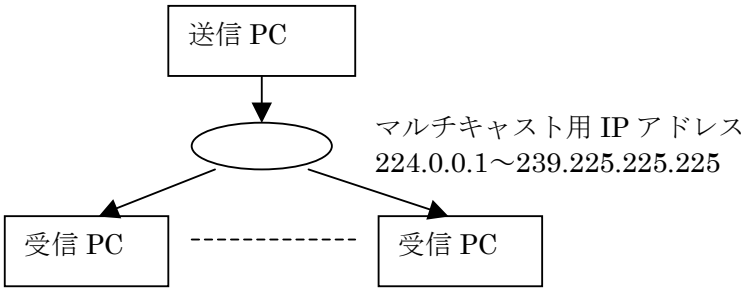


図 5.1　マルチキャストの基本型

5.2　RTP 通信

映像、音楽などを、インターネットを介して配信する場合、受信側でいったんデータを全て保存し、全て保存後に再生する方法と、受信しながら再生する方法が考えられる。前者はダウンロード方式と呼ばれ、後者はストリーミング方式と呼ばれる。ストリーミングは受信側で保存用メモリが必要なく、ビデオ会議やインターネット放送などリアルタイム映像送信に使用される。メディアデータをリアルタイムで送る場合、高速転送が不可欠となる。その際、多少情報の欠落があっても遅延無くデータを配信することが重要になる。インターネットの世界ではデータ欠落が起こらない通信プロトコルとして TCP が知られ、HTTP や FTP に用いられている。これに対し高速性を重視しデータ欠落をチェックしないプロトコルに UDP がある。TCP と UDP の上位に位置し、音声や映像のような実時間データ転送のためのインターネット標準プロトコル

として RTP がある（図 5.2)。一般に RTP は UDP の上に作られている。RTP はユニキャスト、マルチキャストいずれのネットワークサービス上でも使用できる。ユニキャストの場合は、送信元からデータのコピーが各送信先に送られる。マルチキャストの場合、送信元からは 1 度だけデータが送り出され、ネットワークが各送信先へのデータの複製と配信を行う。このため、ビデオ会議などの応用ではマルチキャストの方が効率がよい。

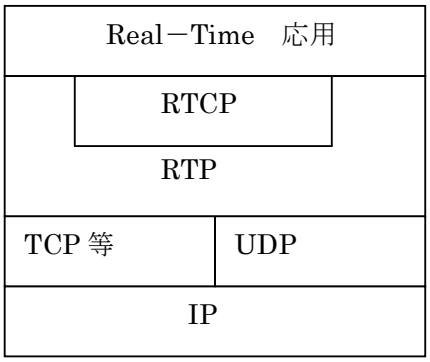


図 5.2　インターネットプロトコル

RTP ではデータが失われることはあるが、受信側でヘッダの情報を元に送信順に並べ替えたりすることはできる。しかし、利用者に不満の出ないよう転送を制御しサービスの品質を管理することはできない。これを行う制御プロトコルが RTCP である。図 5.3　は RTP 方式の動作を示している。通信にはデータパケットとコントロールバケットが組みになって動作し、それぞれにポートが割り当てられる。送信側は送信データの入っている RTP パケットを送ると共に、時々通信制御用 RTPC パケットを送る。受信側は RTP パケット（データ）を受け取ると共に時々制御用 RTPC パケットを返す。このように時々しか通信制御を行わないので負荷が軽くなる。

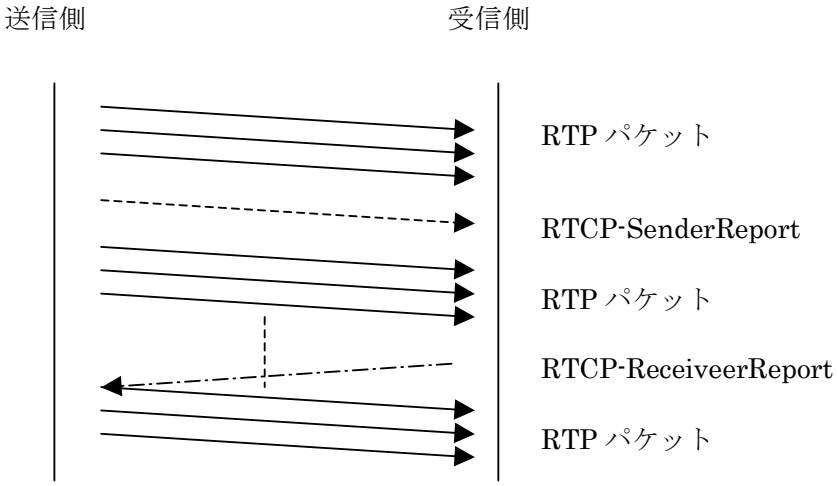


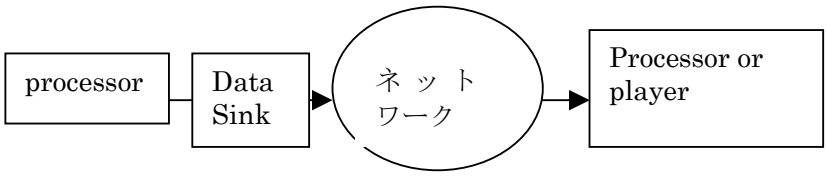
図 5.3　RTP 通信方式

RTP で通信している状態はセッションと呼ばれ、１つのセッションは１つのネットワークアドレスと２つのポートで識別される。２つのポートの内、１つは RTP 用、もう一つは RTCP 用である。セッションに参加しているマシンやユーザはパーティシパント（参加者）と呼ばれ、データを送信したり受信したりできる。複数のメディアデータが送られる場合、各メディアは別のセッションで転送される。例えば、ビデオ会議で音声と映像が流される時、音声と映像は別のセッションで流される。これにより参加者は自分のシステム性能に合ったメディアを選択できる。例えばバンド幅の狭いユーザは音声だけで会議に参加できる。RTP パケットはヘッダとペイロード（転送するデータ）からできている。RTCP パケットでは転送品質に関する情報が送られる。例えば、送信側からは送信パケット数、受信側からは紛失パケット数やタイムスタンプなどが送られる。また、RTCP パケットには CNAME という送信元を特定する情報も含まれる。

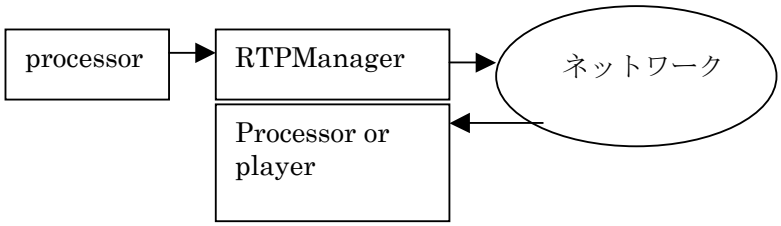
ネットワーク転送ではパケットの寿命を示す TTL（Time to Live）を指定する。JMF のマルチキャストでは“１”（１ホップ：ルータを１回通す）がデフォールトになっている。

VI　通信プログラム

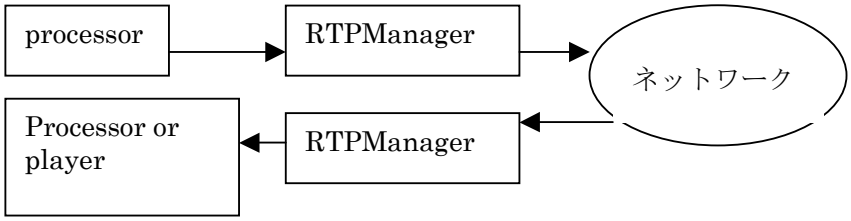
JMF は、ネットワークを介して通信するための道具立てとしてデータシンク（ファイル保存の時使用されたものと同じ）と RTP マネージャという機能を用意している。データシンクと RTP マネージャを用いた基本的通信形態は次のようなものが考えられる。



(a) データシンクでネットワークに送信し、メディアロケータで受信する



(b) RTP マネージャで送信し、メディアロケータで受信



(c)　RTP マネージャで送信し、RTP マネージャで受信する

図 6.1 ネットワーク通信形態

この他にデータシンクで送信しRTPマネージャで受信する形態も考えられる。
以下の節ではこれらのタイプについて音声の送受信を例に取ってプログラムを作成する。

6.1　データシンクで送信、メディアロケータで受信（図6.1(a)）
最も簡単で基本的な通信形態は、送信側と受信側が互いに相手のIPアドレスとポート番号を知っているというものである。先に説明した通り、送信側には出力を持つプロセッサが使われ、受信側は単に再生だけを行えばよいのでプレーヤが使われる。ここでは先ず送信側から見ていく。
送信側の構成は図6.2のようにマイクからの音声入力をメディアロケータ型に変換しプロセッサへ入力し、出力をデータシンクでネットワークに送り出す。プレーヤへはURL型を入力できるが、プロセッサはメディアロケータ型かデータソース型しか入力として受け付けないため、音声入力をメディアロケータ型に変換している。プロセッサを作成するためには先ずプロセッサモデルと呼ばれるものを作成するのはIV章で述べた。プロセッサモデル作成の引数には、入力（メディアロケータ)、送信データフォーマット、出力タイプ（コンテントデスクリプタ）の指定が必要である。このプログラムでは出力をネットワークに送り出すため、コンテントデスクリプタはファイルの時と異なり、ContentDescriptor.RAW_RTP を用いる。以上でできたプロセッサモデルを引数にしてプロセッサを作成する。

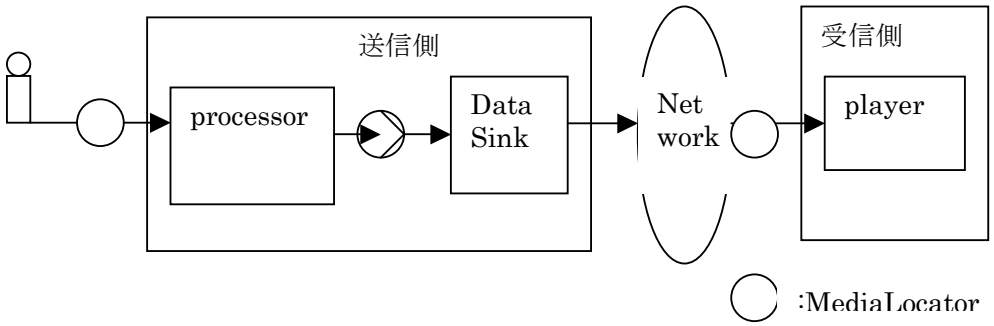


図 6.2　　送信側と受信側の構成

図6.3は、このフローチャートを示している。特に、データシンクとプロセッサをスタートさせる順序は見やすい形にしてある。

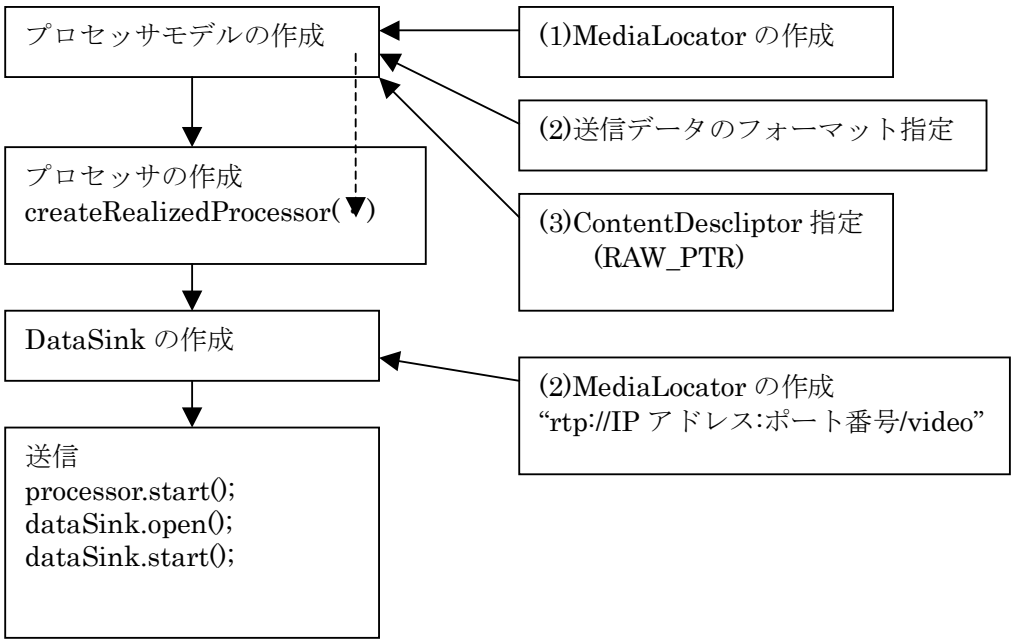


図 6.3　　送信プロセス

次のプログラムは IP アドレス 192.168.1.20 のコンピュータから 192.168.1.20 のコンピュータへポート 4500 を通して音声データを転送するものとしている。入力マイクは予めプログラム４で調べ dsound://が使用可能であることを確認している。

プログラム８－１ ：送信プログラム

```
import java.awt.*;
import java.awt.event.*;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.format.*;
import javax.swing.*;

public class T1 extends JFrame {

    private MediaLocator mediaLocator = null;
    private DataSink dataSink = null;
    private Processor processor = null;
    private Format[] formats = new Format[] {
        new AudioFormat( AudioFormat.LINEAR ) };---①
    private ContentDescriptor descriptor =
        new ContentDescriptor( ContentDescriptor.RAW_RTP );---②
```

```
    public T1() {
        try {
            MediaLocator locator = new MediaLocator("dsound://");---③
            processor = Manager.createRealizedProcessor(
                new ProcessorModel( locator, formats, descriptor ) );
            mediaLocator
             = new MediaLocator( "rtp://192.168.1.21:4500/audio" );---④
            dataSink = Manager.createDataSink(
                   processor.getDataOutput(), mediaLocator );---⑤

           processor.start();
           dataSink.open();
           dataSink.start();
        } catch (Throwable t) {}
    }

    public static void main(String[] args) {
        T1 transmitter = new T1();
    }
}
```

＜プログラムの解説＞
送信プログラム本体は次のステップで実行される。
ステップ 1:マイクからの音声をメディアロケータ型に変換してプロセッサの入力とし③、
ステップ 2:音声フォーマット①と、プロセッサの出力タイプ②を指定してプロセッサを生成。
ステップ 3:プロセッサの出力をデータシンクにつなぎ⑤、データシンクの出力は送信先を示すメディアロケータ④を指定する。
④は RTP 通信であり、転送先 IP アドレスは 192.168.1.21 でポート番号が 4500 で、音声データであることを示している。④の記法は唐突な感じであるが、これは RTP 通信の URL 記法で、以下のように決まっている。

音声の場合：”rtp” + IPAddress + “:” + Port + “/audio”;
映像の場合：”rtp” + IPAddress + “:” + Port + “/video”;

ステップ 4:プロセッサとデータシンクをスタートさせる。

受信プログラムはさらに簡単である。送信側の IP アドレスとポート番号を入力（メディアロケータ）にしてプレーヤをスタートさせるだけである。

プログラム８-２ ：受信プログラム

```
import java.awt.*;
import java.awt.event.*;
```

```
import javax.media.*;
import javax.swing.*;

public class R1 extends JFrame {

    private MediaLocator mediaLocator = null;
    private Player player = null;

    public R1() {
        try {
            mediaLocator
             = new MediaLocator( "rtp://192.168.1.20:4500/audio" );---①
            player = Manager.createRealizedPlayer( mediaLocator );
            player.start();
        } catch (Throwable t) {}
    }

    public static void main(String[] args) {
        R1 receiver = new R1();
    }
}
```

＜プログラムの解説＞
①で、音声が送信される通信タイプ RTP、IP アドレス、ポート番号、音声、という指定を行う。
＜プログラムの使い方＞
コンパイル
送信側： javac T1.java
受信側： javac R1.java
実行
送信側： java T1
受信側： java R1

6.2 RTP マネージャを使用する送受信（図 6.1(c)）
RTP でネットワーク通信を行うもう一つの方法は RTP マネージャを使う方法である。図 6.4 が RTP マネージャを使った場合の詳細構成である。この図から、前節でデータシンクを使っていた所が RTP マネージャになっていることが分かる。RTP マネージャを用いることによりデータシンクより柔軟な通信が可能になる。データシンクではセッションの間中 JMF がオブジェクトの管理と制御を行うのに対し、RTP マネージャではプログラムでセッションを制御できる。例えば、RTP マネージャではストリームや参加者を追跡し、受信はストリームの到着をもって作業を開始したり、ヘッダ情報を操作したり、RTCP イベントに反応して動くプログラムを作成したりできる。また、データシンクではプロセッサから複数のストリームが送り出されている場合、１つ（最初）のストリームしか転送できない、といった制限がある。

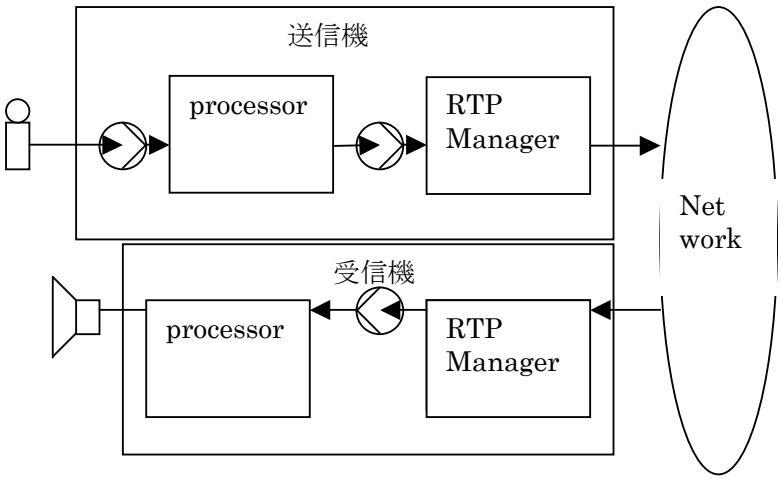


図 6.4　RTP マネージャを使った通信

様々な機能を持つ RTP マネージャは、実装する通信形態により操作手順が異なる。以下に３種類の通信形態とそれに対応した RTP マネージャのし様手順をまとめる。

（1） ユニキャストの場合：１対１の通信
＜通信開始処理＞
java.net.*と javax.media.rtp.*を import する。
RTP マネージャの生成
RTPManager rtpManager= RTPManager.newInstance();
ローカルエンドポイント（送信元）の生成
SessionAdress localAddress= new SessionAddress();
RTP マネージャの initialize。
rtpManager.initilize(localAddress);
（受信の場合、ReceiveStreamListener を付加）
リモートエンドポイント（通信の相手先）を作る
InetAddress ipAddress= InetAddress.getByName(“168.1.2.3”);
SessionAddress remoteAddress= new SessionAddress(ipAddress,3000);
結合をオープン

```
rtpManager.addTarget(remoteAddress);
プロセッサの出力に対して送信ストリームを作り、それをスタート
DataSource dataOutput= createDataSource();
SendStream sendStream= rtpSession.createSendStream(dataOutput,1);
SendStream.start();
＜通信終了処理＞
rtpManager.removeTarget(remoteAddress,” disconnected”);
rtpManager.dispose();
```

（２） マルチ-ユニキャストセッション：(アドレスで特定される) 複数のコンピュータに送信
ユニキャストと似ている。SendStream を生成しスタートした後、リモートエンドポイントを付け加える。例えば、

```
rtpManager.addTarget(remoteAddress2);
rtpManager.addTarget(remoteAddress3);
```

（３） マルチキャストセッション：マルチキャストアドレスに送信
ユニキャストに似ている。ローカル及びリモートエンドポイントを同一のマルチキャストアドレスにする。例えば、

```
InetAddress ipAddress= InetAddress.getByName(“224.1.1.0”);
SessionAddress multiAddress= new SessionAddress(ipAddress,3000);
rtpManager.initilize(multiAddress);
rtpManager.addTarget(multiAddress);
```

次に RTP マネージャを使い、マルチキャストで通信するプログラムを紹介する。送信側はマルチキャスト用アドレス ″224.0.0.1″ (ポート 5004) に音声ストリームを送信し、受信側は音声ストリームが到着次第プレーヤを生成して再生を開始する。送信側と受信側では、RTP マネージャの扱いが少し異なる (色々なプログラム法を紹介するため、意図的に変えてみた)。送信側は送信ストリームが未知の場合を想定しており、受信側は受信ストリームが１本の場合を想定している。送信側では、プロセッサの出力を PushBufferDataSource という型で受け、ストリーム数を検出し、各ストリームに対して RTP マネージャを生成している。
図 6.5 は図 6.4 を実現するプログラムの流れを示している。プログラム９は図 6.5 を書き下したものである。

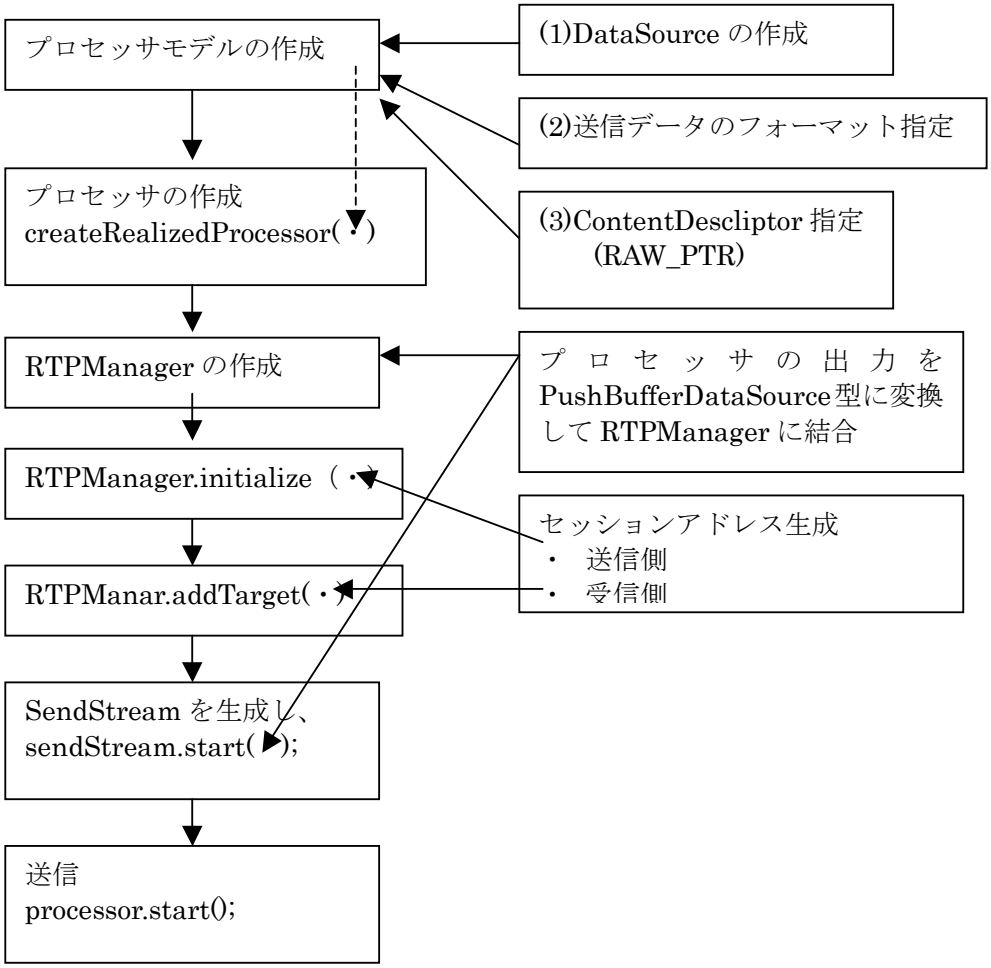


図 6.5　送信プログラム

プログラム９－１　RTP マネージャを使った送信プログラム

```
import java.awt.*;
import java.io.*;
import java.net.URL;
import java.net.InetAddress;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.format.*;
import javax.media.rtp.*;
import javax.media.rtp.rtcp.*;

public class T2 {
```

```
MediaLocator locator;
String ipAddress;
int portBase;
Processor processor= null;
RTPManager rtpMgrs[];---①
DataSource dataOutput= null;
URL url;
Format[] formats= new Format[] {new AudioFormat(AudioFormat.LINEAR)};
SessionAddress localAddr, destAddr;
SendStream sendStream;

public T2() {
   InetAddress ipAddr;
   int port;

   try  {
      locator= new MediaLocator("dsound://");
      ipAddress= "224.0.0.1";
      portBase= 5004;
      processor= Manager.createRealizedProcessor(
         new ProcessorModel(locator, formats,
            new ContentDescriptor(ContentDescriptor.RAW_RTP)));
      dataOutput= processor.getDataOutput();
      PushBufferDataSource pbds= (PushBufferDataSource)dataOutput;---②
      PushBufferStream pbss[]= pbds.getStreams();
      rtpMgrs= new RTPManager[pbss.length];

      for( int i=0; i<pbss.length; i++)  {
         rtpMgrs[i]= RTPManager.newInstance();
         port= portBase + 2*i;
         ipAddr= InetAddress.getByName(ipAddress);
         localAddr= new SessionAddress( InetAddress.getLocalHost(), port);---③
         destAddr= new SessionAddress( ipAddr, port);
         rtpMgrs[i].initialize( localAddr);
         rtpMgrs[i].addTarget( destAddr);
         sendStream= rtpMgrs[i].createSendStream( dataOutput, i);
         sendStream.start();
         processor.start();
         System.out.println("No"+i);

      }
   } catch (Throwable t)  {}
}
```

```
    public static void main(String[] args)  {
       T2 at= new T2();
    }
}
```

＜プログラムの解説＞①は複数のストリームに備え RTP マネージャの配列を用意。②の PushBufferDataSource は DetaSource の一種で、プロセッサの出力は、この形になっている。

次に、RTP マネージャを用いて受信するプログラムを紹介する。受信側では、ストリームの到着を検出し、ストリームごとにプレーヤを生成する。新しいストリームを検出するために RTP マネージャに ReceiveStreamListener を付加する。また、新しいストリームの到着時に動作する関数 updata()を用意する。プログラムの流れは図 6.6 であり、プログラムはプログラム９－２である。このプログラムは簡単化のため、送信プログラムと異なり、ストリームは１本と仮定している。

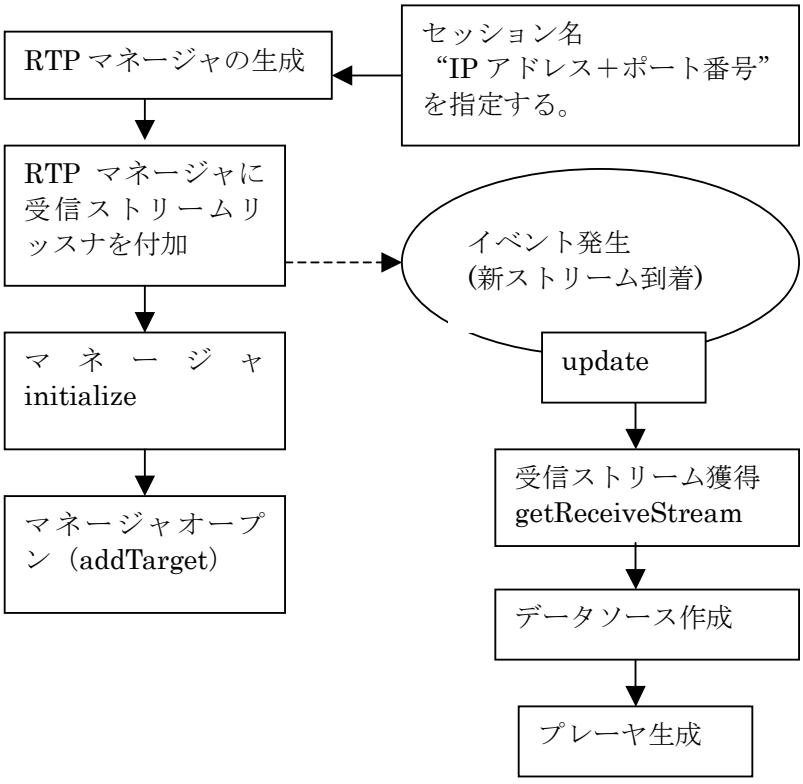


図 6.6　受信プログラム

プログラム９－２　RTP マネージャを使った受信プログラム

```
import java.io.*;
```

```
import java.awt.*;
import java.net.*;
import java.awt.event.*;
import javax.swing.*;
import javax.media.*;
import javax.media.rtp.*;
import javax.media.rtp.event.*;
import javax.media.rtp.rtcp.*;
import javax.media.protocol.*;

public class R2 extends JFrame implements ReceiveStreamListener
{
    Player player= null;

    RTPManager mgrs = null;

    public R2() {
        String sessionip= "224.0.0.1";
        int sessionport= 5004;
        int sessionttl=1;---①

        try {
            InetAddress ipAddr;

            SessionAddress localAddr;
            SessionAddress destAddr;

            System.err.println("  - Open RTP session for: addr: " + sessionip + " port:
" + sessionport + " ttl: " + sessionttl);


            mgrs = (RTPManager) RTPManager.newInstance();
            mgrs.addReceiveStreamListener(this);---②

            ipAddr = InetAddress.getByName(sessionip);

   //マルチキャストでは local and remote address は同一
                localAddr= new SessionAddress( ipAddr,
                                               sessionport,
                                               sessionttl);
                destAddr = new SessionAddress( ipAddr,
                                               sessionport,
                                               sessionttl);
```

```
            mgrs.initialize(localAddr);
            mgrs.addTarget(destAddr);

        // 1 分間待つ
            System.err.println("  - Waiting for RTP data to arrive (1 min.)");
Thread.sleep(60000);

        } catch (Throwable t) {}
    }

    public synchronized void update(ReceiveStreamEvent evt){---③
        ReceiveStream stream = evt.getReceiveStream();

        if (evt instanceof NewReceiveStreamEvent) {---④
            System.out.println("newstream");
            try {
                stream = ((NewReceiveStreamEvent)evt).getReceiveStream();---⑤
                DataSource ds = stream.getDataSource();---⑥
                player = Manager.createRealizedPlayer(ds);---⑦
                player.start();
                show();

            } catch (Exception e) {
                System.err.println(" connection error of new stream");
                return;
            }
        }
    }

    public static void main(String argv[]) {
        R2 receiver = new R2();
    }
}
```

＜プログラムの解説＞①の ttl はパケットの寿命でデフォールトは１コンパイル
②はストリーム到着検出用リッスナーの付加。③でストリーム到着をチェックし④で新しいストリームかを調べている。新しいストリームの場合⑤でストリームを獲得し、⑥でストリームの DataSource 型を得て、⑦で到着ストリームに対するプレーヤを生成している。

＜プログラムの使い方＞
送信側　javac T2.java
受信側　javac R2.java
実行
送信側　java T2

受信側　java R2
注意：受信開始後１分以内に送信を開始すること。

以上、２つの通信形態のプログラムを紹介したが、この他にも色々な形態が考えられる。
（１）データシンクで送りだし RTP マネージャで受信するもの
（２） RTP マネージャで送信し LRU で受信するもの、
などであるが、大抵のプログラムは上の送受信プログラムを組み合わせることにより作成できる。

6.3　ビデオ会議もどき
ビデオ会議の原型を紹介する。ビデオ会議と言っても機能は最低限で、音声もないし、圧縮も行っていない。また、自分が送信するマルチキャストアドレスとポート番号も予めプログラム中に書き込んでおかなくてはならない。ただ、通信方式は RTP マネージャで送信し、RTP マネージャで受信するという最高機能を実現している。先ず、本ビデオ会議の基本構成を参加者３名の場合について紹介する。図 6.7 は３人の参加者が、それぞれ異なるマルチキャストアドレスにカメラからの映像を送り出している様子を示している。各参加者の大きな四角は、各参加者のモニタ画面を示している。図 6.7 のモニタ画面には 2 名の参加者の映像と、接続ウインドウ（ip-Address,port,connect）が表示されているが、最初は接続ウインドウしか表示されていない。

ウインドウの接続ウインドウには接続先 IP アドレスとポート番号を書き込む領域と”connect”ボタンがある。接続したい IP アドレスとポート番号を入力し”connect”ボタンを押すと図の点線部が接続され接続先 PC のカメラ画像が表示される。例えば、参加者１が接続ウインドウに”224.0.0.2”” 2000”を入力し、”connect”ボタンを押すと“参加者２の映像”ウィンドウが開き、参加者２のカメラ画像が参加者１の PC の画面に表示される。さらに、参加者３の映像を見たい場合には、接続ウインドウに参加者３の IP アドレスとポート番号を入力し”connect”ボタンを押す。ウインドウは全て画面の左上に表示されるので、ドラッグで好きな場所に移動する。図 6.7 ではマルチキャストアドレスを示しているが、これらは１例であり、自由に決められる。

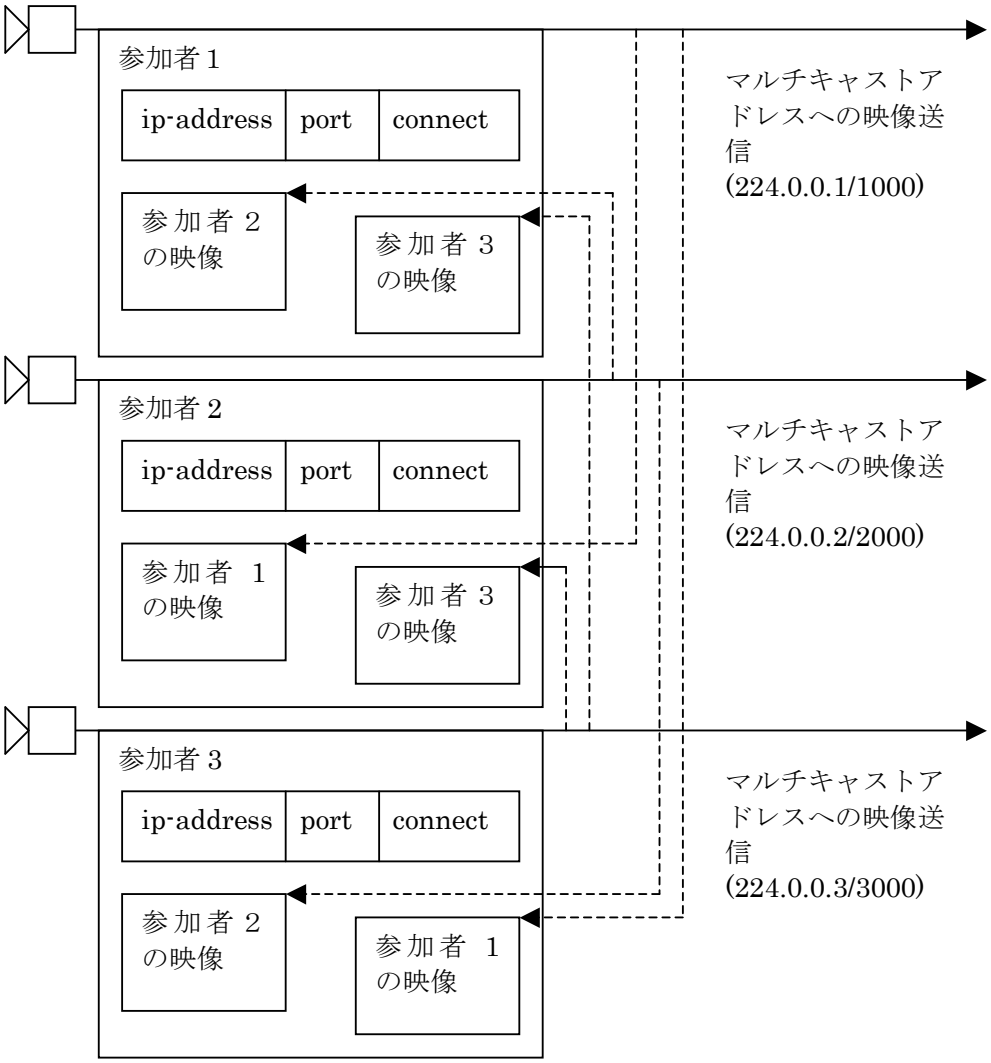


図 6.7　ビデオ会議もどき

参加者 1 の JMF モデルは図 6.8 である。プログラムを走らせると、図の上部（送信部）

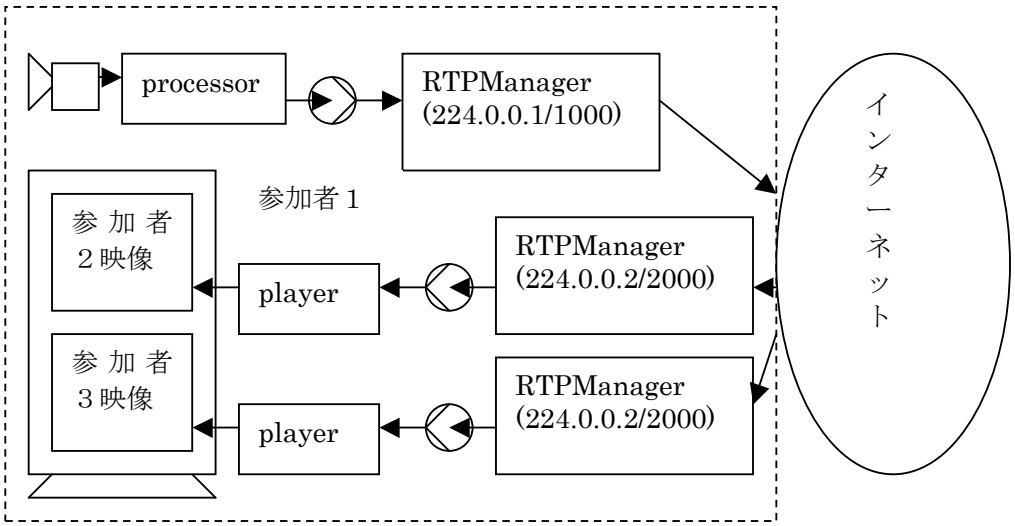


図 6.8　参加者１の内部構成

が作られ送信を開始する。その後接続ウインドウに”connect”ボタンを押すと受信用 RTP マネージャが生成される。RTP マネージャはインターネットを監視し、他の参加者が映像を送り出すのを待つ。RTP マネージャは他参加者の映像を確認すると、プレーヤを生成し映像再生を開始する。

プログラムは、ユーザインタフェースがある以外プログラム９に近い。

プログラム１０：ビデオ会議もどき

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.InetAddress;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.format.*;
import javax.media.rtp.*;
import javax.media.rtp.rtcp.*;
import javax.swing.*;

public class Conf extends Frame implements ActionListener {
   Button bt;
   TextField tf1,tf2;
   ReceivePlayer rp;
   MediaLocator locator;
```

```
String ipAddress;
int portBase;
Processor processor= null;
RTPManager rtpMgrs[];
DataSource dataOutput= null;
InetAddress ipAddr;
int port;
Format[] formats= null;
SessionAddress localAddr, destAddr;
SendStream sendStream;

public Conf() {
   try  {
      locator= new MediaLocator("vfw://0");
      ipAddress= "224.0.0.1";---①
      portBase= 1000;---②
      processor= Manager.createRealizedProcessor(
         new ProcessorModel(locator, formats,
            new ContentDescriptor(ContentDescriptor.RAW_RTP)));
      dataOutput= processor.getDataOutput();
      PushBufferDataSource pbds= (PushBufferDataSource)dataOutput;
      PushBufferStream pbss[]= pbds.getStreams();
      rtpMgrs= new RTPManager[pbss.length];
      SourceDescription srcDesList[];

      for( int i=0; i<pbss.length; i++)  {
         rtpMgrs[i]= RTPManager.newInstance();
         port= portBase + 2*i;
         ipAddr= InetAddress.getByName(ipAddress);
         localAddr= new SessionAddress( InetAddress.getLocalHost(), port);
         destAddr= new SessionAddress( ipAddr, port);
         rtpMgrs[i].initialize( localAddr);
         rtpMgrs[i].addTarget( destAddr);
         sendStream= rtpMgrs[i].createSendStream( dataOutput, i);
         sendStream.start();
         processor.start();
      }
   } catch (Throwable t)  {}
}

public void connect(){
   setLayout(new FlowLayout());
   tf1= new TextField("IP-address",10);
   add(tf1);
```

```
        tf2= new TextField("port",5);
        add(tf2);
        bt=new Button("connect");
        add(bt);
        bt.addActionListener(this);
        addWindowListener(new WindowAdapter(){   ---③
          public void windowClosing(WindowEvent e){
            processor.stop(); processor.close(); System.exit(0);}});
    }

    public void actionPerformed(ActionEvent e) {   ---④
        String addr;
        int port;
        if(e.getSource()==bt) {
          addr = tf1.getText();
          port = Integer.parseInt(tf2.getText());
          rp =  new ReceivePlayer();
          rp.initialize(addr, port);
        }
    }

    public static void main(String args[]){
        Conf conf=new Conf();
        conf.connect();
        conf.setSize(250,70);
        conf.setLocation(500,0);
        conf.show();
    }
}

// ReceiverPlayer.java---⑤
import java.io.*;
import java.awt.*;
import java.net.*;
import javax.swing.*;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.rtp.*;
import javax.media.rtp.event.*;

public   class   ReceivePlayer   extends   JFrame   implements      SessionListener,
ReceiveStreamListener {
    Player player = null;
    String sessions[] = null;
```

```
    RTPManager mgrs = null;
    String url;
    String host;
    Component canvas;
    Component panel;

    boolean initialize(String session_ip , int port){
        int ttl=1;
        try {
            InetAddress ipAddr;
            SessionAddress localAddr = new SessionAddress();
            SessionAddress destAddr;
            System.err.println("  - Open RTP session for: addr: " + session_ip +
" port: " + port + " ttl: " + ttl);
            mgrs = (RTPManager) RTPManager.newInstance();
            mgrs.addSessionListener(this);
            mgrs.addReceiveStreamListener(this);
            ipAddr = InetAddress.getByName(session_ip);
            localAddr= new SessionAddress( ipAddr, port, ttl);
destAddr = new SessionAddress( ipAddr, port, ttl);
            mgrs.initialize( localAddr);
            mgrs.addTarget(destAddr);
            } catch (Exception e) { return false;}
        return true;
    }

    public synchronized void update(SessionEvent evt){
        System.out.println(evt.toString() );
    }

    public synchronized void update(ReceiveStreamEvent evt){
        ReceiveStream stream = evt.getReceiveStream();

        if (evt instanceof NewReceiveStreamEvent) {
            System.out.println("newstream");
            try {
                stream = ((NewReceiveStreamEvent)evt).getReceiveStream();
                DataSource ds = stream.getDataSource();
                Player p = Manager.createRealizedPlayer(ds);
                canvas= p.getVisualComponent();
                getContentPane().add(canvas);
                pack();
                setVisible(true);
                p.start();
```

```
            } catch (Exception e) { return; }
        }
        else if (evt instanceof StreamMappedEvent) {}
        else if (evt instanceof ByeEvent) {}
    }
}
```

＜プログラムの解説＞
各参加者は自分が送り出すマルチキャストアドレスとポート番号を決めておいて①②、プログラム中に書き込む。④は connect ボタンが押された時の処理で、⑤は受信した他の参加者の映像を表示するプレーヤである。③はウインドウが閉じられた時の終了処理である。

＜プログラムの使い方＞
コンパイル
参加者１:Conf.java の送信先アドレスとポートを(例えば)マルチキャストアドレス 224.0.0.1 ポート 1000 とする。Conf.java と ReceiverPlayer.java を用意し、
javac Conf.java
参加者２：Conf.java の送信先アドレスとポートを
（例えば）マルチキャストアドレス 224.0.0.2　ポート 2000 とする。Conf.java と ReceiverPlayer.java を用意し、
javac Conf.java

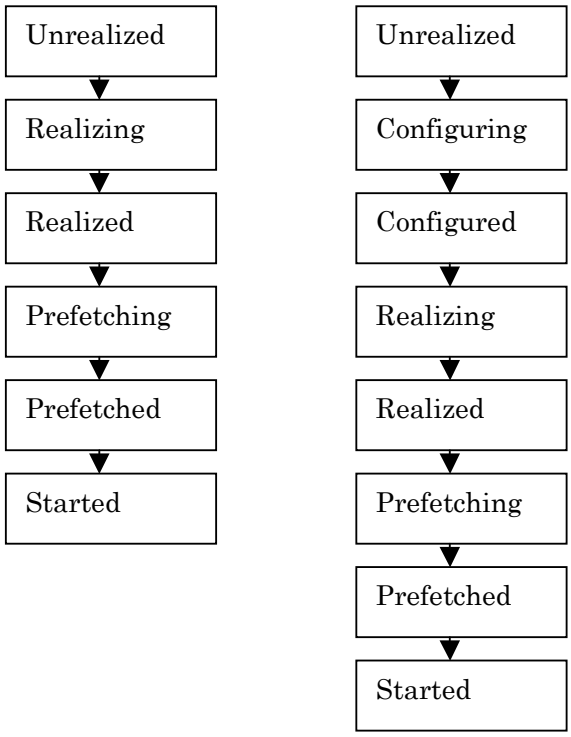
実行
参加者１：java Conf
と入力。モニタ画面に接続ウインドウが現れるので、相手のマルチキャストアドレスとポート番号（上の場合 224.0.0.2 と 2000）を入力し connect ボタンを押す。
参加者２：java Conf
と入力。モニタ画面に接続ウインドウが現れるので、相手のマルチキャストアドレスとポート番号（上の場合 224.0.0.1 と 1000）を入力し connect ボタンを押す。
注意：映像が表示されなかったり、映像が変化しないコンピュータがあった（コンピュータの精？プログラムの問題？)。その時は映像ウインドウの位置を動かすと正しく動作した。

VII　プロセッサの生成と映像圧縮

本章では先ず、プレーヤやプロセッサを細かく機能指定しながら生成していく方法を紹介し、次にそれを用いた圧縮映像の転送プログラムを紹介する。

7.1　プレーヤ、プロセッサ再入門

ここまでプレーヤやプロセッサの生成に関して、話を先延ばししてきたことがある。前章までは、プレーヤの生成には Manager.createRealizedPlayer を用い、プロセッサの生成には createRealizedProcessor を用いてきた。これに対し、Manager.createPlayer と Manager.createProcessor という命令がある。しかし、これらを用いるとプログラムが複雑になるため今までは Manager.Realized…の方を使ってきた。様々なサイトのプログラム例を見ると、(必要もないのに) プレーヤやプロセッサの生成が複雑になっている。多くの応用は Manager.createRealized…の方で済むが、独自の機能を持ったプロセッサを作る場合には Manager.createProcessor の方を使う必要がある。



(a)プレーヤの状態　　(b)プロセッサの状態

図 7.1　プレーヤとプロセッサの状態遷移

本節では、特別な機能を持ったプレーヤやプロセッサを作るための、プレーヤやプロセッサの生成法を紹介する。JMF は自分の目的に合ったプロセッサを生成できるようにするため、プロセッサが完成するまでに、”初期状態”から”スタート状態”までに６つの状態を用意し、各ステージで与える情報を規定している（プレーヤの場合は５状態)。即ち、独自の機能を持ったプロセッサを生成するには、プロセッサの状態遷移を管理し、適切なタイミングで必要な機能設定をしていく必要がある。図 7.1 はプロセッサが完成するまでの状態遷移である。

プレーヤの場合を見ると、Unrealized,Realized,Prefetched という３つの安定状態と、それらの状態間を遷移している途中の状態（Realizing,Prefetching）に分かれている。プレーヤやプロセッサを Manager.cratePlayer や Manager.createProcessor で生成すると、図 7.1 の１番上の Unrealize 状態になる。Unrealized 状態にある時、realize()を呼ぶことにより Realizing 状態を経由して Realized 状態に到達する。Realize 状態にある時 prefatch()を呼ぶと Prefetching 状態を経由して Prefatched 状態に到達する。状態が下に遷移するに従い、プレーヤは資源を獲得していき、Prefetch 状態では全ての資源が準備され実行開始直前の状態になる。Player を動作させるには start()命令をだす。前章まで使っていた Manager.createRealizedPlayer は Realized 状態のプレーヤを一気に作るものである。Realized 状態まで進むと、必要な資源は獲得できているので、後は start で実行を開始させられる。プロセッサの場合、さらに configured という状態が加わる（図 7.1 参照)。これは、プロセッサの場合、トラックの制御が必要になるためである。図 7.2 には、プロセッサ生成時に与える情報のタイミングと、取り出し可能になる情報のタイミングが示してある。

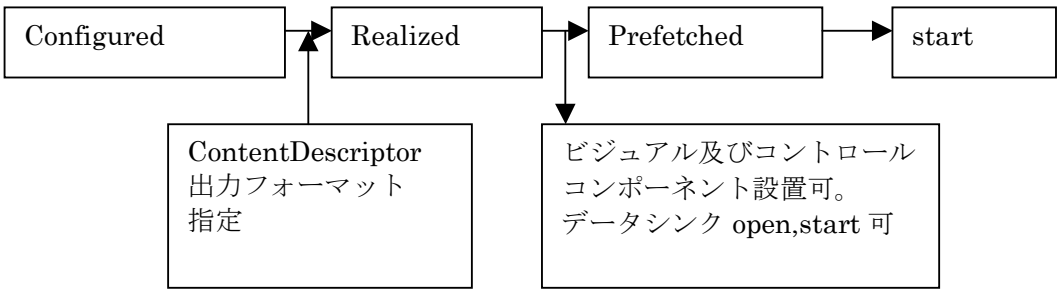


図 7.2　スタートまでの状態遷移（生成過程）

この状態遷移制御が厄介なのは、状態遷移を制御する命令が非同期命令になっている点である。

非同期とは、その命令が完了しない内に命令の実行が次に進んでしまい、命令終了は別の方法で検出しなければならないことである。例えば、図 7.2 に示されるように、Realized にならないとビジュアルコンポーネントが取り出せない規則になっている。そのため、もし Manager.createPlayer でプレーヤを生成し、直後に start()をかけると映像が取り出せないことになる。状態遷移を使う場合、１つの命令実行が済んでからでないと次の命令実行に移ってはまずいため、命令実行終了確認が必要になり、プログラムは複雑になる。

ここで、状態遷移を制御する方法で、よくサンプルプログラムにでてくる２つの方法を紹介する。

（1） リッスナーで状態遷移終了を検知する方法

次に、リッスナーを使ったプログラムの一部を示す。ここではプロセッサにコントローラリッスナーを付け①イベントを監視し、controllerUpdate④でイベント発生時の処理をするものである。①で Unraelize 状態になったプロセッサは③で configured 状態に遷移するよう指示される。しばらくして Configured 状態になるとイベントが発生し④に移る。Configured になると⑤処理が行われ、⑥で Realized 状態に遷移させる。Realized に状態遷移すると再びイベントとなり④飛び⑦で Realized になった時の処理を行う。

```
        processor = Manager.createProcessor( locator );---①
        processor.addControllerListener( this );---②
        processor.configure();---③
                :
                :
                :
                :
public void controllerUpdate( ControllerEvent evt ) {   ---④
    if( evt instanceof ConfigureCompleteEvent ) {  ---⑤
        ここで Configured 時の処理
        processor.realize();---⑥
    }
    else if( evt instanceof RealizeCompleteEvent ) {  ---⑦
        ここで Realized 時の処理
    }
                :
}
```

（2） ステートヘルパーを利用する方法

プロセッサの状態遷移を簡単にするため、JMF-API ガイドの付録に StateHelper というクラスがある。これを StateHelper.java というファイルにして一緒にコンパイルすると状態遷移のプログラムは簡単になる。プロセッサをステートヘルパに渡し、ステートヘルパに状態遷移を管理させると、状態が遷移が完了するまで次の命令に進まない。次のプログラムは、プロセッサの状態を Configured、Realized と状態遷移させるもので、処理をそのままの順で書いている。即ち、ステートヘルパにより状態遷移命令(configure(),realize()など)は、同期命令になる。

```
StateHelper sh = null;
        :
p = Manager.createProcessor(di.getLocator());
sh = new StateHelper(p);
        :
// Configured への状態遷移命令
sh.configure(10000)
 ここに Configured 時の処理
//Realized への状態遷移命令
sh.realize(10000)
 ここに Realized 時の処理
```

上のプログラムでの状態遷移命令の引数 10000 は状態遷移命令の時間切れ時間（ミリ秒）である。

この他にも、状態を管理する方法はいくつか存在するが、上の２つが分かっていれば大体想像は付く。

7.2　映像を JPEG に圧縮してネットワークへ送り出す

今までに音声や映像の圧縮ファイルを実時間で伸張・再生したり、キャプチャーした音声・映像を実時間で圧縮ファイルに格納することが簡単に行われることを示してきた。本節ではキャプチャーした映像を JPEG 圧縮してネットワークに送り出し、受信側では実時間で伸張し再生するプログラムを作成する（図 7.3)。ここではリッスナーを使った状態遷移を利用している。

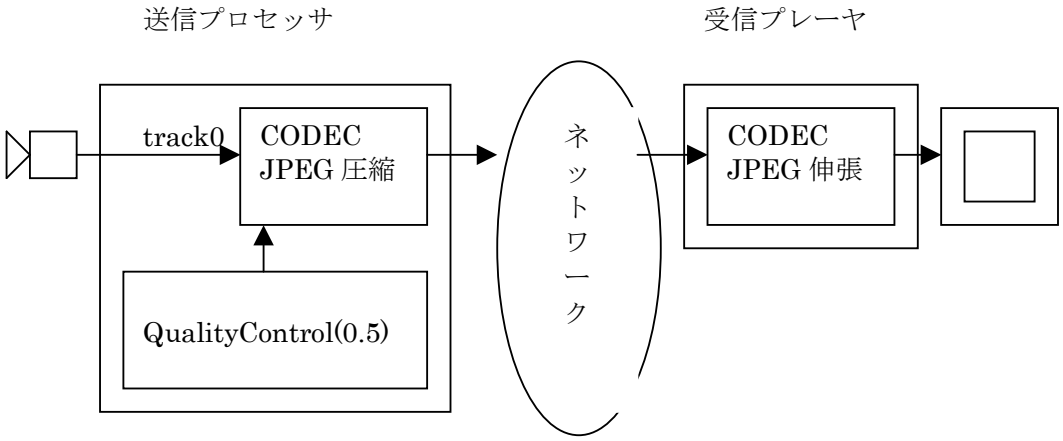


図 7.3　実時間圧縮配信システム

音楽・映像を圧縮する場合プロセッサやプレーヤのトラックに CODEC と呼ばれる機能を付加する。CODEC とは COder/DECoder（符号化―復号化）あるいは COmpression/DECompression（圧縮

―伸張）を短く縮めたもので、送信側では圧縮符号化を行い受信側では伸張復号化を行っている。JMF には標準的な CODEC がいくつの用意されている。今回の実時間圧縮で用いるものは JPEG である。また JPEG 用 CODEC にはコントロールと呼ばれるものの一種である QualityControl を付けて JPEG の品質を制御できる。そのため、送信側プロセッサの生成には様々な設定が必要になり、前章までに使ったプロセッサモデルを用いたプロセッサの生成は使えない。プロセッサの状態遷移を制御しながら情報を与えてプロセッサを生成する。受信側のプレーヤは送られてくるデータのフォーマットを自動的に検出し伸張する。

本節で紹介するプログラムは、キャプチャー映像を JPEG 圧縮（Quality=0.5）するプロセッサを生成し、データシンクでネットワークに送り出す。受信側はプレーヤで受信する、というものである。

圧縮を行うプロセッサの生成手順の流れは次の通りである。
ステップ１：プロセッサの入力データソースを決めて createProcessor でプロセッサ生成
（メディアロケータからの情報により、トラック数と、おおまかなフォーマットが決まる。大まかとは例えば VideoFormat など)。
configured 状態への状態遷移を指示。
ステップ２（Configured になった時)：フォーマットの詳細を決める。例えば、JPEGFormat 詳細など。SetFormat 命令でトラックに詳細フォーマットをセット。
ステップ３：contentDescriptor を生成しプロセッサにセット
realized 状態への遷移を指示。
ステップ４（Realized になった時)：JPEGQuality をセット
ステップ５：プロセッサの出力データソースを作る
ステップ６：シンクの生成
ステップ７：シンクの起動（open(), start())
ステップ８：プロセッサ出力データソース start()

プログラム１１-１のプロセッサ生成は次の図のようになっている。

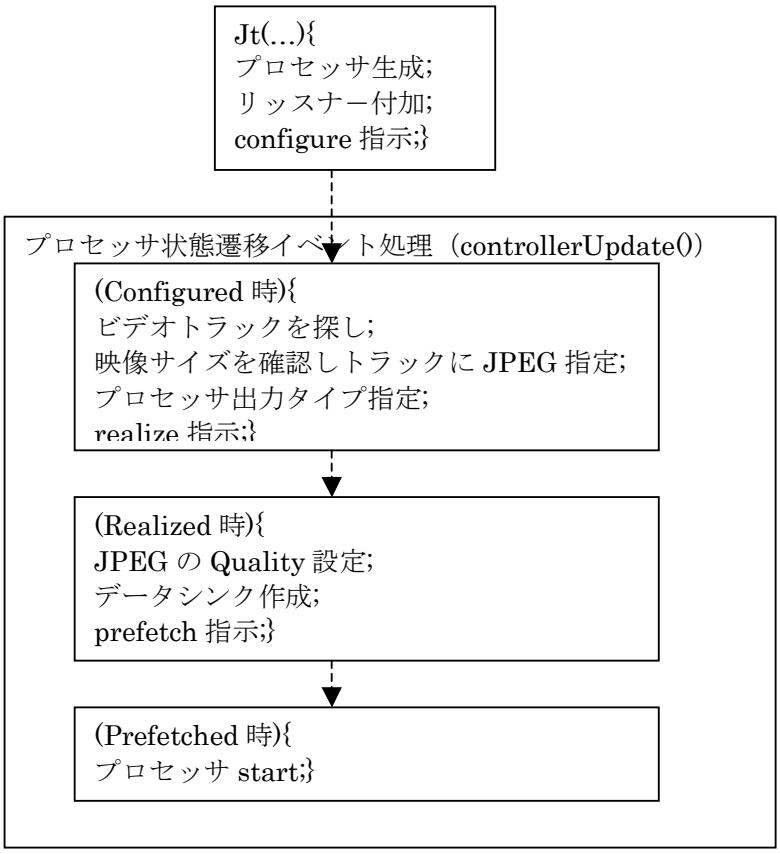


図 7.4　プロセッサ生成時の状態遷移

JPEGQuality 指定
プロセッサの CODEC には色々なコントロールが付いている。例えば BitRateControl, FrameRateControl, H261Control, QualityControl などがある。JPEG の QualityControl を設定する手順は次の通りである。
ステップ１：プロセッサに付いている Control を getControls()で読み出す。
ステップ２：QualityControl がセットされているかを調べ
ステップ３：そのオーナが CODEC なら、CODEC に付属しているフォーマットを調べ、JPEG なら quality を設定する。

プログラム１１-１：　送信プログラム

```
import java.awt.*;
import javax.media.*;
import javax.media.protocol.*;
import javax.media.protocol.DataSource;
import javax.media.format.*;
```

```
import javax.media.control.TrackControl;
import javax.media.control.QualityControl;
import java.io.*;

public class Jt implements ControllerListener {
    MediaLocator locator;
    String ipAddress;
    String port;
    Processor processor = null;
    DataSink  rtptransmitter = null;
    DataSource dataOutput = null;

    public Jt(MediaLocator locator, String ipAddress, String port) {
        this.locator = locator;
        this.ipAddress = ipAddress;
        this.port = port;
        DataSource ds = null;

        try {
                ds = Manager.createDataSource(locator);
                processor = Manager.createProcessor(ds);---①
                processor.addControllerListener(this);
                processor.configure();---②
        } catch (Throwable t) {}
    }

    // 終了処理
    public void stop() {
        synchronized (this) {
                processor.stop();
                processor.close();
                processor = null;
                rtptransmitter.close();
                rtptransmitter = null;
        }
    }

   // JPEG encoder quality を  0.5 ( default) に設定
   void setJPEGQuality(Player p, float val) {
      Control cs[] = p.getControls();
      QualityControl qc = null;
      VideoFormat jpegFmt = new VideoFormat(VideoFormat.JPEG);
      //JPEG encoder の Quality control を探す
      for (int i = 0; i < cs.length; i++) {
```

```
      if (cs[i] instanceof QualityControl && cs[i] instanceof Owned) {
         Object owner = ((Owned)cs[i]).getOwner();
         // owner が Codec なら output format をチェックし Quality をセット.
         if (owner instanceof Codec) {
            Format fmts[] = ((Codec)owner).getSupportedOutputFormats(null);
            for (int j = 0; j < fmts.length; j++) {
               if (fmts[j].matches(jpegFmt)) {
                  qc = (QualityControl)cs[i];
                  qc.setQuality(val);
                  break;
               }
            }
         }
      }
   }
}

 // Controller リッスナー
 public synchronized void controllerUpdate(ControllerEvent event) {
     if (event instanceof ConfigureCompleteEvent) {  ---③
        // プロセッサから tracks を取り出す
        TrackControl [] tracks = processor.getTrackControls();
        boolean programmed = false;
        for (int i = 0; i < tracks.length; i++) {   //video track を探す
           Format format = tracks[i].getFormat();
           if (  tracks[i].isEnabled() &&
                format instanceof VideoFormat && !programmed) {
              // 映像サイズを 8 の倍数にする
              Dimension size = ((VideoFormat)format).getSize();
              float frameRate = ((VideoFormat)format).getFrameRate();
              int w=(int)(size.width / 8) * 8;
              int h=(int)(size.height / 8) * 8;
              VideoFormat jpegFormat = new VideoFormat(VideoFormat.JPEG_RTP,
                         new Dimension(w, h), Format.NOT_SPECIFIED,
                         Format.byteArray, frameRate);
              tracks[i].setFormat(jpegFormat);
              programmed = true;
           } else
              tracks[i].setEnabled(false);
        }
        // content descriptor (RAW_RTP)セット
        ContentDescriptor cd =
                        new ContentDescriptor(ContentDescriptor.RAW_RTP);
        processor.setContentDescriptor(cd);
```

```
            processor.realize();---④
           } else if (event instanceof RealizeCompleteEvent) {  ---⑤
            setJPEGQuality(processor, 0.5f);   // JPEG quality セット
            dataOutput = processor.getDataOutput();
            // 送信先作成、例えば:  rtp://192.168.1.21:20000/video
            String rtpURL = "rtp://" + ipAddress + ":" + port + "/video";
            MediaLocator outputLocator = new MediaLocator(rtpURL);
            try {
              rtptransmitter = Manager.createDataSink(dataOutput, outputLocator);
              rtptransmitter.open();
              rtptransmitter.start();
              dataOutput.start();
              processor.prefetch();---⑥
            } catch(Throwable t) {};
            } else if (event instanceof PrefetchCompleteEvent) {  ---⑦

               processor.start();
            }
         }

      //   main()
      public static void main(String [] args) {
          // "使用法: Jt <sourceURL> <destIP> <destPort>"

          Jt vt = new Jt(new MediaLocator(args[0]), args[1], args[2]);
          try {                //  30 秒間転送
                  Thread.currentThread().sleep(30000);
          } catch (InterruptedException ie) {}
          vt.stop();           // 転送終了
          System.exit(0);
    }
}
```

＜プログラムの解説＞
プログラムは 30 秒間映像を圧縮してネットワークへ送信し、受信側では実時間で伸張して再生する。プログラム構造は Quality 設定が少し複雑だが、基本構造は図 7.4 の通りで①から⑦まで、必要なデータを与えながら順に状態遷移を行っている。

受信側は送られてくるフォーマットに合わせて自動的に復号するため、簡単なプレーヤでよい。

プログラム１１-２：　受信プログラム

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
```

```
import java.io.*;
import javax.media.*;

public class Jr extends Frame {
    String url;
    Player player = null;
    Component visualComponent = null;
    Component controlComponent = null;

    public Jr(String url) {
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        dispose(); System.exit(0); }
      });
    this.url = url;
    MediaLocator mrl= new MediaLocator(url);
    try {
        player = Manager.createRealizedPlayer(mrl);
        visualComponent = player.getVisualComponent();
        add ("Center", visualComponent);
        controlComponent = player.getControlPanelComponent();
        add("South", controlComponent);
        pack();
        setVisible(true);
        player.start();
        } catch (Throwable t) {}
     }

        public static void main(String argv[]) {
        // "使用法: Jr <address>:<port>/{audio|video}[/<ttl>]"

           Jr jr = new Jr("rtp://"+argv[0]);
        }
}
```

＜プログラムの使い方＞
192.168.1.21 から 192.168.1.21 へポート 20000 を使って送る場合
コンパイル
送信側：javac Jt.java
受信側：javac Jr.java

実行
送信側：java Jt vfw://0 192.168.1.21 20000
受信側：java Jr 192.168.1.20:20000/video/1

受信側最後の１は ttl である。

VIII　EFFECT と CODEC の製作
JMF は次の２つの方法で機能拡張ができるようになっている。
（１） トラックに EFFECT や CODEC といった自作 Plug-in を実装することにより
JMF には様々なデータ圧縮、伸張機能が用意されている。これらは、Plug-in という形でシステムに組み込まれている。もし、ユーザが独自の圧縮プログラムなどを組み込みたい場合、新しい CODEC の Plug-in として登録して使用できる。
（２） 新しい DataSource や MediaHandler を実装することにより

EFFECT と CODEC に関しては 7.2 節で簡単に述べた。本章では EFFECT や CODEC を自作する方法を紹介する。EFFECT と CODEC はいずれも 1 入力 1 出力であるが、EFFECT は入力と出力が同じフォーマットで同じサイズのものを言う。例えば、音量を 2 倍にするようなものは EFFECT で、データ圧縮は CODEC となる。本章では EFFECT と CODEC の自作法を紹介するが、マニュアル等にはこれに関する細かな記述が見当たらず（筆者の調査不足かもしれないが）、サンプルプログラムを参考にまとめたものである。

以下の説明では EFFECT を CODEC の 1 形態と捉え、厳密に区別せず EFFECT も CODEC として扱っている部分もある。

8.1　EFFECT と CODEC 作成の際の約束事
EFFECT や CODEC は下図のように入力バッファ，process()，出力バッファの３つの部分を中心に構成されている。

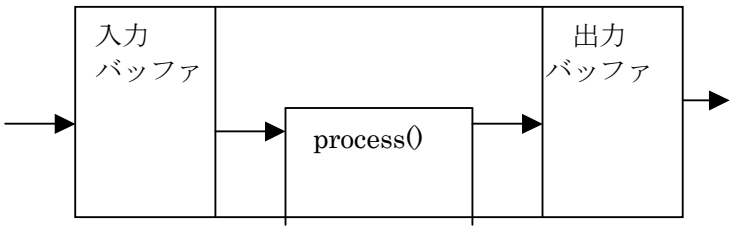


図 8.1　EFFECT,CODEC の構造

EFFECT と CODEC の基本動作は、入力バッファにデータが到着すると process()が処理を開始し、処理結果を出力バッファに書き出す、というものである。
マニュアルによれば、EFFECT や CODEC を自作する場合、次の３つの事が必要になる、とある。
（１） getSupportedInputFormats と getSupportedOutputFormats を作成する。これらのメソッドは、自作する CODEC のサポートする入出力 Format を JMF に知らせるために必要である。
（２） etInputFormat と satOutputFormat を作成。これらのメソッドは、1.の入出力 Format を選択することができるようにするために必要である。
（３） Process()の作成。これは EFFECT や CODEC の処理を記述する部分。

しかし、プログラム例を見るともう少し複雑で次の構成になっている（図 8.2)。

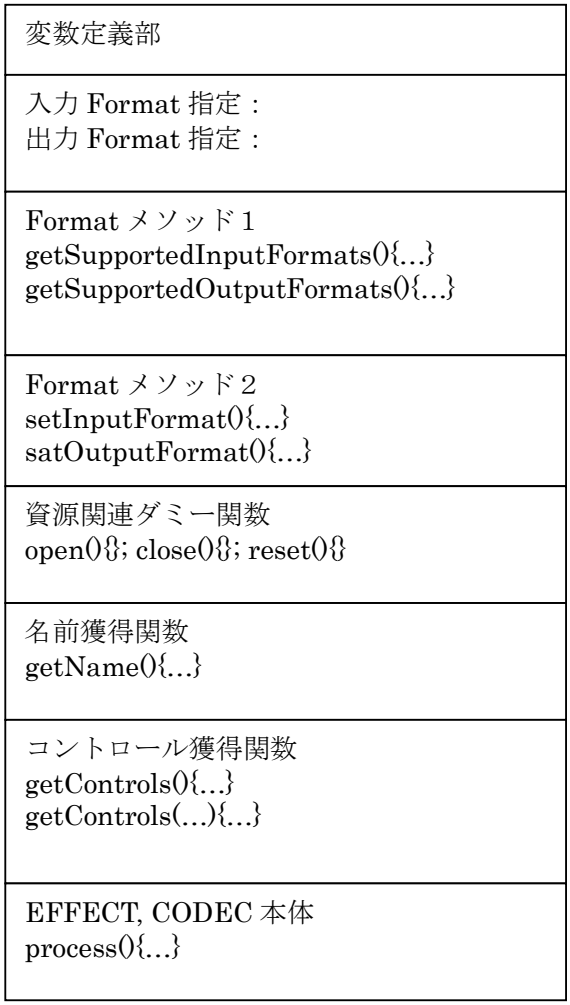


図 8.2　EFFECT,CODEC のプログラム構成

図 8.2 の構成は、ほとんどの EFFECT,CODEC に共通で、process()以外はほとんどおまじない的である。図の上から 2 番目にある入出力 Format 指定は、EFFECT や CODEC の SupportedInputFormat と SuportedOutputFormat と呼ばれる Format を決めるものである。SupportedInputFormat と SuportedOutputFormat は EFFECT や CODEC が受け入れることのできる Format を規定するもので、ここで制限を厳しくすれば、扱える Format が限定される。入出力 Format の詳細は、実際に信号を受信する段階に決められる。

CODEC や EFFECT を作成する上でもう一つ重要な概念“Buffer”がある。Buffer は javax.media

の下のクラスで、マニュアルによれば、“バッファはプロセッサ内の処理ステージ間でメディアデータを運ぶコンテナ”とある。CODEC 内では図 8.1 の入力バッファと出力バッファに使われる。バッファには、（1）長さ、（2）中に入れるデータのフォーマット、（3）タイムスタンプなどの情報が付いている。バッファからのデータを取り出すには、getData()メソッドを使う。注意が必要なのは、バッファへの書き込みである。直接バッファへ書き込むメソッドはなく、setData()メソッドでデータの格納場所を登録し、その格納場所にデータを書き込むことによってバッファにデータを書き込んだと同じことになる。また、バッファには offset というポインタがあり、現在処理している場所を指定するのに利用できる。

以下のプログラムで使われる Buffer 関連の主なメソッド
1．getData():データの取り出し（注意事項：8.3 節参照）
2．setData(a):a を buffer に接続する
（a にデータを入れると buffer に送られる）
3．setFormat(a):buffer のフォーマットを a にする。
4．getLength():長さを求める
5．setLength(a):長さを a にする
6．getOffset():有効なデータの入っている場所を示す
7．setOffset(a):オフセットを a にする

EFFECT や CODEC の処理を記述する部分が process()である。入力バッファ名と出力バッファ名は、 process(入力バッファ,出力バッファ), の引数として指定する。CODEC の基本動作は、入力データが入力バッファにセットされると process()という関数が自動的に動き出し、処理結果を出力バッファに書き出して処理が終了する。この終了の際に return で返す値により process()の動きは大きく変わる。BUFFER_PROCESSED_OK を返した場合が基本で、出力バッファへの書き終わりを意味し、完全に process()を終える。後に出てくる INPUT_BUFFER_NOT_CONSUMED や OUTPUT_BUFFER_NOT_FILLED では繰り返し process()が実行される。

8.2　音声スルーEFFECT プログラム

以下のプログラムは、Sun の JMF サイトにある GainEffect という音量調整 EFEECT プログラムを図 8.2 の構成に合わせ、だきるだけ簡単化したものである。音量も変えていない単なるスループログラムである。これを見ると本質処理（process()）以外のおまじない的部分が非常に多いことが分かる。プログラムは Ge.java, GainEffect.java, StateHelper.java の３つである(Javac Ge.java で全てコンパイルされる)。Ge.java は StateHelper を用いたプロセッサの生成で、プロセッサ生成過程が分かりやすいと思う。今までのプログラムとの違いは、Audio トラックを探し出し、そこに GainEffect という CODEC を取り付けている点である。GainEffect.java は、図 8.2 に沿ってプログラムしたものである。一般に、最初の入出力 SupportedFormat 指定では、受け入れ可能なフォーマットを指定する。即ち、ここの制限を厳しくすると扱えるフォーマットが制限される。

プログラム１２：音声スルーEFFECT

```
import java.awt.*;
import java.awt.event.*;
import javax.media.*;
import javax.media.control.TrackControl;
import javax.media.Format;
import javax.media.format.*;

public class Ge extends Frame {
    Processor p;
    StateHelper sh= null;

    public Ge(MediaLocator ml) {
        try {
            p = Manager.createProcessor(ml);
            sh = new StateHelper(p);
            sh.configure(10000);
            p.setContentDescriptor(null);

          // トッラクを獲得
            TrackControl tc[] = p.getTrackControls();
            TrackControl audioTrack = null;
            for (int i = 0; i < tc.length; i++) {
               if (tc[i].getFormat() instanceof AudioFormat) {
                  audioTrack = tc[i];
                  break;
               }
            }

           // トラックにEFFECTを付加
            Effect effect[] = { new GainEffect() };
            audioTrack.setCodecChain(effect);
            sh.prefetch(10000);    // prefetched状態にする
            Component cc;
            cc = p.getControlPanelComponent();
            add(cc);
            p.start();     // processor start
            setVisible(true);
        } catch (Throwable t) {}
        addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent we) {
              p.close();   System.exit(0);  }});
```

```
    }

    // Main program 使用法: java Ge <url>例 java Ge dsound://
    public static void main(String [] args) {
        String url = args[0];
        MediaLocator ml;
        ml = new MediaLocator(url);
        Ge ge = new Ge(ml);
    }
}

//--GainEffect.java--
import javax.media.*;
 import javax.media.format.*;
 import javax.media.format.AudioFormat.*;

 public class GainEffect implements Effect {
      // EFFECT 名
     private static String EffectName="GainEffect";
      // 使用する入力フォーマット
     protected AudioFormat inputFormat;
      // 使用する出力フォーマット
     protected AudioFormat outputFormat;
      // 使用可能な入力フォーマット
     protected Format[] supportedInputFormats=new Format[0];
      // 使用可能な出力フォーマット
     protected Format[] supportedOutputFormats=new Format[0];

     // フォーマットの初期化
     public GainEffect() {
         supportedInputFormats = new Format[] {
             new AudioFormat(
                 AudioFormat.LINEAR,
                 Format.NOT_SPECIFIED,
                 16,
                 Format.NOT_SPECIFIED,
                 AudioFormat.LITTLE_ENDIAN,
                 AudioFormat.SIGNED,
                 16,
                 Format.NOT_SPECIFIED,
                 Format.byteArray
             )
         };
         supportedOutputFormats = new Format[] {
```

```
        new AudioFormat(
            AudioFormat.LINEAR,
            Format.NOT_SPECIFIED,
            16,
            Format.NOT_SPECIFIED,
            AudioFormat.LITTLE_ENDIAN,
            AudioFormat.SIGNED,
            16,
            Format.NOT_SPECIFIED,
            Format.byteArray
        )
    };
 }

 // 使用可能な入力フォーマット
public Format [] getSupportedInputFormats() {
    return supportedInputFormats;
}
 // 使用する入力フォーマットのための出力フォーマット
public Format [] getSupportedOutputFormats(Format in) {
   Format outs[]= new Format[1];
   outs[0]= in;
   return outs;
}

// 入力フォーマットをセット
public Format setInputFormat(Format input) {
    // the following code assumes valid Format
    inputFormat = (AudioFormat)input;
    return (Format)inputFormat;
}
// 出力フォーマットをセット
public Format setOutputFormat(Format output) {
    // the following code assumes valid Format
    outputFormat = (AudioFormat)output;
    return (Format)outputFormat;
}

// EFFECT に必要な資源の獲得
public void open() {}
public void close() {}
public void reset() {}

// EFFECT 名を返す
```

```
    public String getName() {
        return EffectName;
    }

    // ここでは使用しない
    public Object[] getControls() {
        return (Object[]) new Control[0];
    }

    // コントロールタイプに合ったコントロールを返す
    public Object getControl(String controlType) {
        try {
            Class cls = Class.forName(controlType);
            Object cs[] = getControls();
            for (int i = 0; i < cs.length; i++) {
                if (cls.isInstance(cs[i]))
                return cs[i];
            }
            return null;
        } catch (Exception e) {
            return null;
        }
    }


// ここからがEFFECTの本体
    public int process(Buffer inputBuffer, Buffer outputBuffer){---①

        // 前処理
        byte[] inData = (byte[])inputBuffer.getData();---②
        int inLength = inputBuffer.getLength();
        byte[] outData= new byte[inLength];---③
        outputBuffer.setData(outData);---④

        // 主処理 (スルー)
         for (int i=0; i< inLength; i++) ---⑤
            outData[i] = inData[i];

        // 後処理
        outputBuffer.setFormat(outputFormat);
        outputBuffer.setLength(inLength);
        return BUFFER_PROCESSED_OK;---⑥
    }
}
```

＜プログラムの解説＞
① 以下が CODEC の本質的処理
② では入力バッファに送られてきたデータを inData[]に取り込む
③ で outData[]という配列を作り、④で出力バッファに接続している
④ では inData[]のデータを outData[]に移しているが、これによりデータは出力バッファに送られる。⑥で CODEC の処理終了を報告する。

＜プログラムの使い方＞
用意するプログラム
Ge.java
GainEffect.java
StateHelper.java　--- (Sun 社の HP から入手可能)

コンパイル
javac Ge.java

実行
java Ge dsound://

8.3　映像処理 EFFECT プログラム
次は映像処理プログラムである。映像の 1 部を黒くして表示している。これは Sun のサイトにあった FrameAccess をベースにしている。映像処理を行いたい人へのヒントになるかと考え、映像の 1 部を黒く表示している。
このプログラムはプログラム１２と扱うフォーマット以外よく似ているが、CODEC は内部クラスとして定義している。

プログラム１３：映像処理
```
import java.awt.*;
import javax.media.*;
import javax.media.control.TrackControl;
import javax.media.Format;
import javax.media.format.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

public class Fa extends Frame  {
    Processor p;
    StateHelper sh= null;

    boolean open(MediaLocator ml) {
        try {
            p = Manager.createProcessor(ml);
```

```
            sh = new StateHelper(p);
            sh.configure(10000);
            p.setContentDescriptor(null);
            TrackControl tc[] = p.getTrackControls();
            TrackControl videoTrack = null;
            for (int i = 0; i < tc.length; i++) {
               if (tc[i].getFormat() instanceof VideoFormat) {
                  videoTrack = tc[i];
                  break;
               }
            }
        System.err.println("Video format: " + videoTrack.getFormat());
            Codec codec[] = { new DrawLineCodec()};
            videoTrack.setCodecChain(codec);
            sh.prefetch(10000);    // prefetched 状態にする
            setLayout(new BorderLayout());
            Component cc;
            Component vc;
            if ((vc = p.getVisualComponent()) != null) {
              add("Center", vc);
            }
            if ((cc = p.getControlPanelComponent()) != null) {
              add("South", cc);
            }
            pack();
            p.start();
            setVisible(true);
            } catch (Throwable t) {}
            addWindowListener(new WindowAdapter() {
               public void windowClosing(WindowEvent we) {
                  p.close(); System.exit(0);}});
        return true;
    }

     /* Main program*/
    public static void main(String [] args) {
     //使用法  java Fa vfw0://
        String url = args[0];
        MediaLocator ml= new MediaLocator(url);
        Fa fa = new Fa();
        fa.open(ml);
    }

//  内部クラス
```

```
public class DrawLineCodec implements Codec {
    // 全てのvideo formatsを扱えるよう広く宣言
    protected Format supportedIns[] = new Format [] {
        new VideoFormat(null)
    };
    protected Format supportedOuts[] = new Format [] {
        new VideoFormat(null)
    };

    protected Format input = null, output = null;

    public Format [] getSupportedInputFormats() {
        return supportedIns;
    }
    public Format [] getSupportedOutputFormats(Format in) {
        if (in == null){
            return supportedOuts;}
        else {
            // input format が与えられたら output formatもそれに合わせる
            Format outs[] = new Format[1];
            outs[0] = in;
            return outs;
        }
    }

    public Format setInputFormat(Format format) {
        input = format;
        return input;
    }
    public Format setOutputFormat(Format format) {
        output = format;
        return output;
    }

    public void open() {}
    public void close() {}
    public void reset() {}

    public String getName() {
        return "DrawLine Codec";
    }

    public Object[] getControls() {
        return new Object[0];
```

```
        }
        public Object getControl(String type) {
            return null;
        }

        public int process(Buffer in, Buffer out) {

           byte []  inData = (byte[]) in.getData();

           out.setData(inData);
            // 映像に黒帯を入れる
           for(int i=8000; i<14000; i++) inData[i]= 0x00;

     /*    重ね書き---①
           byte[] outData= (byte[]) out.getData();
           for(int i=30000; i<40000; i++) outData[i]= 0x00;  */

           out.setFormat(in.getFormat());
           out.setLength(in.getLength());
           out.setOffset(in.getOffset());

           return BUFFER_PROCESSED_OK;
        }
    }
}
```

＜プログラムの解説＞
このプログラムで getData()に関する注意点を示す。Process()の中で

```
           byte []  inData = (byte[]) in.getData();
```

という命令があり、入力バッファのデータを inData[]に取り込んでいる。これに対しプログラム中で“重ね書き①”としてコメントアウトしている部分があり、このコメントアウトをはずすと２本の線が描かれる。即ち、out.getData()では out.setData()で出力バッファにセットした inData[]が返され inData[]と outData[]は同一のものとなる（どちらにデータを入力しても同時に他方に書かれて出力バッファに送られる、というより inData と outData がポインタで出力バッファを指し、データは出力バッファに書かれている感じ)。

＜プログラムの使い方＞
準備するプログラム
Fa.java
StateHelper.java
コンパイル
javac Fa.java
実行
java Fa vfw://0

8.4　圧縮・伸張 CODEC

独自の圧縮プログラムを開発したい場合などは、専用の CODEC を開発し、Plug-in の形で JMF に組み込む。また、圧縮データをインターネットに送り出すには、映像データをネットワーク転送用パケットに分解するパケッタイザ CODEC を開発する必要がある。受信側では圧縮されて送られてくる映像を伸張するため、送信側の逆、即ちパケットから圧縮データを再現するデパケッタイザ CODEC と、圧縮データから映像を作る伸張 CODEC が必要になる(図 8.3)。

本節では、自作圧縮 CODEC を作る場合の基本要素含む簡単なプログラムを作成した。プログラムはスウェーデン Lule 大学の mDec プログラムを参考にした。ここで作成するプログラムは、実際には圧縮を行っていないが、圧縮プログラムを作成する必要最小限の機能を持っている。

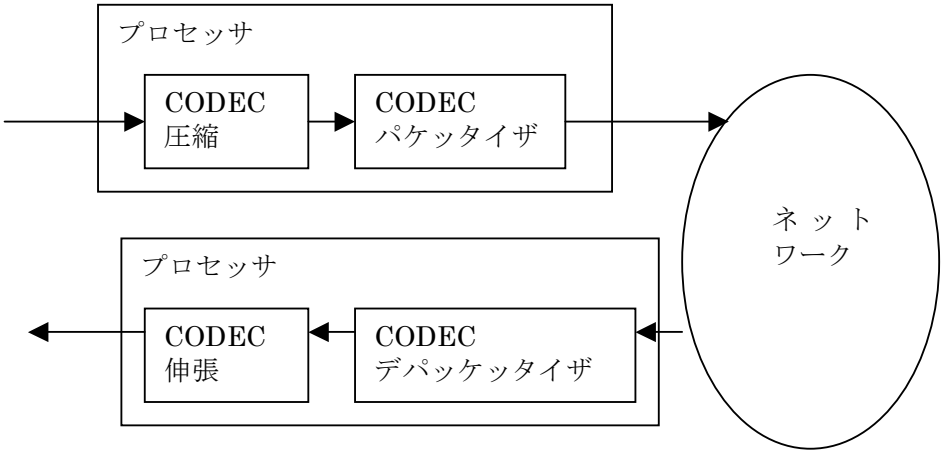


図 8.3　独自の圧縮・伸張

図 8.4　は、その構成を示している。送信側はキャプチャーしたフレームを Encoder で圧縮してパケッタイザに送る。今回のプログラムでは圧縮をせずフレームの下部分(320*24*3 バイト)だけをパケッタイザに送っている。パケッタイザは受け取ったデータを 480 バイトのパケットに分割しネットワークに送り出す。受信側のデパケッタイザは受け取ったパケットをまとめて圧縮されたデータ（本節のプログラムではフレームの下部分）を復元する。Decoder は圧縮されたデータから元のフレームを復元する（このプログラムでは画像の下部分がそのまま画面に表示される)。

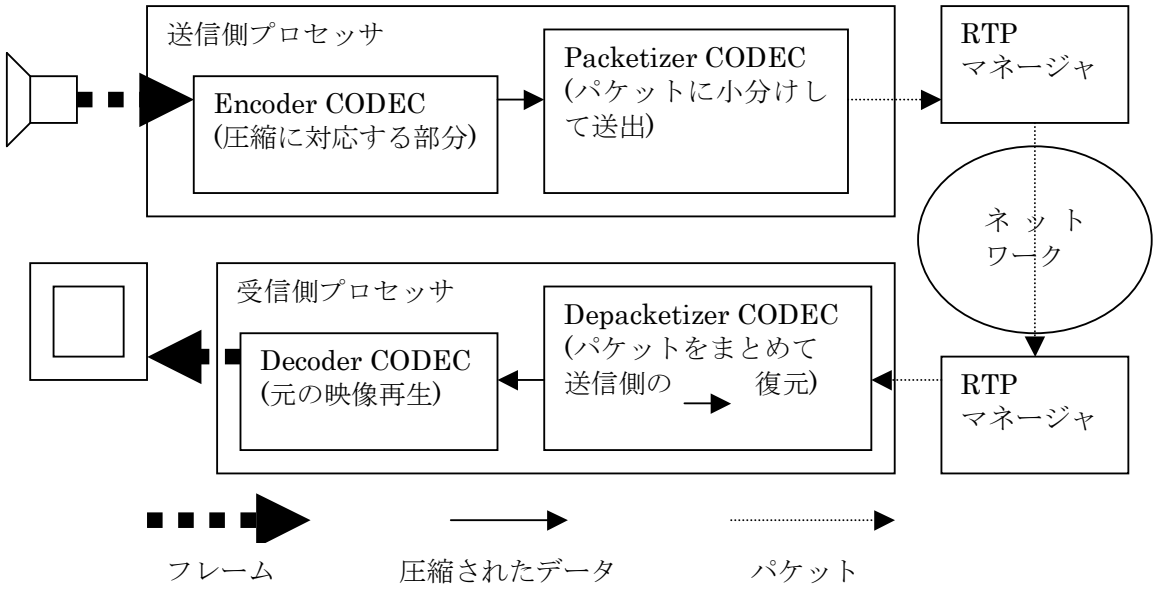
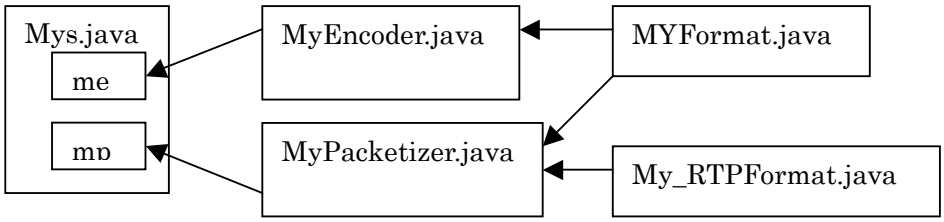


図　8.4　映像圧縮転送を真似たプログラム

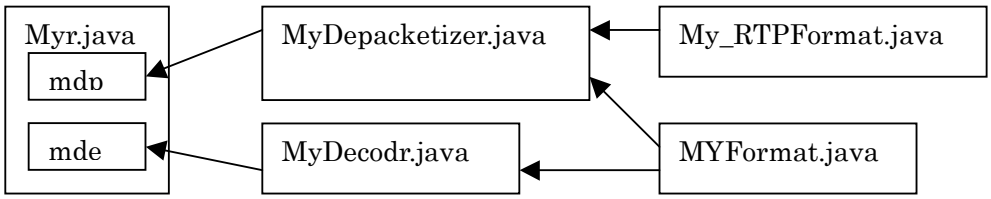
プログラムの構成は図 8.5　である。送信側の主プログラムは Mys.java である。ここに２つの CODEC(MyEncoder.java と MyPacketizer)が組み込まれる。受信側の主プログラムは Myr.java で、ここに２つの CODEC(MyDepacketizer.java と MyDecoder.java)が組み込まれる。各 CODEC には次の様な入出力 Format が設定される。

入力 Format	CODEC 名	出力 Format
RGBFormat	MyEncoder	MYFormat
MYFormat	MyPacketizer	MY_RTPFormat
MY_RTPFormat	MyDepacketizer	MYFormat
MYFormat	MyDecoder	RGBFormat

プログラムとの関係は図　8.6　である。



(a)　送信プログラム



(b)　受信プログラム

図 8.5　プログラムの構成

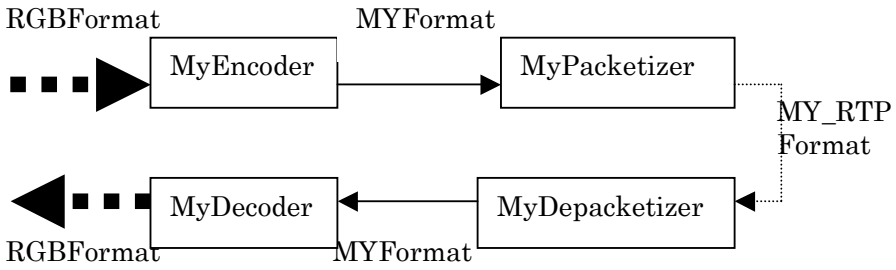


図 8.6　　CODEC とフォーマットの変遷

送信側プログラム
Mys. java の中で CODEC に関係する主な命令の実行順は次の通りである。
ステップ 1-1：EncoderCODEC 生成
ステップ 1-2：PacketizerCODEC 生成
ステップ 2-1：プロセッサ生成開始
ステップ 2-2：プロセッサのビデオトラックに２つの CODEC を設置
ステップ 2-3：ビデオトラックの Format を Packetizer の出力 Format にする

ステップ 3-1：RTP マネージャ生成
ステップ 3-2：RTP マネージャの Format を Packetizer の出力 Format にする。
ステップ 4：Stream を生成しスタート
ステップ 5：プロセッサスタート

プログラム１４－１　圧縮転送用送信プログラム

```
//…Mys.java
import java.awt.*;
import javax.media.*;
import javax.media.control.TrackControl;
import javax.media.Format;
import javax.media.format.*;
import java.awt.event.*;
import javax.media.rtp.*;
import javax.media.rtp.rtcp.*;
import java.net.InetAddress;
import javax.media.protocol.*;

public class Mys extends Frame  {
    public static int MY_PAYLOAD = 101;
    Processor p;
    StateHelper sh= null;
    static MyEncoder me = new MyEncoder();
    static MyPacketizer  mp = new MyPacketizer();
    private Format myFormat;

    RTPManager rtpMgrs[];
    public boolean open(MediaLocator ml) {
        try {
            p = Manager.createProcessor(ml);
            sh = new StateHelper(p);
            sh.configure(10000);
            myFormat = (mp.getSupportedOutputFormats(null))[0];

            TrackControl tc[] = p.getTrackControls();
            TrackControl videoTrack = null;
            for (int i = 0; i < tc.length; i++) {
               if (tc[i].getFormat() instanceof VideoFormat) {
                  videoTrack = tc[i];
                  break;
               }
            }
        System.err.println("Video format: " + videoTrack.getFormat());
```

```
            Codec codec[] =  {me,mp};
            videoTrack.setCodecChain(codec);
            videoTrack.setFormat(new MY_RTPFormat());
            sh.prefetch(10000);    // prefetched 状態にする

        PushBufferDataSource pbds = (PushBufferDataSource)p.getDataOutput();
        PushBufferStream pbss[] = pbds.getStreams();

        rtpMgrs = new RTPManager[pbss.length];
        SessionAddress localAddr, destAddr;
        String ipAddress="224.1.1.1";
        InetAddress ipAddr;
        SendStream sendStream;
        int portBase= 2000;
        int port;
        SourceDescription srcDesList[];

        for (int i = 0; i < pbss.length; i++) {
           rtpMgrs[i] = RTPManager.newInstance();
           rtpMgrs[i].addFormat(myFormat, MY_PAYLOAD);
           port = portBase + 2*i;
           ipAddr = InetAddress.getByName(ipAddress);
           localAddr = new SessionAddress( ipAddr, port);
           destAddr = new SessionAddress( ipAddr, port);
           rtpMgrs[i].initialize( localAddr);
           rtpMgrs[i].addTarget( destAddr);
           System.err.println("Created RTP session: " + localAddr + " "+ port);
           sendStream = rtpMgrs[i].createSendStream(p.getDataOutput(), i);
           sendStream.start();

           Component cc;
           if ((cc = p.getControlPanelComponent()) != null) {
              add(cc);
           }
           pack();
           p.start();
           setVisible(true);
           }
        } catch (Throwable t) {}
        addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent we) {
              p.close(); System.exit(0);}});
        return true;
    }
```

```
     /* Main program*/
    public static void main(String [] args) {
     //使用法  java Mys vfw0://又は java Mys file:/c:/foo.mov
        String url = args[0];
        MediaLocator ml= new MediaLocator(url);
        Mys Mys = new Mys();
        Mys.open(ml);
    }
}
```

エンコーダ
このプログラムは映像１フレームを受け取り、画面下部のデータを次の CODEC（パケッタイザ）に送るだけである。
この CODEC では、おまじないの部分（process()以外）が、前節の EFFECT より少し丁寧になっている。具体的には match()という関数を用意し、入出力フォーマットをサポーテッドフォートと比較して処理している。

また、このプログラムでは新たに次の２つの命令が使われる。

１．自作 Plug-in の追加
PlugInManager.addPlugIn(String className, Format[] in, Format[] out, int type);
ClassName:Plug-in 名
in:Plug-in がサポートする入力フォーマット
out:Plug-in がサポートする出力
type:Plug-in のタイプ。例えば DEMULTIPLEXER, CODEC, EFFECT, MULTIPLEXER, or RENDERER
２．追加した Plug-In の登録（これを行わないと、セッションが終わると消える）
PlugInManager.commit();

```
//…MyEncoder.java
import javax.media.*;
import javax.media.format.*;
import java.awt.*;
import java.io.IOException;

public class MyEncoder implements Codec {

    private Format inputFormat;
    private Format outputFormat;
    private Format[] supportedInputFormats;
    private Format[] supportedOutputFormats;
```

```
    static String PLUGIN_NAME = "My encoder";

    public MyEncoder() {
       supportedInputFormats = new Format[] {
            new RGBFormat(null,
                          Format.NOT_SPECIFIED,
                          Format.byteArray,
                          Format.NOT_SPECIFIED,
                                    24,
                          3, 2, 1,
                          3, Format.NOT_SPECIFIED,
                          Format.TRUE,
                          Format.NOT_SPECIFIED)
                };

            supportedOutputFormats = new Format[] {
                new MYFormat()
                    };

            // プラグイン追加
            PlugInManager.addPlugIn(PLUGIN_NAME,
supportedInputFormats,supportedOutputFormats, PlugInManager.CODEC);
            try{
                PlugInManager.commit();
            }
            catch(IOException e){
                System.out.println("IOException occured");
                e.printStackTrace();
                System.exit(1);
            }
    }

    static public boolean matches(Format input, Format supported[]) {
        for (int i = 0; i < supported.length; i++) {
            if (input.matches(supported[i]))
                return true;
        }
        return false;
    }

    public Format [] getSupportedInputFormats() {
        return supportedInputFormats;
    }
```

```
public Format [] getSupportedOutputFormats(Format input) {
    if (input == null || matches(input, supportedInputFormats))
        return supportedOutputFormats;
    return new Format[0];
}

public Format setInputFormat(Format format) {
    if (!matches(format, supportedInputFormats))
        return null;
    inputFormat = (RGBFormat) format;
    return format;
}

public Format setOutputFormat(Format format) {
    if (!matches(format, supportedOutputFormats))
        return null;
    System.out.println(format.toString());
    outputFormat = format;
    return format;
}

public String getName() {
    return PLUGIN_NAME;
}

public void open() {}
public void close() {}
public void reset() {}

public Object getControl(String controlType) {
    return null;
}

public Object[] getControls() {
    return null;
}

public synchronized int process(Buffer inBuffer, Buffer outBuffer) {

    byte [] inData = (byte[]) inBuffer.getData();
    byte [] outData = new byte[320*24*3];

    System.arraycopy(inData,0,outData,0,320*24*3);
```

```
        outBuffer.setFormat(outputFormat);
        outBuffer.setData(outData);
        outBuffer.setLength(320*24*3);
        return BUFFER_PROCESSED_OK;
    }
}
```

パケッタイザ
映像１フレームの下部データを 482 バイトのパケット(データ 480 バイト＋ヘッダ 2 バイト)に分割し、ネットワークに次々と送り出す。プログラムは１フレーム分を送り終えるまで新しいフレームの到着を受け入れてはいけない。そのため１フレームの送出が終わるまで、process()を INPUT_BUFFER_NOT_CONSUMED でぬけている。これでぬけると、出力はネットワークに送り出すが、入力は受け入れない形で process()を繰り返し実行することができる。
パケットは 480 バイトのデータ (payload) と２バイトのヘッダからなる。ヘッダの第１バイト目はフレーム番号を示す（０～２５５を繰り返す)。ヘッダの第２バイト目は、そのパケットがフレームの最後のパケットか否かを示す。

```
//…MyPacketizer.java
import javax.media.*;
import javax.media.format.*;
import java.awt.*;
import java.io.IOException;

public class MyPacketizer implements Codec{

    //パケットヘッダの”フレーム最後のパケットか”を示すフラグ
    static final int LAST_PACKET = 1;
    static final int NOT_LAST = 0;

    private Format inputFormat;
    private Format outputFormat;
    private Format[] supportedInputFormats;
    private Format[] supportedOutputFormats;

    //パケットの Sequence number (0-255)
    private int seqNo = 0;

    static String PLUGIN_NAME = "My packetizer";

    //パケットサイズ＝ヘッダサイズ＋データサイズ
    static int DATA_SIZE = 480;
    static int HDR_SIZE = 2;
    static int PACKET_SIZE = DATA_SIZE + HDR_SIZE;
```

```
    public MyPacketizer() {
        supportedInputFormats = new Format[] {
            new MYFormat()
        };

        supportedOutputFormats = new Format[] {
            new MY_RTPFormat()
        };

        PlugInManager.addPlugIn(PLUGIN_NAME,
supportedInputFormats,supportedOutputFormats, PlugInManager.CODEC);
    }

    public Format [] getSupportedInputFormats() {
                return supportedInputFormats;
    }

    public Format [] getSupportedOutputFormats(Format input) {
                if (input == null || matches(input, supportedInputFormats))
                        return supportedOutputFormats;
                return new Format[0];
    }

    public Format setInputFormat(Format format) {
        if (!matches(format, supportedInputFormats))
            return null;
                inputFormat = format;
        return format;
    }

    public Format setOutputFormat(Format format) {
                if (!matches(format, supportedOutputFormats))
                        return null;
                outputFormat = format;
                return format;
    }

    public String getName() {
        return PLUGIN_NAME;
    }

    public void open() {}
    public void close() {}
    public void reset() {}
```

```
 public Object getControl(String controlType) {
    System.out.println(controlType);
        return null;
}

public Object[] getControls() {
    return null;
}

static public boolean matches(Format input, Format supported[]) {
    for (int i = 0; i < supported.length; i++) {
        if (input.matches(supported[i]))
            return true;
    }
    return false;
}

public synchronized int process(Buffer inBuffer, Buffer outBuffer) {

    byte[] inData = (byte[])inBuffer.getData();
    byte[] outData = (byte[])outBuffer.getData();
    int inLength = inBuffer.getLength();

    if (outData == null || outData.length < PACKET_SIZE) {
       outData = new byte[PACKET_SIZE];
       outBuffer.setData(outData);
    }

    outBuffer.setLength(PACKET_SIZE);
    outBuffer.setFormat(outputFormat);
    outBuffer.setOffset(0);

    outData[0] = (byte) ((seqNo = seqNo%256) & 0xFF);// 続き番号 set.max255

    if (inLength > DATA_SIZE) {    // フレーム最後のパケットか否か
       System.arraycopy(inData, inBuffer.getOffset(),
                                    outData, HDR_SIZE, DATA_SIZE);
       inBuffer.setOffset( inBuffer.getOffset() + DATA_SIZE);
       inBuffer.setLength( inLength - DATA_SIZE);

       outBuffer.setFormat(outputFormat);
       outBuffer.setLength(PACKET_SIZE);
       outBuffer.setOffset(0);
```

```
            outData[1] = NOT_LAST; // 続きあり
            return INPUT_BUFFER_NOT_CONSUMED ;
          }
          //最後のパケットの場合
          System.arraycopy(inData, inBuffer.getOffset(),
                           outData, HDR_SIZE, inLength);

          outBuffer.setFormat(outputFormat);
          outBuffer.setLength(inLength + HDR_SIZE);
          outBuffer.setOffset(0);
          seqNo++;
          outData[1] = LAST_PACKET;  // １フレーム最後のパケット
         return BUFFER_PROCESSED_OK;
         }
}
```

受信プログラム：
Myr.java の中で CODEC に関係する主な命令の実行順は次の通りである。
ステップ 1-1：DecoderCODEC 生成
ステップ 1-2：DepacketizerCODEC 生成
ステップ 2：RTP マネージャを生成し、ストリームの到着を StreamListener で待つ
＜ストリーム到着時（update()）の処理＞
ステップ 3-1：プロセッサ生成開始
ステップ 3-2：ビデオトラックにステップ１で準備した２つの CODEC を設置
ステップ 3-3：ビデオトラックの Format を RGBFormat にする。
ステップ 4：プロセッサスタート

プログラム１４－２　受信プログラム

```
//Myr.java
import java.io.*;
import java.awt.*;
import java.net.*;
import java.awt.event.*;
import javax.media.*;
import javax.media.rtp.*;
import javax.media.rtp.event.*;
import javax.media.rtp.rtcp.*;
import javax.media.protocol.*;
import javax.media.control.TrackControl;
import javax.media.Format;
import javax.media.format.*;
import java.net.InetAddress;
```

```
public class Myr extends Frame implements ReceiveStreamListener
{
    public static int MY_PAYLOAD = 101;
    Processor p= null;
    RTPManager mgrs = null;
    StateHelper sh;
    MyDecoder mde = new MyDecoder();
    MyDepacketizer mdp = new MyDepacketizer();
    static Format MyFormat;

    public Myr() {
        MyFormat = (mdp.getSupportedInputFormats())[0];
        String sessionip= "224.1.1.1";
        int sessionport= 2000;
        int sessionttl=1;
        try {
            InetAddress ipAddr;
            SessionAddress localAddr;
            SessionAddress destAddr;
            System.err.println("  - Open RTP session for: addr: " + sessionip + " port:
" + sessionport + " ttl: " + sessionttl);
            mgrs = (RTPManager) RTPManager.newInstance();
            mgrs.addFormat(MyFormat, MY_PAYLOAD);
            mgrs.addReceiveStreamListener(this);
            ipAddr = InetAddress.getByName(sessionip);
        //マルチキャストでは local と remote は同じ
            localAddr= new SessionAddress( ipAddr, sessionport, sessionttl);
            destAddr = new SessionAddress( ipAddr, sessionport, sessionttl);
            mgrs.initialize(localAddr);
            mgrs.addTarget(destAddr);
        //１分間ストリームの到着待ち
            System.err.println("  - Waiting for RTP data to arrive...");
            Thread.sleep(60000);
        } catch (Throwable t) {}
    }

    public synchronized void update(ReceiveStreamEvent evt){
        ReceiveStream stream = evt.getReceiveStream();
        if (evt instanceof NewReceiveStreamEvent) {
            try {
                stream = ((NewReceiveStreamEvent)evt).getReceiveStream();
                DataSource ds = stream.getDataSource();
                p = Manager.createProcessor(ds);
```

```
            sh = new StateHelper(p);
            sh.configure(10000);
            p.setContentDescriptor(null);

            // トラックコントロール取得
            TrackControl tc[] = p.getTrackControls();
            TrackControl videoTrack = null;

            for (int i = 0; i < tc.length; i++) {
                    if (tc[i].getFormat() instanceof MY_RTPFormat) {
                    videoTrack = tc[i];
                    break;
                    }
            }

            // CODEC 設置
            Codec codec[] = {mdp, mde};
            videoTrack.setCodecChain(codec);
            videoTrack.setFormat(new RGBFormat());
            sh.prefetch(10000);

            setLayout(new BorderLayout());
            Component cc;
            Component vc;
            if ((vc = p.getVisualComponent()) != null) {
              add("Center", vc);
            }
            if ((cc = p.getControlPanelComponent()) != null) {
              add("South", cc);
            }
            pack();
            p.start();
            setVisible(true);
        } catch (Throwable t) {
            System.err.println(" connection error of new stream");
            return;
        }
    }
}

public static void main(String argv[]) {
    Myr receiver = new Myr();
}
```

}

デパケッタイザ
このプログラムは、パケット（482 バイト）をためていって映像１フレームの下部データを再現する。CODEC の本体である process()では、１フレームのデータを作り上げるまで OUTPUT_BUFFER_NOT_FILLED で process()をぬける。このようにして process()をぬけると CODEC の出力バッファにデータがそろってないことを意味し、バッファの中身が次の CODEC（デコーダ）に送られない。そして再び入力にパケットが到着すると process()が動作する。
ネットワークを介した RTP データ転送ではパケットの追い越し、消滅などが発生する。本プログラムでは、そのような場合の修復処理は行っていない。単に、パケットエラーを表示している。筆者の経験では、処理速度の速いコンピュータから処理速度の遅いコンピュータに映像を送ったとき多くのエラーが発生した。

```
//…MyDepacketizer.java
import javax.media.*;
import javax.media.format.*;
import java.awt.*;
import java.io.IOException;

public class MyDepacketizer implements Codec {

    static String PLUGIN_NAME = "My depacketizer";
    static int HDR_SIZE = 2;
    static int MAX_OUT_DATA = 320*240*4;

    private Format inputFormat;
    private Format outputFormat;
    private Format[] supportedInputFormats;
    private Format[] supportedOutputFormats;

    // Debug 用
    private int currentFrame = -1;
    private int counter = 0;
    private boolean debug = true;

    public MyDepacketizer() {
       supportedInputFormats = new Format[] {
          new MY_RTPFormat()
       };

       supportedOutputFormats = new Format[] {
          new MYFormat()
       };
```

```
       PlugInManager.addPlugIn(PLUGIN_NAME,
supportedInputFormats,supportedOutputFormats, PlugInManager.CODEC);
    }

    public String getName() {
      return PLUGIN_NAME;
    }

    static public boolean matches(Format input, Format supported[]) {
        for (int i = 0; i < supported.length; i++) {
            if (input.matches(supported[i]))
                return true;
        }
        return false;
    }

    public Format [] getSupportedInputFormats() {
                return supportedInputFormats;
    }

    public Format [] getSupportedOutputFormats(Format input) {
                if (input == null || matches(input, supportedInputFormats)) {
                        return supportedOutputFormats;
                }
                return new Format[0];
    }

    public Format setInputFormat(Format format) {
        if (!matches(format, supportedInputFormats))
                return null;
        inputFormat = format;
        return format;
    }

    public Format setOutputFormat(Format format) {
                if (!matches(format, supportedOutputFormats))
                        return null;
                outputFormat = format;
                return format;
    }

    public void open() {}
    public void close() {}
```

```
public void reset() {}

public Object[] getControls() {
    return new Object[0];
}

public Object getControl(String type) {
    return null;
}

public int process(Buffer inBuffer, Buffer outBuffer) {

    counter++;
    byte[] inData = (byte[]) inBuffer.getData();
    byte [] outData;

    if(outBuffer.getData() == null){
       outData = new byte[MAX_OUT_DATA];
       outBuffer.setData(outData);
    }
    else{
       outData = (byte[]) outBuffer.getData();
    }
    // ストリーム終了
    if (inBuffer.isEOM()) {
       outBuffer.setData(outData);
       outBuffer.setLength(0);
       outBuffer.setEOM(true);
       return BUFFER_PROCESSED_OK;
    }

    int in_offset = inBuffer.getOffset();
    int out_offset = outBuffer.getOffset();
    int frameStatus = inData[in_offset + 1];

    // フレーム最初のパケット？
    if(currentFrame == -1)
       currentFrame = (inData[in_offset+0] & 0xFF);

    // 他のフレーム用パケット
    if(currentFrame != (inData[in_offset+0] & 0xFF)){
       if(debug)
            System.out.println("Pac_Err"+currentFrame+";"+inData[in_offset+0]);
       currentFrame = -1;
```

```
            counter = 0;
            return INPUT_BUFFER_NOT_CONSUMED;
        }

        // フレーム末パケットの場合
        if(frameStatus == MyPacketizer.LAST_PACKET){
            currentFrame = -1;

System.arraycopy(inData,in_offset+MyPacketizer.HDR_SIZE,outData,out_offset,inBuffer.
getLength()-MyPacketizer.HDR_SIZE);
            outBuffer.setLength(outBuffer.getLength()    +    inBuffer.getLength()    -
MyPacketizer.HDR_SIZE);
            counter = 0;
            return BUFFER_PROCESSED_OK;
        }

        // フレーム末以外
        System.arraycopy(inData,in_offset+MyPacketizer.HDR_SIZE,outData,out_offset,i
nBuffer.getLength()-MyPacketizer.HDR_SIZE);
        outBuffer.setLength(outBuffer.getLength()       +       inBuffer.getLength()-
MyPacketizer.HDR_SIZE);
        outBuffer.setOffset(outBuffer.getOffset()     +     inBuffer.getLength()     -
MyPacketizer.HDR_SIZE);
        return OUTPUT_BUFFER_NOT_FILLED;
    }
}
```

デコーダ
これは単にデパケッタイザから送られてくるデータをウインドウに表示するだけである。

```
//…MyDecoder.java
import javax.media.*;
import javax.media.format.*;
import java.awt.*;
import java.io.IOException;

public class MyDecoder implements Codec {

    private Format inputFormat;
    private Format outputFormat;
    private Format[] supportedInputFormats;
    private Format[] supportedOutputFormats;

    static String PLUGIN_NAME = "My decoder";
```

```
        public MyDecoder() {
        supportedInputFormats = new Format[] {
           new MYFormat()
        };

        supportedOutputFormats = new Format[] {
            new RGBFormat(new java.awt.Dimension(320,240),320*240*3,
                                Format.byteArray,
                                15,
                                24,
                                3, 2, 1,
                                3, 320*3, //!"!!
                                Format.TRUE,
                                1)
           };

        PlugInManager.addPlugIn(PLUGIN_NAME,
supportedInputFormats,supportedOutputFormats, PlugInManager.CODEC);
        }

     static public boolean matches(Format input, Format supported[]) {
        for (int i = 0; i < supported.length; i++) {
            if (input.matches(supported[i]))
                return true;
        }
        return false;
    }

    public Format [] getSupportedInputFormats() {
                return supportedInputFormats;
    }

    public Format [] getSupportedOutputFormats(Format input) {
                if (input == null || matches(input, supportedInputFormats))
                        return supportedOutputFormats;
                return new Format[0];
    }

    public Format setInputFormat(Format format) {
                if (!matches(format, supportedInputFormats))
                        return null;
                inputFormat = format;
                return format;
    }
```

```
    public Format setOutputFormat(Format format) {
                if (!matches(format, supportedOutputFormats))
                        return null;
                System.out.println(format.toString());
                outputFormat = (RGBFormat) format;
                return format;
    }

    public String getName() {
        return PLUGIN_NAME;
    }

    public void open() {}
    public void close() {}
    public void reset() {}

    public Object getControl(String controlType) {
        System.out.println(controlType);
            return null;
    }

    public Object[] getControls() {
        return null;
    }

   public synchronized int process(Buffer inBuffer, Buffer outBuffer) {

      byte[] inData = (byte[]) inBuffer.getData();
      byte[] outData = new byte[320*240*3];
      outBuffer.setLength(320*240*3);
      outBuffer.setData(outData);

      //下部に少し表示
      System.arraycopy(inData,0,outData,0,320*24*3);
      return BUFFER_PROCESSED_OK;
   }
}
```

以下２つのプログラムは今回新たに定義したフォーマットを定義している。

```
// MYFormat.java
import javax.media.format.*;
```

```
public class MYFormat extends VideoFormat{

        static String DESCRIPTION = "my/raw";

        public MYFormat(String s){
                super(s);
        }

        public MYFormat(){
                super(DESCRIPTION);
        }
}

// MY_RTPFormat.java
import javax.media.format.*;

public class MY_RTPFormat extends VideoFormat{

        static String DESCRIPTION = "my/rtp";

        public MY_RTPFormat(String s){
                super(s);
        }

        public MY_RTPFormat(){
                super(DESCRIPTION);
        }
}
```

＜プログラムの使い方＞
必要なプログラム
ys.java,　MyEncoder.java,　MyPacketizer.java,
Mys.java,　MyDecoder.java,　MyDepacketizer.java
MYFormat.java,　MY_RTPFormat.java, StateHelper.java

コンパイル
送信側 javac Mys.java
受信側 javac Myr.java

実行
送信側 java Mys vfw://0
受信側 java myr

注意：受信スタート後１分以内に送信スタートさせること

終わりに
JMF を筆者の興味にまかせて一通り見てきた。これから、JMF に取り組む方のお役に立てば幸いである。

参考にしたサイト
国内
http://www.sk.cs.chubu.ac.jp/kenkyu/eizo/00/naraki/Kankyo/jmf/jmfin.html
http://cappuccino.jp/keisuken/java/jmf.html
http://www.be.wakwak.com/~bsd/java/jmf/movie.html
http://www.wakhok.ac.jp/~tatsuo/mmp2003/jmf/jisshuu.html
http://www.asahi-net.or.jp/~dp8h-izn/jmf.html
海外（sun 社以外）
http://bart.sm.luth.se/~mathed-8/mDec2/