CS 5143 NETWORK DESIGN AND MANAGEMENT

Class Project

Application-Centric Real Time Video Synthesis and Streaming Over Internet Using Java Media Framework (JMF)

Submitted to:

Dr. Anindya Das

Submitted by:

Jianting Zhang
ShaoJun Wang

Submitted date:

05/10/2001
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 1</td>
<td>Overall System Architecture</td>
<td>6</td>
</tr>
<tr>
<td>Fig. 2</td>
<td>Dynamic Video Synthesis From Different Types of Multimedia Data sources</td>
<td>6</td>
</tr>
<tr>
<td>Fig. 3</td>
<td>JMF Package</td>
<td>7</td>
</tr>
<tr>
<td>Fig. 4</td>
<td>High-level JMF Architecture</td>
<td>8</td>
</tr>
<tr>
<td>Fig. 5</td>
<td>JMF Data Model</td>
<td>9</td>
</tr>
<tr>
<td>Fig. 6</td>
<td>JMF State Transition Diagram</td>
<td>10</td>
</tr>
<tr>
<td>Fig. 7</td>
<td>Getting Drive Directions From Expedia.com</td>
<td>10</td>
</tr>
<tr>
<td>Fig. 8</td>
<td>Text-based Driving Directions from Expedia.com</td>
<td>11</td>
</tr>
<tr>
<td>Fig. 9</td>
<td>Turn-By-Turn Maps of Intersections</td>
<td>11</td>
</tr>
<tr>
<td>Fig. 10</td>
<td>Video Synthesis Process from still images</td>
<td>12</td>
</tr>
<tr>
<td>Fig. 11</td>
<td>Structure of customer data source</td>
<td>13</td>
</tr>
<tr>
<td>Fig. 12</td>
<td>Structure of customer data stream</td>
<td>14</td>
</tr>
</tbody>
</table>
Introduction

Video Streaming over network becomes more and more popular in recent years, such as Internet TV, Video on demand (VOD), etc. With the wide spread of Internet and fast development of wide bandwidth network infrastructure, we can foresee that multimedia streaming technology will be one of the major technologies in the Internet application in the near future. A lot of studies on multimedia streaming in general and video streaming in specific have been performed. However, most of the studies make the assumption that video files are pre-generated in file format. However, in real practices, the demanded videos are more likely need to be dynamically generated in different scenarios. This problem can be further described as having a set of still images, video clips, audio clips and related text information (with or without database support), how to generate the dynamic video based on the given parameters and streaming it over internet to end user. In this study, we try to take travel guide as an example and build a prototype as the basis for our further research.

Java technologies have been very popular in the past years and many extension packages have been added to the language core which provides great convenience for application developer. Java Media Framework (JMF), together with other Java media packages, such as Java 2D/3D, Java advanced imagining (JAI), Java Sound and Java Speech, is one of such good packages that specialized on multimedia related development. JMF provides both excellent architecture and powerful API for developing real time media streaming applications. In this study, we use JMF to synthesis pre-stored still images into videos and steaming it over Internet for travel guidance.
The prototype includes two parts, the first part is to dynamically generate video based on user’s demands and the second part is to stream the video and send it back to user. For the first part, we first send requests to Expedia.com using a place in Norman as the source and Hollywood as the destination to retrieve travel directions. We then save the turn-by-turn road intersection maps to our local machine. We write a Java program using JMF package to generate a video and put it in our WWW directory. Then users can retrieve the video by accessing a standard HTML page which has an embedded Java Applet to playback the video generated in user’s browser.

The rest of this report is arranged as follows: Section 2 presents the overall architecture for video synthesis and streaming over Internet. Section 3 gives an overview of Java Media Framework. Section 4 deals with data collection and implementation issues. Finally, Section 5 presents results and discussion and Section 6 gives future work directions.

**Architecture**

The overall system architecture can be shown as Fig. 1. The client sends request to the server through popular Graphic User Interface (GUI), HTML for example. The server receives requests from client and responses by synthesizing dynamic video and sending it back to client over Internet. The process usually involves the following issues:

The server parses the parameters from client. Based on the retrieved parameters and application semantics associated with the parameters which are pre-stored in the database on the server side, the server knows where to get raw materials for video synthesis. They can multimedia files stored in local machine and managed by file management system (FMS), or in local or remote multimedia databases, such as Oracle or Informix, or just from other websites which are publicly available.
The server must have the capability to deal with various multimedia types, such as images (JPEG, GIF, etc.), audios (wav file, mpeg audio, etc.), videos (real media, quicktime, etc.). Operations usually include clipping and segmentation, scaling and rotating, composting with specific order, etc. For the texts and numbers, the system must be able to render them into pixel-based format (image or video). Finally the system must be able to synchronize audios and moving pictures and synthesize them into demanded videos. Fig 2. Gives more details in video synthesis.

Fig. 1 Overall System Architecture

Fig. 2 Dynamic Video Synthesis From Different Types of Multimedia Data sources
**JMF Overview**

Our prototype system is built on top of Java Media Framework (JMF). It is a large and versatile API used to process time-based media. One of the key unique features JMF has is multimedia streaming over Internet. The organization of the package is briefly shown as Fig 3.

```
javax.media:
javax.media.bean.playerbean
javax.media.control
javax.media.datasink
javax.media.format
javax.media.protocol
javax.media.renderer
javax.media.rtp
javax.media.rtp.event
javax.media.rtp.rtcp
javax.media.util
```

**Fig 3 JMF Package**

The basic JMF classes, such as time control classes, player, processor and manager, basic data sink and data caching classes are included directly in javax.media sub-package. Other sub-packages each includes classes for a specific issue, such as java bean, control, data sink, protocol, render and utility. Note that RTP related classes are included in javax.media.rtp sub-package and its two sub-sub-package. By supporting RTP, JMF is able to conduct real time multimedia capturing and dissemination. Our work on real time video streaming currently does not use RTP and thus does not support real time multimedia capturing and dissemination although we plan to do so in the future.

The JMF layered architecture can be shown as Fig 4. The basic components are at the very bottom level which are multiplxers and de multiplxers, Codecs for encoding and decoding, renders for rendering data and effects for changing the way of rendering. These components are wrapped into plug-in APIs. They are also further wrapped into JMF presentation and processing API and on top of which various Java Applications, Applets and Beans can be built.
JMF data model can be shown as Fig 5. JMF media players usually use DataSources to manage the transfer of media-content. A DataSource encapsulates both the location of media and the protocol and software used to deliver the media. A DataSource manages a set of SourceStream objects. A DataSource can be either Pull or Push, Stand or Buffered. A standard data source uses a byte array as the unit of transfer. A buffer data source uses a Buffer object as its unit of transfer. A special Pull Data Source is URL based Data Source. Similarly, a SourceStream can also be either either Pull or Push, Stand or Buffered.

When the media is played by a player, the player have several states to go through and switch back and forth. The state diagram of a player is shown as fig 6. The possible states can be briefly described as follows:

- A Player in the Unrealized state has been instantiated, but does not yet know anything about its media. When a media Player is first created, it is Unrealized.
- A Realizing Player is in the process of determining its resource requirements. A Realizing Player often downloads assets over the network.
- A Realized Player knows what resources it needs and information about the type of media it is to present. A can provide visual components and controls.
- A **Prefetching** Player is preparing to present its media. During this phase, the Player preloads its media data, obtains exclusive-use resources, and does whatever else it needs to do to prepare itself to play. **Prefetching** might have to recur if a Player object's media presentation is repositioned, or if a change in the Player object's rate requires that additional buffers be acquired or alternate processing take place.

- When a Player finishes **Prefetching**, it moves into the **Prefetched** state. A **Prefetched** Player is ready to be started.

- A **Started** Player object's time-base time and media time are mapped and its clock is running, though the Player might be waiting for a particular time to begin presenting its media data.

---

![Fig. 5 JMF Data Model](image-url)
When the player is embedded into a web browser and get started, it can play back the dynamically generated video. This is exactly how we use of JMF in this project.

**Data Collection**
The application we take for example is a travel guide from expedia.com. We take a place in Norman as the source and Hollywood as the destination to get the driving directions. The process is shown in Fig 7.
Expedia.com not only gives the text-based driving directions as shown in Fig 8 but also gives the turn-by-turn maps of intersections as shown in Fig 9.

For long and complex travel route, there could be tens of intersections, scrolling up and down of the browser to look into all the maps is both time-consuming and very difficult. Since human are more capable in interpreting moving objects, transforming these still images into a video might help user to understand the travel directions better. This is our very first motivation of the prototype system. We save the still images to our local machine and generate a quicktime video.
movie file by composting these images according to their sequence using a Java program which will be described in detail below.

**Implementation**

In this prototype system, only very limited functionalities are implemented as compared to the proposed architecture. One of the most important implemented feature is to synthesis still images into video.

**General Process**

The synthesis process based on JMF is shown as Fig 10.

![Diagram of video synthesis process](image)

Fig. 10 Video Synthesis Process from still images

The file names of the series images are the only input parameters needed and the output video file name is the only output parameters needed. The program first creates a data source taken the image file names as parameter and then call manager to create a processor. After the processor is created and control listener is added to the processor, the program call CONFIG function of the processor and wait the processor to be configured. Similarly, the program call REALIZE function and wait the processor to be realized. After the processor is realized, the
program create a data sink associated with the processor which takes the output media locator as parameter. The program is then call the start functions of both the processor and the data sink. The all the input images files are read, the program will generate a video file to the specific location.

Customer Data Source

The structure of customer data source is shown in Fig 11.

```java
ImageDataSource extends PullBufferDataSource
{
    ImageSourceStream streams[];//Created During Initialization
    public PullBufferStream[] getStreams() { return streams;}
    public setLocator(MediaLocator source);
    public MediaLocator getLocator()
    public String getContentType();
    public void connect();
    public void disconnect();
    public void start();
    public void stop();
    public Time getDuration();
    public Object[] getControls()
}
```

Fig. 11 Structure of customer data source

The most important function in a customer data source is “getStreams” which returns the source stream(s) which are the private variables of the class and created in the constructor function. The rest abstract functions can use defaults.

Class Data Stream

The structure of customer data stream is shown in Fig 12. The most important function in a customer data stream is “read”. The callback function tells the system how to read a image file into a buff to be used in creating a video frame. Also there is a very important variable “format” tells the system which format to use in creating the video. We can specify several necessary parameters, such as width, height and frame rate.
class ImageSourceStream implements PullBufferStream
{
    VideoFormat format=new VideoFormat(VideoFormat.JPEG,new Dimension(width, height),Format.NOTSPECIFIED,Format.byteArray,(float)frameRate);
    boolean ended;//initially false, change to false after finishing reading data
    public void read(Buffer buf);
    public Object[] getControls();
    public long getContentLength();
    public ContentDescriptor getContentDescriptor();
    public Format getFormat();
}

Fig. 12 Structure of customer data stream

The applet we used to play the synthesized video comes with JMF as an example. It is included in the appendix.

**Result and Discussion**

The result can be viewed online at [http://ada.cs.ou.edu/pubs/project/ToHollyWood.htm](http://ada.cs.ou.edu/pubs/project/ToHollyWood.htm).

The quick time movie is generated by composting the still images of turn-by-turn intersection maps from expedia.com. When the HTML page is first uploaded we can see the initialization process of the applet. The GUI control of the applet appears first. When the video data is streamed, the applet begins to playback the video. We can pause, backward and forward the video or repositioning to any frame. We can get the same amount information in a much smaller presentation space by composting still images into a video. The video sequence of frames might better help user to understand the travel directions.

**Conclusion and Future Work**

Through this study, we found that JMF has a good architecture and a set of powerful API for video streaming over network. In our future study, we plan to use further explore the RTP and
RTCP classes in the package and enable video streaming over RTP and RTCP protocols to make it more like “real time”. We also want to build semantic databases for different applications to enable our system to generate video more relevant to user demands intelligently. More GUI to accept user parameters and interact with user should also be one of the priorities.
Appendix

Java Video Synthesis Program

```java
import java.io.*;
import java.awt.*;
import java.awt.image.*;
import java.awt.geom.*;
import java.util.*;
import java.awt.Dimension;
import javax.media.*;
import javax.media.control.*;
import javax.media.protocol.*;
import javax.media.datasink.*;
import javax.media.util.*;
import javax.media.format.VideoFormat;
import java.awt.image.renderable.*;
import java.awt.image.*;
import java.awt.Image;
import javax.media.jai.*;
import com.sun.media.jai.codec.*;
import java.awt.event.*;
import com.sun.image.codec.jpeg.*;
import com.sun.media.jai.codec.SeekableStream;
import com.sun.media.jai.codec.FileSeekableStream;

public class RTVideo implements ControllerListener, DataSinkListener {

    public boolean doIt(int width, int height, int frameRate, Vector inFiles, MediaLocator outML) {
        ImageDataSource ids = new ImageDataSource(width, height, frameRate, inFiles);

        Processor p;

        try {
            System.out.println("- create processor for the image datasource ... ");
            p = Manager.createProcessor(ids);
        } catch (Exception e) {
            System.out.println("Cannot create a processor from the data source.");
            return false;
        }

        p.addControllerListener(this);

        // call configure and wait to be configured
        p.configure();
        if (!waitForState(p, p.Configured)) {
            System.out.println("Failed to configure the processor.");
            return false;
        }

        // Set the output content descriptor to QuickTime.
        p.setContentDescriptor(new ContentDescriptor(FileTypeDescriptor.QUICKTIME));

        // Query for the processor for supported formats, then set it on the processor.
        TrackControl tcs[] = p.getTrackControls();
        Format f[] = tcs[0].getSupportedFormats();
        if (f == null || f.length <= 0) {
            System.out.println("The mux does not support the input format: "+ tcs[0].getFormat());
        }
```

return false;
}
tcs[0].setFormat(f[0]);

System.out.println("Setting the track format to: " + f[0]);

// realization
p.realize();
if (!waitForState(p, p.Realized)) {
    System.out.println("Failed to realize the processor.");
    return false;
}

// Create a DataSink.
DataSink dsink;
if ((dsink = createDataSink(p, outML)) == null) {
    System.out.println("Failed to create a DataSink for the given output MediaLocator: " + outML);
    return false;
}

dsink.addDataSinkListener(this);
fileDone = false;
System.out.println("Start processing...");

// Start the actual transcoding.
try {
    p.start();
    dsink.start();
} catch (IOException e) {
    System.out.println("IO error during processing");
    return false;
}

// Wait for EndOfStream event.
waitForFileDone();

// Cleanup.
try {
    dsink.close();
} catch (Exception e) {
}
p.removeControllerListener(this);

System.out.println("...done processing.");

return true;
}

DataSink createDataSink(Processor p, MediaLocator outML) {

    DataSource ds;

    if ((ds = p.getDataOutput()) == null) {
        System.out.println("Something is really wrong: the processor does not have an output DataSource");
        return null;
    }
DataSink dsink;

try {
    System.out.println("- create DataSink for: "+ outML);
    dsink = Manager.createDataSink(ds, outML);
    dsink.open();
} catch (Exception e) {
    System.out.println("Cannot create the DataSink: "+ e);
    return null;
}

return dsink;

Object waitSync = new Object();
boolean stateTransitionOK = true;

// waiting for a specific state
boolean waitForState(Processor p, int state) {
    synchronized (waitSync) {
        try {
            while (p.getState() < state & stateTransitionOK)
                waitSync.wait();
        } catch (Exception e) {}
    }
    return stateTransitionOK;
}

/**
 * Controller Listener.
 */
public void controllerUpdate(ControllerEvent evt) {

    if (evt instanceof ConfigureCompleteEvent ||
        evt instanceof RealizeCompleteEvent ||
        evt instanceof PrefetchCompleteEvent) {
        synchronized (waitSync) {
            stateTransitionOK = true;
            waitSync.notifyAll();
        }
    } else if (evt instanceof ResourceUnavailableEvent) {
        synchronized (waitSync) {
            stateTransitionOK = false;
            waitSync.notifyAll();
        }
    } else if (evt instanceof EndOfMediaEvent) {
        evt.getSourceController().stop();
        evt.getSourceController().close();
    }
}

Object waitFileSync = new Object();
boolean fileDone = false;
boolean fileSuccess = true;

// Block until file writing is done.
boolean waitForFileDone() {
    synchronized (waitFileSync) {
        try {
            while (!fileDone)
                waitFileSync.wait();
        } catch (Exception e) {}
    }
    return fileSuccess;
}

// Event handler for the file writer.
public void dataSinkUpdate(DataSinkEvent evt) {
    if (evt instanceof EndOfStreamEvent) {
        synchronized (waitFileSync) {
            fileDone = true;
            waitFileSync.notifyAll();
        }
    } else if (evt instanceof DataSinkErrorEvent) {
        synchronized (waitFileSync) {
            fileDone = true;
            fileSuccess = false;
            waitFileSync.notifyAll();
        }
    }
}

public static void main(String args[]) {
    int width = 180, height = 140, frameRate = 1;
    Vector inputFiles = new Vector();
    String outputURL = new String("ToHollyWood.mov");
    for(int i=1;i<=17;i++) inputFiles.add(new String("d:\jzhang\courses\cs5143\project\turn"+i+".gif"));

    // Check for output file extension.
    if (!outputURL.endsWith(".mov") && !outputURL.endsWith(".MOV")) {
        System.out.println("The output file extension should end with a .mov extension");
        prUsage();
    }

    if (width < 0 || height < 0) {
        System.out.println("Please specify the correct image size.");
        prUsage();
    }

    // Check the frame rate.
    if (frameRate < 1)
        frameRate = 1;

    // Generate the output media locators.
    MediaLocator oml;
    if ((oml = createMediaLocator(outputURL)) == null) {
        System.out.println("Cannot build media locator from: "+ outputURL);
        System.exit(0);
    }
}
RTVideo imageToMovie = new RTVideo();
imageToMovie.doIt(width, height, frameRate, inputFiles, oml);

System.exit(0);
}

static void prUsage() {
    System.out.println("Usage: java RTVideo -w <width> -h <height> -f <frame rate> -o <output URL>
<input JPEG file 1> <input JPEG file 2> ...");
    System.exit(-1);
}

//Create a media locator from the given string
static MediaLocator createMediaLocator(String url) {

    MediaLocator ml;
    if (url.indexOf(":" > 0 && (ml = new MediaLocator(url)) != null)
        return ml;

    if (url.startsWith(File.separator)) {
        if ((ml = new MediaLocator("file:" + url)) != null)
            return ml;
    } else {
        String file = "file:" + System.getProperty("user.dir") + File.separator + url;
        if ((ml = new MediaLocator(file)) != null)
            return ml;
    }

    return null;
}

//customer data source
class ImageDataSource extends PullBufferDataSource {

    ImageSourceStream streams[];

    ImageDataSource(int width, int height, int frameRate, Vector images) {
        streams = new ImageSourceStream[1];
        streams[0] = new ImageSourceStream(width, height, frameRate, images);
    }

    public void setLocator(MediaLocator source) {
    }

    public MediaLocator getLocator() {
        return null;
    }

    public String getContentType() {
        return ContentDescriptor.RAW;
    }

    public void connect() {
    }

    public void disconnect() {
}
public void start() {
}

public void stop() {
}

public PullBufferStream[] getStreams() {
    return streams;
}

public Time getDuration() {
    return DURATION_UNKNOWN;
}

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

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

//customer data stream
class ImageSourceStream implements PullBufferStream {

    Vector images;
    int width, height, maxlen;
    VideoFormat format;
    ImageToBuffer itb;

    int nextImage = 0;
    boolean ended = false;

    public ImageSourceStream(int width, int height, int frameRate, Vector images) {
        this.width = width;
        this.height = height;
        if(width>height) maxlen=width;
        else maxlen=height;
        this.images = images;
        itb=new ImageToBuffer();

        format = new VideoFormat(VideoFormat.JPEG,
                                new Dimension(width, height),
                                Format.NOT_SPECIFIED,
                                Format.byteArray,
                                (float)frameRate);
    }

    //if data is read from file, we do not need to block it
    public boolean willReadBlock() {
        return false;
    }

    //call back function to read data from a file into a buffer
    public void read(Buffer buf) throws IOException {

    }
}
// Check if we've finished all the frames.
if (nextImage >= images.size()) {
    // We are done. Set EndOfMedia.
    System.out.println("Done reading all images.");
    buf.setEOM(true);
    buf.setOffset(0);
    buf.setLength(0);
    ended = true;
    return;
}

String imageFile = (String)images.elementAt(nextImage);
nextImage++;
String tempStr="d:\jzhang\courses\cs5143\project\temp.jpg";
System.out.println(" - reading image file: " + imageFile);
if(transferImage(imageFile,tempStr)>0)
{
    RandomAccessFile  raFile = new RandomAccessFile(tempStr, "r");
    byte data[] = null;
    if (buf.getData() instanceof byte[]) data = (byte[])buf.getData();
    if (data == null || data.length < raFile.length())
    {
        data = new byte[(int)raFile.length()];
        buf.setData(data);
    }
    raFile.readFully(data, 0, (int)raFile.length());
    System.out.println("   read " + raFile.length() + " bytes.");
    buf.setOffset(0);
    buf.setLength((int)raFile.length());
    buf.setFormat(format);
    buf.setFlags(buf.getFlags() | buf.FLAG_KEY_FRAME);
    raFile.close();
}

public int transferImage(String inStr,String outStr)
{
    RenderedImage img;
    try {
        SeekableStream stream = new FileSeekableStream(inStr);
        String[] names = ImageCodec.getDecoderNames(stream);
        ImageDecoder dec = ImageCodec.createImageDecoder(names[0], stream, null);
        img=dec.decodeAsRenderedImage();
        FileOutputStream out = new FileOutputStream(outStr);
        ImageEncoder encoder = ImageCodec.createImageEncoder("JPEG", out, null);
        encoder.encode(img);
        out.close();
        return(1);
    } catch (Exception e) {
        e.printStackTrace();
        return(-1);
    }
}
public Format getFormat() {
    return format;
}

public ContentDescriptor getContentDescriptor() {
    return new ContentDescriptor(ContentDescriptor.RAW);
}

public long getContentLength() {
    return 0;
}

public boolean eofStream() {
    return ended;
}

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

public Object getControl(String type) {
    return null;
}
}
Java Applet for playback video (from JMF package)

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.lang.String;
import java.net.URL;
import java.net.MalformedURLException;
import java.io.IOException;
import java.util.Properties;
import javax.media.*;
//import com.sun.media.util.JMFSecurity;

/**
 * This is a Java Applet that demonstrates how to create a simple
 * media player with a media event listener. It will play the
 * media clip right away and continuously loop.
 *
 * <!-- Sample HTML
 * <applet code=SimplePlayerApplet width=320 height=300>
 * <param name=file value="sun.avi">
 * </applet>
 * -->
 *
 */

public class SimplePlayerApplet extends Applet implements ControllerListener {

// media Player
Player player = null;
// component in which video is playing
Component visualComponent = null;
// controls gain, position, start, stop
Component controlComponent = null;
// displays progress during download
Component progressBar = null;
boolean firstTime = true;
long CachingSize = 0L;
Panel panel = null;
int controlPanelHeight = 0;
int videoWidth = 0;
int videoHeight = 0;

/**
 * Read the applet file parameter and create the media
 * player.
 */
public void init() {
  //$ System.out.println("Applet.init() is called");
  setLayout(null);
  setBackground(Color.white);
  panel = new Panel();
  panel.setLayout( null );
  add(panel);
  panel.setBounds(0, 0, 320, 240);

  // input file name from html param
  String mediaFile = null;
  // URL for our media file
  MediaLocator mrl = null;
  URL url = null;

  // Get the media filename info.
  // The applet tag should contain the path to the
  // source media file, relative to the html page.
  if ((mediaFile = getParameter("FILE")) == null)
    Fatal("Invalid media file parameter");

  try {
    url = new URL(getDocumentBase(), mediaFile);
    mediaFile = url.toExternalForm();
  } catch (MalformedURLException mue) {
  }

  try {
    // Create a media locator from the file name
    if ((mrl = new MediaLocator(mediaFile)) == null)
      Fatal("Can't build URL for " + mediaFile);

    /*
    try {
      JMFSecurity.enablePrivilege.invoke(JMFSecurity.privilegeManager,
      JMFSecurity.writePropArgs);
      JMFSecurity.enablePrivilege.invoke(JMFSecurity.privilegeManager,
      JMFSecurity.readPropArgs);
      JMFSecurity.enablePrivilege.invoke(JMFSecurity.privilegeManager,
      JMFSecurity.connectArgs);
    */
catch (Exception e) {} */

// Create an instance of a player for this media
try {
    player = Manager.createPlayer(mrl);
} catch (NoPlayerException e) {
    System.out.println(e);
    Fatal("Could not create player for " + mrl);
}

// Add ourselves as a listener for a player's events
player.addControllerListener(this);

} catch (MalformedURLException e) {
    Fatal("Invalid media file URL!");
} catch (IOException e) {
    Fatal("IO exception creating player for " + mrl);
}

// This applet assumes that its start() calls
// player.start(). This causes the player to become
// realized. Once realized, the applet will get
// the visual and control panel components and add
// them to the Applet. These components are not added
// during init() because they are long operations that
// would make us appear unresponsive to the user.
}

/**
 * Start media file playback. This function is called the
 * first time that the Applet runs and every
 * time the user re-enters the page.
 */

public void start() {
    // System.out.println("Applet.start() is called");
    // Call start() to prefetch and start the player.
    if (player != null)
        player.start();
}

/**
 * Stop media file playback and release resource before
 * leaving the page.
 */

public void stop() {
    // System.out.println("Applet.stop() is called");
    if (player != null) {
        player.stop();
        player.deallocate();
    }
}

public void destroy() {
    // System.out.println("Applet.destroy() is called");
    player.close();
}
/**
 * This controllerUpdate function must be defined in order to
 * implement a ControllerListener interface. This
 * function will be called whenever there is a media event
 * /
 * public synchronized void controllerUpdate(ControllerEvent event) {
   // If we're getting messages from a dead player,
   // just leave
   if (player == null)
     return;

   // When the player is Realized, get the visual
   // and control components and add them to the Applet
   if (event instanceof RealizeCompleteEvent) {
     if (progressBar != null) {
       panel.remove(progressBar);
       progressBar = null;
     }

     int width = 320;
     int height = 0;
     if (controlComponent == null)
       if (( controlComponent =
             player.getControlPanelComponent()) != null) {
         controlPanelHeight = controlComponent.getPreferredSize().height;
         panel.add(controlComponent);
         height += controlPanelHeight;
       }

     if (visualComponent == null)
       if (( visualComponent =
             player.getVisualComponent())!= null) {
         panel.add(visualComponent);
         Dimension videoSize = visualComponent.getPreferredSize();
         videoWidth = videoSize.width;
         videoHeight = videoSize.height;
         width = videoWidth;
         height += videoHeight;
         visualComponent.setBounds( 0, 0, videoWidth, videoHeight);
       }

     panel.setBounds(0, 0, width, height);
     if (controlComponent != null) {
       controlComponent.setBounds(0, videoHeight,
                                 width, controlPanelHeight);
       controlComponent.invalidate();
     }
   } else if (event instanceof CachingControlEvent) {
     if (player.getState() > Controller.Realizing)
       return;

     // Put a progress bar up when downloading starts,
     // take it down when downloading ends.
     CachingControlEvent e = (CachingControlEvent) event;
     CachingControl cc = e.getCachingControl();

     // Add the bar if not already there ...
     if (progressBar == null) {

if ((progressBar = cc.getControlComponent()) != null) {
    panel.add(progressBar);
    panel.setSize(progressBar.getPreferredSize());
    validate();
}

} else if (event instanceof EndOfMediaEvent) {
    // We've reached the end of the media; rewind and
    // start over
    player.setMediaTime(new Time(0));
    player.start();
} else if (event instanceof ControllerErrorEvent) {
    // Tell TypicalPlayerApplet.start() to call it a day
    player = null;
    Fatal(((ControllerErrorEvent)event).getMessage());
} else if (event instanceof ControllerClosedEvent) {
    panel.removeAll();
    }

void Fatal (String s) {
    // Applications will make various choices about what
    // to do here. We print a message
    System.err.println("FATAL ERROR: "+ s);
    throw new Error(s); // Invoke the uncaught exception
        // handler System.exit() is another
            // choice.
        }
    }
HTML page with Video Playback Applet

<center>
<applet code=SimplePlayerApplet.class width=180 height=160>
<param name=file value="ToHollyWood.mov">
</applet>
</center>
References


J.D. Gibson (editor), Multimedia Communications: Directions and Innovations, Academic Press, 2001

[HREF 1] Budi Kurniawan, Program multimedia with JMF, Part 1,
