제 4 강좌 : DataSource를 이용한 JMF 기능 확장하기

지난 강좌를 통하여 독자들은 JMF에서 새롭게 제공하는 비디오 및 오디오 데이터에 대한 라이브 데이터의 획득기능, 획득된 데이터를 가공하여 특정한 데이터 포맷으로의 저장기능, 포맷 변환과 더불어 비디오 캡쳐시의 캡쳐 정보를 획득하는 방법등에 관하여 살펴보았으며, 오디오와 비디오 데이터를 동시에 캡쳐하는 방법을 ProcessorModel로의 구현과 각각의 독립된 Processor를 두어 구현하는 방법에 관해서도 알아보았다. 지난 강좌를 통하여 실제 오디오 및 비디오 데이터의 캡쳐작업과 이를 처리하는 모든 부분이 소개되었으나, 만약 JMF에서 표준적으로 제공하지 않는 캡쳐장치나 데이터를 이용하고자 한다면 어떻게 구현해야 할까. 이번 강좌에서는 지난 강좌에서 충분하게 다루지 못한 오디오 비디오 데이터 캡쳐 부분에서 캡쳐작업을 위한 버퍼크기와 Renderer 버퍼크기를 직접 실험적으로 세팅하는 방법을 알아보고 비표준적인 데이터소스를 생성하고 이를 등록하여 이용하는 방법을 구현하며, JMStudio에서의 RTP 통신이용 방법에 관하여 알아본다. 본 강좌에 대한 문의 사항이나 관련된 프로그램은 아래의 홈페이지를 참조하기 바란다.

이재훈 전임연구원
삼성테크윈 정밀기기연구소 (Samsung Techwin.)
kingseft@samsung.co.kr
http://myhome.naver.com/kingseft

Capture Buffer 크기와 Renderer 크기의 조절 방법

이번 강좌에서의 첫부분은 오디오 캡쳐시의 캡쳐과정에서 필요한 캡쳐 버퍼의 크기조절과 이렇게 획득된 데이터의 Renderer 버퍼크기를 조정하는 방법을 구현해보고자 한다. 실제로 캡쳐 작업에서 이용하는 캡쳐 버퍼의 크기가 작다면 latency를 감소시키지만, 좀더 심한 오디오의 절음 현상이 발생되기 때문에 개발자들이 서로 다른 시스템들에서 버퍼를 구현한다면 몇몇 실험적인 방법으로 알맞은 버퍼의 크기를 찾아야한다. 오디오에서는 AudioBufferControl을 이용하여 앞에서 언급한 방법을 구현할 수 있는데, DataSource 또는 Player에서 각각 AudioBufferControl을 얻어낼 수 있다. DataSource에서 얻어진 AudioBufferControl은 캡쳐되는 버퍼의 크기를 조절할수 있으며, Player에서 얻은 AudioBufferControl을 이용하여 Renderer에 이용되는 버퍼의 크기를 조절할수있다. [그림 1]은 구현된 프로그램의 실행 모습이다.

[그림 1] Audio Buffer Control 실행화면

실제적인 프로그램 구현부분을 살펴보도록 하자. 먼저 main()부에서는 디폴트 버퍼 사이즈를 밀리 세컨드단위로 정의해두었다.

[리스트 1] 캡쳐 및 Render 버퍼크기의 정의

```
static final int DEF_CAPTURE_SIZE = 62;
static final int DEF_RENDER_SIZE = 400;
```

다음은 main()의 명령형 인자로서 캡쳐 버퍼크기와 Renderer 버퍼크기를 입력받고 이를 알맞은 값으로 변환하는 부분이다. 아래와 같은 판독 조건을 만족하기위하여 실제 프로그램 실행은 java AudioBufferControl - c 120 - r 300 javasound://44100 과 같이 입력한다.

[리스트 2] 명령행 인자 판독 부분

```
int i = 0;
while (i < args.length) {
  if (args[i].equals("-c")) {
        i++;
        if (i >= args.length) {
          prUsage();
          System.exit(0);
        }
        try {
          captureBufSize = Integer.parseInt(args[i]);
        } catch (NumberFormatException e) {
          prUsage();
          System.exit(0);
        }
    } else if (args[i].equals("-r")) {
        i++;
        if (i >= args.length) {
          prUsage();
          System.exit(0);
        }
        try {
          renderBufSize = Integer.parseInt(args[i]);
```

```
        } catch (NumberFormatException e) {
          prUsage();
          System.exit(0);
        }
    } else {
        mlStr = args[i];
    }
    i++;
  }
  if (mlStr == null) {
    prUsage();
    System.exit(0);
  }
```

위 프로그램 구현과 같이 원하는 버퍼 사이즈를 입력받은 후에는 명령형 인자로부터 오디오 캡쳐 장치가 실제적으로 존재하는 위치정보와 프로토콜 정보를 얻기위하여 MediaLocator를 얻어오며, 이러한 MediaLocator로부터 Manager 클래스의 createDataSource를 이용하여 플레이가 가능한 DataSource를 생성할 수 있다.

[리스트 3] 데이터 소스의 획득 부분

```
MediaLocator ml;
if ((ml = new MediaLocator(mlStr)) == null) {
  System.err.println("Cannot build media locator from: " + mlStr);
  prUsage();
  System.exit(0);
}
DataSource ds = null;
try {
    ds = Manager.createDataSource(ml);
} catch (Exception e) {
    System.err.println("Cannot create DataSource from: " + ml);
    System.exit(0);
}
```

위에서와 같이 데이터소스를 획득한후 우리가 구현하고자 하는 AudioBufferControl 를 생성하고 구현된 클래스의 eopn() 메쏘드를 이용하여 데이터소스, 오디오 캡쳐 버퍼 크기 및

Renderer 버퍼크기를 인자로 넘겨준다.

[리스트 4] AudioBufferControl 클래스의 생성과 메쏘드 호출

```
AudioBufferControl abc = new AudioBufferControl();
if (!abc.open(ds, captureBufSize, renderBufSize))
    System.exit(0);
}
```

이제는 AudioBufferControl 클래스를 직접 구현하는 단계이다. 이 클래스는 Frame을 상속받고, 기본적인 Player, Processor의 이벤트 처리를 위하여 ControllerListener()를 구현한다. 내부적으로 구현된 메쏘드는 open()이며, 이 open()메쏘드에서 관련된 모든 일을 처리하도록 구현한다. 먼저, 획득된 데이터 소스가 버퍼 컨트롤을 제공해주는지의 여부를 판별해야 한다. 버퍼 컨트롤은 데이터소스의 getControl() 메쏘드를 통해 얻어질 수 있으며, 이때의 반환값은 Control 형이된다. 반환된 Control 형의 버퍼 컨트롤이 Null 값이 아니라면, 이 컨트롤의 setBufferLength()를 통하여 버퍼의 크기를 밀리세컨드 단위로 지정할수있다. [리스트 5]를 살펴보자. 주의할 점은 본 데이터 소스를 이용하는 플레이어와 프로세서의 현재 상태 정보이다. 데이터 소스를 통해 얻어진 버퍼 컨트롤의 값 변경을 위해서는 이 데이터 소스를 이용하고자하는 플레이어나 프로세서가 만들어지기 이전에 버퍼 컨트롤값을 변경하고 그 이후에 이 데이터소스를 플레이어나 프로세서에 연결시켜야 한다는 것이다.

[리스트 5] 데이터소스의 버퍼컨트롤 제어

```
Control c = (Control)ds.getControl("javax.media.control.BufferControl");
if (c != null)
    ((BufferControl)c).setBufferLength(captureBufSize);
```

이제는 위에서 세팅된 데이터소스를 통하여 이를 이용할 프로세서를 만드는 부분이다. [리스트 6]에서는 Manager 클래스의 createProcessor() 메쏘드를 이용하여 프로세서를 생성하였고, 이렇게 생성된 프로세서의 각종 이벤트르 검출하기 위하여 addControllderListener() 메쏘드를 이용하여 이벤트 리스너를 등록 시켰다. 이후 프로세서의 configure() 와 realize() 를 호출하고 각 상태로 안전하게 이동하였는지를 확인하는 과정을 거친다. 이 부분에서 또하나의 주의할 점이있다. 앞서 우리는 캡쳐 버퍼크기의 지정과 더불어 Renderer의 크기를 지정하였다. 그렇다면 이 Renderer의 크기를 직접 지정하는 것은 Processor나 Player의 여러 상태중 어떤 상태에 있을 때 가능한것인가? 독자들이 직접 알맞은 프로세서의 상태를 추측해 보기 바란다.

[리스트 6] 변경된 데이터소스를 이용한 프로세서의 생성

```
try {
  p = Manager.createProcessor(ds);
} catch (Exception e) {
  System.err.println("Failed to create a processor from the given DataSource: " + e);
  return false;
}
p.addControllerListener(this);
p.configure();
if (!waitForState(p.Configured)) {
  System.err.println("Failed to configure the processor.");
  return false;
}
p.setContentDescriptor(null);
p.realize();
if (!waitForState(p.Realized)) {
  System.err.println("Failed to realize the processor.");
  return false;
}
```

Renderer 버퍼의 크기 재지정은 프로세서가 Realized된후 Prefetched 되기 직전에 실행을 해주어야 한다. [리스트 6]을 통하여 프로세서가 realized 된 상태를 확인하고, 이후에 Renderer 버퍼크기의 지정, 프로세서의 prefetched 작업을 명시해주면 된다. [리스트 7]을 통해 구현 방법을 확인하여 보자. 프로세서의 getControls()를 통하여 프로세서에 등록된 컨트롤 배열을 획득하였고, 이러한 컨트롤이 BufferControl 인지를 확인하는 과정이 필요하다. 이렇게 얻어진 버퍼 컨트롤중 이 버퍼컨트롤의 소유자가 Renderer 이라면 우리가 세팅하고자하는 Renderer의 버퍼컨트롤임을 확인할수있고 setBufferLength()를 통하여 Renderer의 버퍼크기를 지정한후에 프로세서를 prefetched 상태로 이동을 시킨다.

[리스트 7] 프로세서의 상태변환과 Renderer 버퍼 크기의 지정

```
Control cs[] = p.getControls();
Object owner;
for (int i = 0; i < cs.length; i++) {
  if (cs[i] instanceof Owned && cs[i] instanceof BufferControl) {
    owner = ((Owned)cs[i]).getOwner();
    if (owner instanceof Renderer) {
      ((BufferControl)cs[i]).setBufferLength(renderBufSize);
```

```
    }
  }
}
p.prefetch();
if (!waitForState(p.Prefetched)) {
 System.err.println("Failed to prefetch the processor.");
 return false;
}
```

다음은 데이터 소스로부터 얻어진 VisualComponent와 ControlPanelComponent를 부착시키는 부분과 플레이어나 프로세서의 이벤트 처리 부분이다. 이 부분은 각각 프로세서의 getVisualComponent() 메쏘드와 getControlPanelComponent() 메쏘드를 통하여 획득할 수 있다. 여기 까지의 작업을 마친후 프로세서를 start() 메쏘드를 통하여 구동을 시킨다.

오디오 캡쳐 버퍼 및 Renderer의 크기는 여러분이 실험하는 컴퓨터의 환경에 따라서 그 값의 차이가 많기 때문에 캡쳐버퍼의 크기와 Renderer 버퍼의 크기를 적절히 조절해가면서 두 값의 적절한 절충점을 찾음으로서 알맞은 음질을 구현할 수 있게된다.

[리스트 8] Component의 부착 부분

```
// Display the visual & control component if there's one.
setLayout(new BorderLayout());
Component cc;
Component vc;
if ((vc = p.getVisualComponent()) != null) {
   add("Center", vc);
}
if ((cc = p.getControlPanelComponent()) != null) {
    add("South", cc);
}
p.start();
setVisible(true);


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) {
      p.setMediaTime(new Time(0));
    } else if (evt instanceof SizeChangeEvent) {
    }
}
```

Custom DataSource 를 이용한 Live 오디오/비디오 데이터의 생성과 재생

이번에는 사용자가 직접 정의하는 커스텀 데이터소스를 이용하여 라이브 오디오/비디오 데이터를 생성하고 JMStudio에서 직접 구동시키는 방법을 구현하여 본다. 이 방법은 JMF에서 표준으로 인식하는 프로토콜을 사용하지 못하는 상황에서 사용자 정의의 데이터를 획득하여 그 데이터를 JMF가 인식할 수 있는 포맷으로 변경하는 방법이 필요하다. 이와 같은 커스텀 데이터 소스 획득 방식을 이용하면 JMF에서 직접적으로 인식하지 못하는 오디오 비디오 캡쳐 장치에서 데이터를 획득하여 이를 JMF가 인식가능하도록 할 수 있다. 하나의 예를 들어보자. 독자들에게 압축되지 않은 Raw 형태의 오디오 및 비디오 데이터 파일이 있다면 이는 바로 JMF에서 구동이 불가능하고, 플레이를 위해서는 독자들이 직접 나름대로의 미디어 플레이어를 작성해야하지만, JMF에서 커스텀 데이터 소스의 생성과 등록을 통하여 이러한 파일을 읽어들여 버퍼에 넣고, 버퍼의 내용을 트랜스코딩을 하여 JMF에서 인식가능하도록 할 수도 있을 것이다. 또한 네트워크를 통해 전송되어지는 데이터들을 서버측에서 수신하여 JMF 에서 인식가능하도록 변환하는 과정도 가능하다.

JMF 2.x 버전이후에는 PushBufferDataSource 라는 클래스가 새롭게 추가되었다. 이 클래스는 버퍼 스트림의 일종인 PushBufferStream 형태의 스트림을 포함하며, 이러한 스트림은 기존의 바이트 배열의 데이터 흐름이 아닌 프레임단위의 오디오/비디오 데이터를 생성해낼 수 있기 때문에 버퍼 객체를 통하여 적절한 크기의 오디오 비디오 데이터를

프레임단위로 처리가 가능하게 된다. 그러므로 우리가 작성해야 하는 클래스는 PushBufferDataSource의 하부 클래스와 PushBufferStream을 구현한 클래스이어야 한다.
먼저 데이터 소스의 개념부터 확인하자.
데이터소스는 미디어의 위치와 이용할 프로토콜 그리고 실제 미디어 데이터를 전달하기위해 사용되는 소프트웨어를 캡슐화한 데이터소스를 구현한 객체이다. 이러한 데이터 소스중에서 우리가 구현할 것은 Push DataSource이다. Push DataSource는 서버측에서 데이터의 전송과 통제를 가능하게하는 데이터 소스이다. 이와 같은 Push DataSource 중에서 버퍼를 이용한 데이터소스가 바로 PushBufferDataSource를 형성하게 된다. 이 클래스는 DataSource를 추상화한 클래스로서 PushStream 형태로 데이터를 관리하며 이러한 데이터소스로부터의 스트림은 PushBufferStream이라는 버퍼를 포함하게된다. 이 클래스에서는 getStream() 메쏘드를 포함하고있으며, 이 메쏘드는 데이터소스로부터의 스트림 컬렉션을 얻어오게된다.
또하나 주의할점은 PushBufferDataSource 역시 javax.media.protocol.DataSource 에서 확장된것이므로 DataSource 클래스의 메쏘드들인 connect(), disconnect(), getContentType(), getLocator(), initCheck(), setLocator(), start(), stop() 등을 구현해 주어야한다. 자 이제 실제적은 프로그램을 구현하여 보자.

우리가 구현할 클래스의 이름은 DataSource 로서 PushBufferDataSource 클래스를 확장한다. 또한 확장 클래스이기 때문에 abstract 메쏘드인 getStreams를 구현해야한다.

[리스트 9] PushBufferDataSource 클래스의 확장

```
package jmfsample.media.protocol.live;

import javax.media.Time;
import javax.media.MediaLocator;
import javax.media.protocol.*;
import java.io.IOException;

public class DataSource extends PushBufferDataSource {
 public PushBufferStream [] getStreams() {
    if (streams == null) {
        streams = new LiveStream[1];
        stream = streams[0] = new LiveStream();
    }
    return streams;
 }
}
```

위의 클래스 정의에서처럼 패키지 선언을 통하여 jmfsample.media.protocol.live라고 명시하였다. 추후에 여러분은 JMFREgistry를 통하여 데이터소스의 등록을 위하여 패키지 부분의 프로토콜 입력단에서 jmfsample을 등록하고 JMStudio에서 Open URL을 통하여 live: 로 경로를 지정하면 여러분이 지정한 커스텀 데이터소스를 획득할수있다. 위에서 정의한 getStreams() 메쏘드는 PushBufferStream 배열을 반환한다. 내부적인 구현에서 보는바와 같이 이 부분에서 LiveStream이라는 새로운 클래스를 이용하게된다. 이 클래스는 PushBufferStream 클래스와 쓰레드 구동을 위한 Runnable을 구현하게된다.

[리스트 10] PushBufferDataSource 클래스의 멤버 변수

```
protected Object [] controls = new Object[0];
protected boolean started = false;
protected String contentType = "raw";
protected boolean connected = false;
protected Time duration = DURATION_UNKNOWN;
protected LiveStream [] streams = null;
protected LiveStream stream = null;
```

[리스트 10]에서 보는것처럼 PushBufferDataSource 를 확장한 DataSource 클래스에서 사용할 멤버변수를 정의하였다. 본 정의에서는 DataSource의 ContentType을 가공되지않은 데이터형 즉 " Raw" 로 정의하였으며, DataSource의 시간정보인 duration은 정의를 할 수 없으므로 DURATION_UNKNOWN으로 지정하였다. 실제 데이터소스가 담겨지는 버퍼타입으로 LiveStream 을 사용하였다. 여기까지 구현함으로서 기본적인 PushBufferDataSource의 클래스 설계가 이루어진다. 그러나 PushBufferDataSource는 javax.media.protocol.DataSource 에서 확장되기 때문에 javax.media.protocol.DataSource에서 선언된 메쏘드들을 다시 정의해주어야 한다.

[리스트 11] javax.media.protocol.DataSource의 메쏘드

```
public String getContentType() {
  if (!connected){
      System.err.println("Error: DataSource not connected");
      return null;
   }
   return contentType;
}
```

```
public void connect() throws IOException {
   if (connected)
      return;
   connected = true;
}

public void disconnect() {
   try {
       if (started)
                       stop();
   } catch (IOException e) {}
       connected = false;
}

public void start() throws IOException {
    if (!connected)
              throw new java.lang.Error("DataSource must be connected before it can be
                                started");
    if (started)
              return;
    started = true;
    stream.start(true);
}

public void stop() throws IOException {
   if ((!connected) || (!started))
            return;
   started = false;
   stream.start(false);
}


public Object [] getControls() {
    return controls;
}
```

```
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;
        }
}

public Time getDuration() {
    return duration;
}
```

먼저 살펴볼 함수는 getContentType()이다. 이 메쏘드는 데이터소스가 내부적으로 어떠한 포맷으로 구성되어있는지를 알려주는 메쏘드이다. 본 구현에서는 " raw" 타입으로 지정을 해주었으며 데이터소스가 Connected 되지 않았다면 Null을 반환하게 된다. 다음으로 살펴볼 메쏘드는 connect(), disconnect() 메쏘드이다. 이 메쏘드들에서는 플래그를 이용하여 실제 데이터 소스의 연결 여부를 결정하여 주는역할을 수행하고, 데이터소스의 연결이 끊겼을 경우 stop()메쏘드를 호출하게끔 구현되었다. 다음은 start(), stop() 메쏘드의 구현이다. 이 부분에서 직접적으로 LiveStream 버퍼를 구동시키게 되며, 내부적으로는 LiveStream의 쓰레드가 구동되어 데이터 읽기 작업을 수행하게된다. 다음으로 구현될 메쏘드는 getControls()이다. 이 메쏘드들은 데이터소스내에 ControlPanelComponent, VisualComponent , CacheingControl Component등의 지원되는 컨트롤의 배열을 반환하는 역할을 수행한다. 마지막으로 구현될 메쏘드는 getDuration() 로서 본 구현에서는 DURATION_KNOWN을 지정하였다.

이제는 PushBufferStream을 구현상속하는 LiveStream을 구현하여보자. 먼저 PushBufferStream에 대해 언급하고자 한다. 이 클래스는 버퍼형태로 데이터를 Push하는 인터페이스로서 소스 스트림을 유저에게 전체 미디어 데이터 묶음으로서 전달하는 역할을 수행한다. 이때 전달되는 미디어 객체는 javax.media.Buffer의 객체형태이며 이러한

스트림을 이용할때는 Buffer 객체를 할당하고 read() 메쏘드의 소스 스트림으로 전달을 해주어야한다. 소스스트림은 이러한 버퍼 객체의 데이터와 헤더정보를 할당하고, 버퍼에 위치시키며 유저에게 데이터를 전달하게된다. 또한 스트림내부의 데이터 및 헤더정보는 버퍼객체의 포맷속성이나 소스스트림의 컨텐츠 타입을 통하여 정의되어질 수 있다.

[리스트 12] LiveStream의 생성자

```
public LiveStream() {
  if (videoData) {
    int x, y, pos, revpos;
    size = new Dimension(320, 240);
    maxDataLength = size.width * size.height * 3;
    rgbFormat = new RGBFormat(size, maxDataLength,
                        Format.byteArray,
                        frameRate,
                        24,
                        3, 2, 1,
                        3, size.width * 3,
                        VideoFormat.FALSE,
                        Format.NOT_SPECIFIED);
    data = new byte[maxDataLength];
    pos = 0;
    revpos = (size.height - 1) * size.width * 3;
    for (y = 0; y < size.height / 2; y++) {
        for (x = 0; x < size.width; x++) {
          byte value = (byte) ((y*2) & 0xFF);
          data[pos++] = value;
          data[pos++] = 0;
          data[pos++] = 0;
          data[revpos++] = value;
          data[revpos++] = 0;
          data[revpos++] = 0;
        }
        revpos -= size.width * 6;
      }
    } else {
      audioFormat = new AudioFormat(AudioFormat.LINEAR,
```

```
                                8000.0,
                                8,
                                1,
                                Format.NOT_SPECIFIED,
                                AudioFormat.SIGNED,
                                8,
                                Format.NOT_SPECIFIED,
                                Format.byteArray);
                                maxDataLength = 1000;
        }
 thread = new Thread(this);
}
```

본 구현에서는 영상과 음성을 지원하는 커스텀 데이터 소스의 구현을 위하여 생성자 부분에서 위와같이 비디오 포맷과 오디오 포맷을 정의하였다. 입력을 위한 영상의 포맷은 320*240 크기로 RGB포맷으로 지정을 하였으며, 오디오의 경우 8000 샘플링 주파수를 이용하였다. 커스텀 데이터소스에 데이터를 생성해주는 부분을 유심히 살펴보기 바란다.

[리스트 13] SourceStream 관련 메쏘드

```
public ContentDescriptor getContentDescriptor() {
      return cd;
}
public long getContentLength() {
      return LENGTH_UNKNOWN;
}
public boolean endOfStream() {
      return false;
}
```

이제 구현할 부분은 SourceStream에서 명시되어있는 3가지의 메쏘드이다. GetContentDescriptor(), getContentLength(), endOfStream()의 3가지 메쏘드를 정의해준다. 본 구현에서는 실제적인 데이터 소스의 길이를 알지못하기 때문에 LENGTH_UNKNOWN을 지정하여 주었다.

[리스트 14] PushBufferStream 관련 메쏘드

```
int seqNo = 0;
```

```
double freq = 2.0;

public Format getFormat() {
  if (videoData)
      return rgbFormat;
  else
    return audioFormat;
}

public void read(Buffer buffer) throws IOException {
  synchronized (this) {
      Object outdata = buffer.getData();
    if (outdata == null ||
        !(outdata.getClass() == Format.byteArray) ||
        ((byte[])outdata).length < maxDataLength) {
        outdata = new byte[maxDataLength];
        buffer.setData(outdata);
    }
    if (videoData) {
      buffer.setFormat( rgbFormat );
      buffer.setTimeStamp( (long) (seqNo * (1000 / frameRate) * 1000000) );
      int lineNo = (seqNo * 2) % size.height;
      int chunkStart = lineNo * size.width * 3;
     System.arraycopy(data, chunkStart,
                                        outdata, 0,maxDataLength - (chunkStart));
     if (chunkStart != 0) {
        System.arraycopy(data, 0, outdata, maxDataLength - chunkStart, chunkStart);
     }
  } else {
      buffer.setFormat( audioFormat );
      buffer.setTimeStamp( 1000000000 / 8 );
      for (int i = 0; i < 1000; i++) {
          ((byte[])outdata)[i] = (byte) (Math.sin(i / freq) * 32);
          freq = (freq + 0.01);
          if (freq > 10.0)
              freq = 2.0;
```

```
        }
    }
    buffer.setSequenceNumber( seqNo );
    buffer.setLength(maxDataLength);
    buffer.setFlags(0);
    buffer.setHeader( null );
    seqNo++;
  }
}
public void setTransferHandler(BufferTransferHandler transferHandler) {
      synchronized (this) {
              this.transferHandler = transferHandler;
              notifyAll();
      }
}

void start(boolean started) {
      synchronized ( this ) {
              this.started = started;
              if (started && !thread.isAlive()) {
               thread = new Thread(this);
               thread.start();
              }
              notifyAll();
       }
}
```

이제 구현할 부분은 PushBufferStream에 관련된 메쏘드들이다. 구현된 메쏘드들중에서 read() 메쏘드 구현에 주의하기 바란다. 이 부분에서는 버퍼내의 데이터처리를 위하여 비디오, 오디오 각 부분별로 구현을 하였으며, 비디오데이터의 경우 클래스 생성과정에서 이미 데이터를 지정해주었으며, 오디오의 버퍼데이터를 위해서는 이 부분에서 오디오 주파수값을 발생시켜 넣어주었으며, 마지막 부분의 setTransferHandler 를 통하여 스트림에 데이터 전송서비스를 등록시키는 역할을 수행하게된다. 독자들은 이러한 데이터 발생 부분을 수정함으로서 파일등에 raw 데이터로 저장된 값을 읽어들여 JMStudio등에서 플레이 하도록 할수있다. 파일입출력을 통한 데이터를 획득하여 이 부분에 대하여 적용하는 것은 직접 구현해보기 바란다.

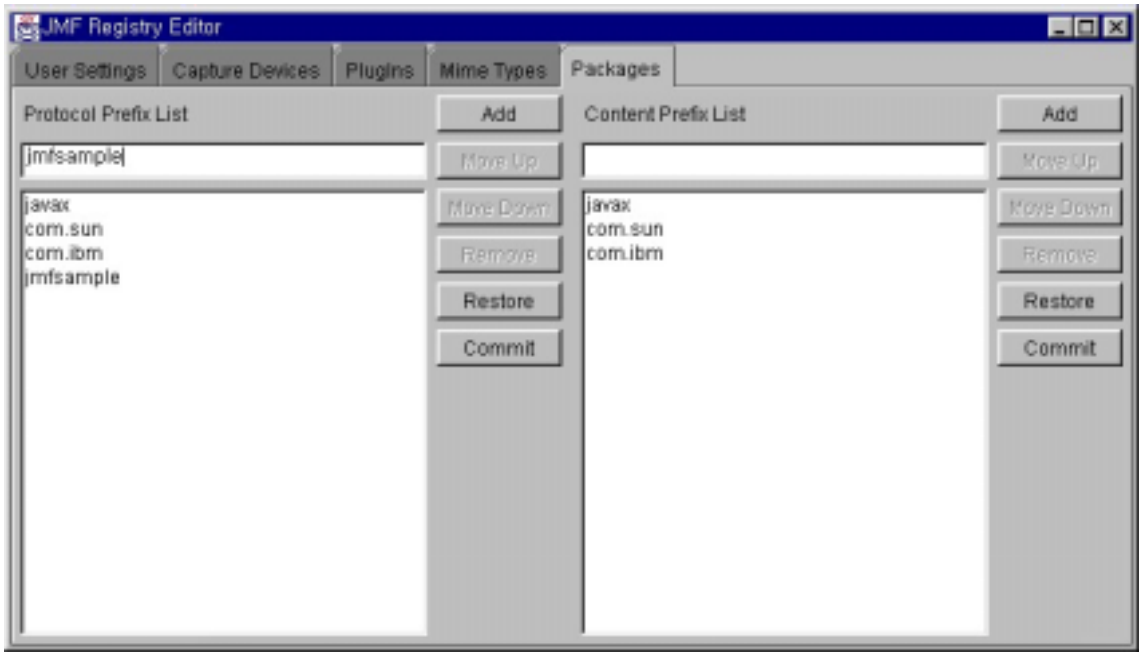
[리스트 15] 쓰레드 객체 구동부분

```
public void run() {
      while (started) {
            synchronized (this) {
                  while (transferHandler == null && started) {
                  try {
                        wait(1000);
                  } catch (InterruptedException ie) {
                  }
            }
       }
   if (started && transferHandler != null) {
            transferHandler.transferData(this);
            try {
                  Thread.currentThread().sleep( 10 );
            } catch (InterruptedException ise) {
                  }
            }
   }
}
public Object [] getControls() {
      return controls;
}
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) {     // no such controlType or such control
              return null;
```

```
        }
    }
}
```

최종적으로 구현될 부분은 앞서 구동한 쓰레드의 run() 메쏘드 구현부분이다. 이 부분에서 여러분은 현재 버퍼의 내용을 직접 전달해주는 transferHandler.transferData(this); 부분을 구동하여야 한다. GetControls() 메쏘드에서는 지원 가능한 컨트롤의 배열을 획득한다.

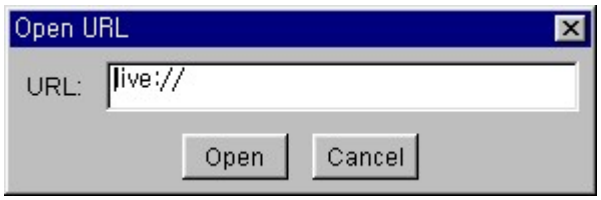


[그림 2] 데이터 소스의 등록

이제는 위에서 구현한 커스텀 데이터소스를 직접 등록해서 JMF가 인식하도록 하여야한다. 우리가 생성한 커스텀 데이터소스는 JMFRegistry의 packages 부분에서 protocol Prefix List를 통하여 등록을 하여야한다. 본 구현에서는 jmfsample.media.protocol.live 와 같이 패키지를 구성하였기 때문에 등록되는 이름은 jmfsample이 된다.

[그림 3] 커스텀 데이터소스 얻어오기

JMFRegistry를 통하여 등록된 데이터소스를 직접 이용하기 위하여 JMStudio를 구동하여 보자. File 메뉴의 Open URL을 통하여 우리는 커스텀 데이터소스의 위치를 지정하여 실제적인 데이터소스로부터 데이터를 획득할 수 있다.



[그림 4] 데이터 소스의 위치지정

Open URL 선택을 하는 부분에서 커스텀 데이터의 실제적인 위치인 live:// 를 입력한다. 본 구현에서는 jmfsample.media.protocol.live 와 같이 지정되었기 때문에 protocol 다음에 지정되는 이름이 실제 커스텀 데이터소스의 URL로 인식되어져 커스텀 데이터소스를 획득하게 된다.

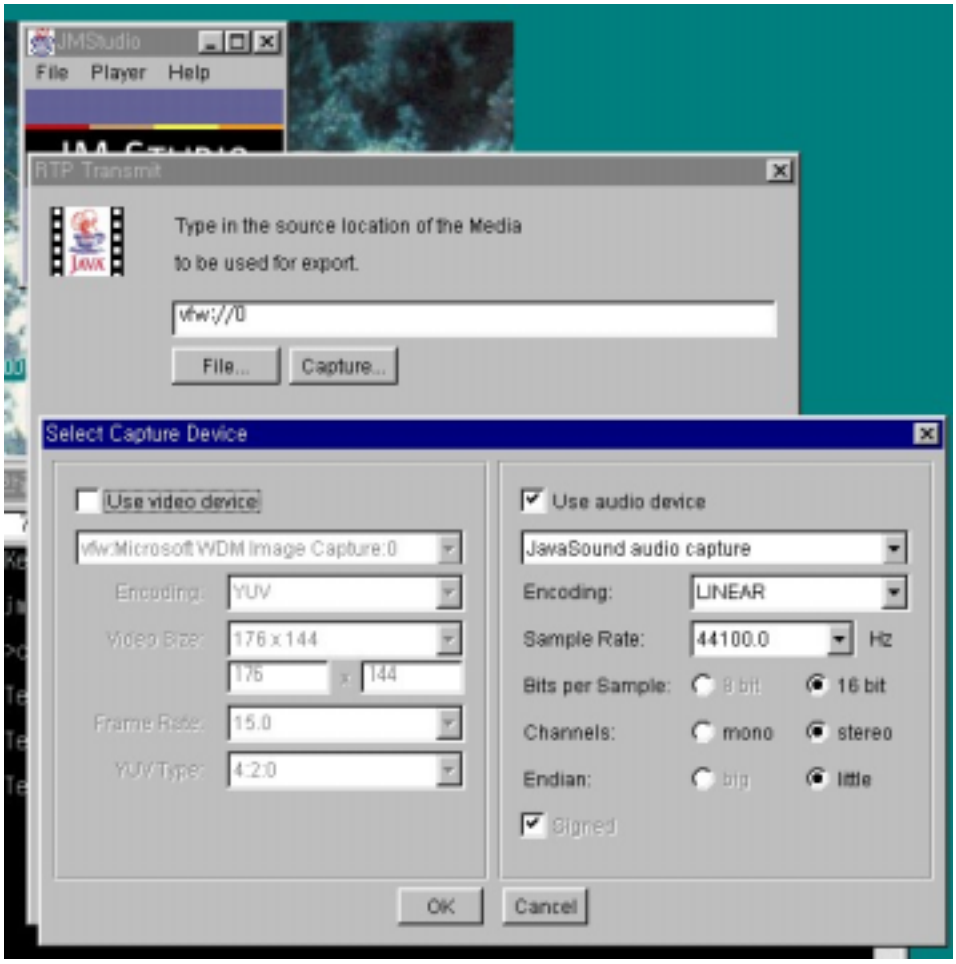
[그림 5] JMStudio를 통한 커스텀 데이터소스의 플레이

이제 JMStudio를 통하여 커스텀데이터가 재생되는 것을 확인하여 보자. 본 구현에서는 커스텀 데이터소스의 실제 버퍼내용에 의미없는 값만을 입력하였기 때문에 [그림 5]에서 처럼 특정색으로 표현되는 화면만 보일것이다. 또한 오디오 커스텀 데이터 소스의 결과를 확인할때는 특정주파수음이 일정한 주기를 가지고 반복되는 것을 확인할 수 있다. 앞서도 언급한바와 같이 독자들은 커스텀 데이터소스의 버퍼 값을 직접 바꾸거나 파일에서 읽어들이는 작업을 수행함으로서 여러분 자신들만의 커스텀 데이터 소스를 작성 할 수

있을것이다.

JMStudio 를 이용한 RTP 통신 이용하기

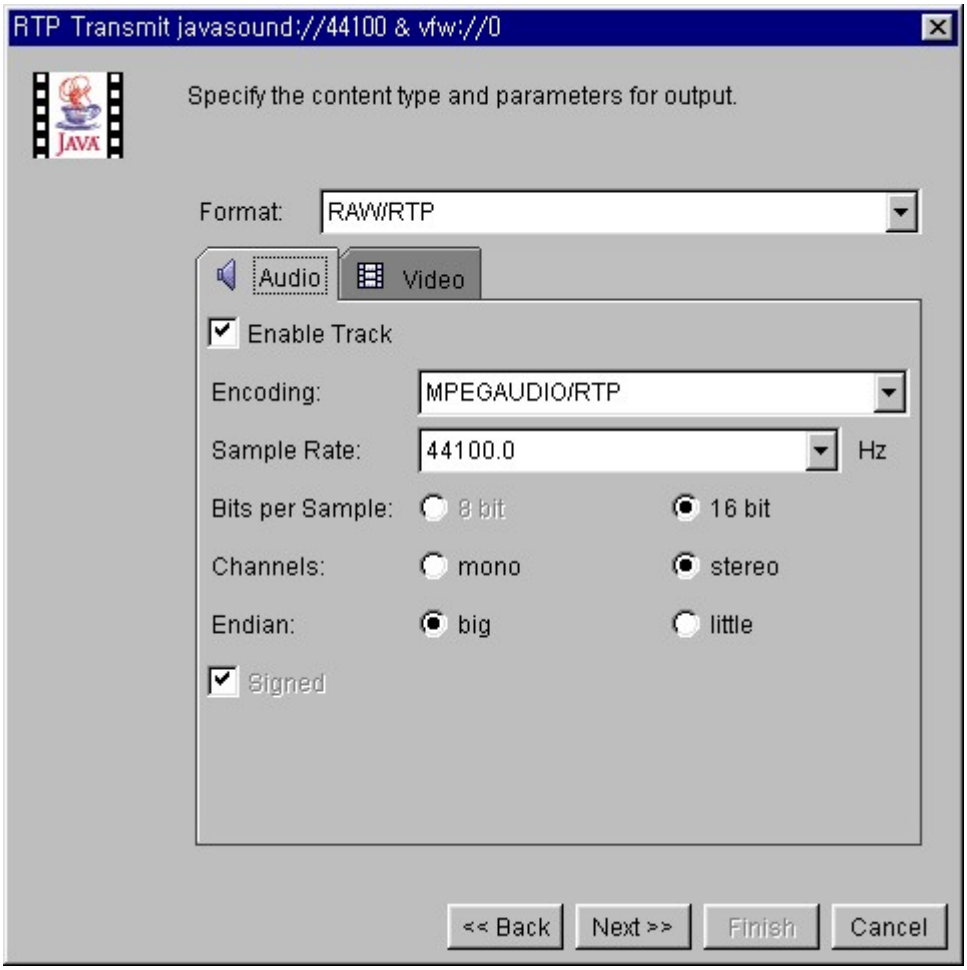
이제까지 상당히 많은 부분에 걸쳐 로컬에서의 JMF 관련 기술들에 언급을 하였다. JMF 2.x에서 향상된 기능중의 하나가 바로 RTP(RealTime Transport Protocol)의 지원이다. 먼저 자세한 RTP의 특징과 구조 및 JMF에서의 프로그래밍 방법을 논의하기 이전에 JMStudio를 통하여 RTP 통신을 직접 경험해보도록 하자.



[그림 6] JMStudio에서의 RTP 통신 설정

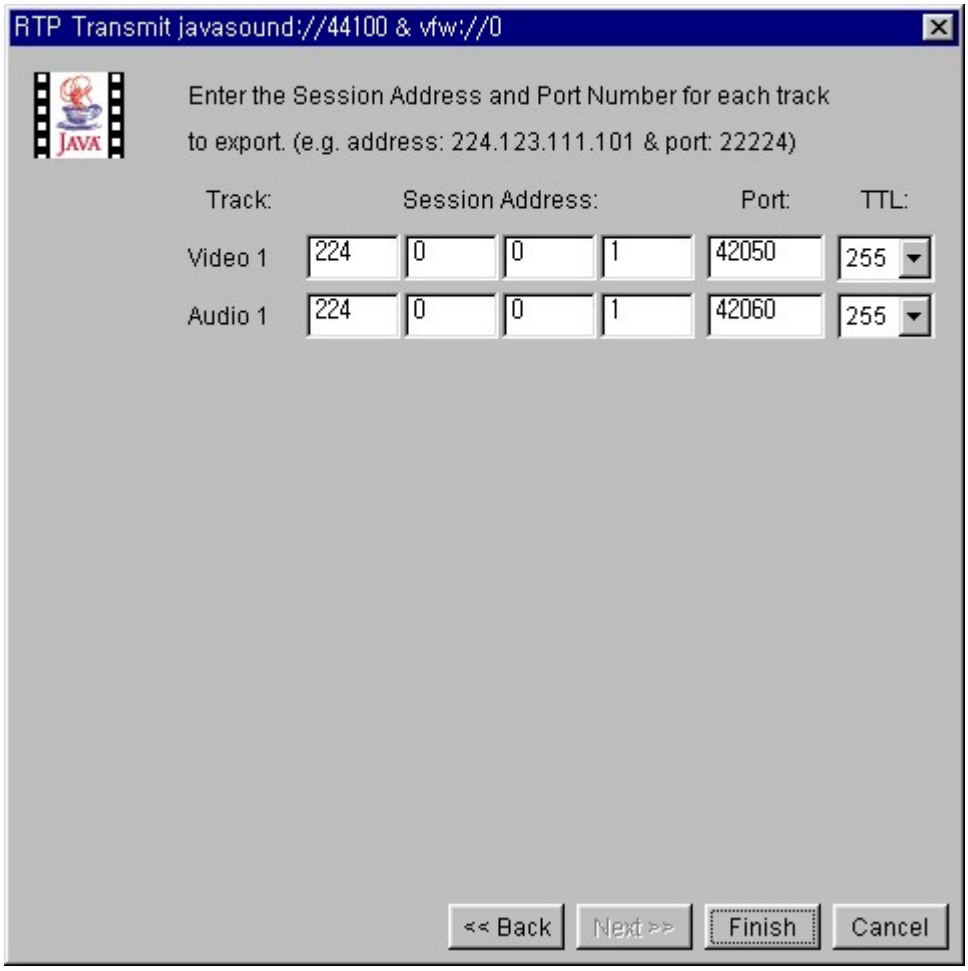
JMStudio를 구동하여 File 메뉴의 Transmit 를 선택하면 RTP Transmit 선택 판넬을 확인할 수 있다. 이 판넬에는 File, Capture의 두개의 버튼이 있는데 눈치빠른 독자들은 이미 감을 잡았을 것이라 생각된다. File 버튼은 로컬 컴퓨터에 있는 동영상/음악 파일을 선택하여 다른 컴퓨터로 데이터를 보내는 것이며, Capture 버튼은 자신의 시스템에 오디오 혹은 비디오 캡쳐장치가 있다면 데이터를 캡쳐해서 상대 컴퓨터에게 데이터를 보내는 방법이다.

Capture 버튼을 선택하면 여러분의 시스템 환경에 따라 설정된 결과치를 Select Capture Device 판넬로 보여준다. 이 부분에서 독자들은 오디오 캡쳐, 비디오 캡쳐의 선택유무를 지정할 수 있으며, 시스템에서 지원하는 각종 캡쳐정보들을 원하는 사양으로 바꾸어 선택할수 있다.



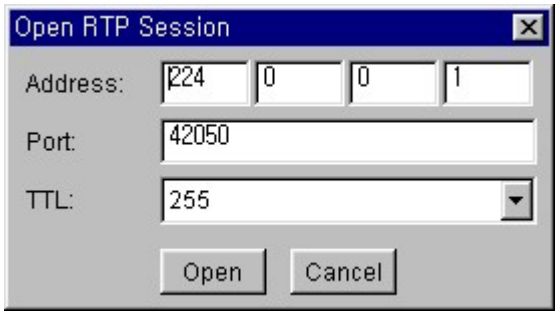
[그림 7] RTP 포맷의 선택

앞서서 오디오와 비디오 캡쳐장치를 선택하였다면 다음은 RTP 프로토콜로 데이터를 전송할 때 각각 어떠한 포맷으로 전송을 할지를 결정하여야 한다. 그림과 같이 오디오 및 비디오 선택 판넬이 있으며, 오디오의 경우 압축/비압축 , 데이터 전송포맷, 샘플링레이트를 조절하고, 비디오의 경우 영상의 크기, 압축방식등을 선택할 수 있다. 독자들의 시스템 사양에 알맞게 선택을 하면 [그림 8]과 같은 판넬이 생성된다. 이 부분에서는 현재 여러분의 컴퓨터에서 데이터들을 어디로 보내야 할런지를 결정해 준다. 즉, 수신측의 IP주소와 포트번호, TTL값을 지정해 주어야한다. 여러분의 컴퓨터 및 상대 컴퓨터의 IP 주소를 확인하고 포트번호는 현재 시스템의 다른 프로그램이나 서비스가 이용하지 않는 번호를 지정하여 준다. TTL값은 충분히 큰값(64 이상의 값)을 선택한다.



[그림 8] 수신측의 IP 주소와 포트 번호 및 TTL 지정

이제는 영상과 음성이 실제로 상대방의 컴퓨터로 전송이 되는지 확인하여 보자. 수신을 원하는 컴퓨터에서 JMStudio를 구동하여 데이터를 송신하는 송신측의 IP 주소와 포트번호 및 TTL값을 지정하여 준다.



[그림 9] 수신측에서의 송신측 정보 입력

이제 수신측에서도 JMStudio에서 송신측의 데이터가 플레이되는 것을 확인할 수 있을것이다. 이로서 간단히 JMStudio를 이용하여 JMF에서 지원하는 RTP 통신에 대하여

엿보기를 하였다.

이번 강좌는 JMF 부분에서도 다소 어려운 캡쳐 버퍼링과 Renderer 버퍼링의 크기조절을 통하여 이기종 시스템간의 특성에 따른 버퍼링 작업의 구현 방법에 관해 언급하고, 이를 플레이어나 프로세서에 적용하는 시점에 관하여 언급하였다. 두번째로는 커스텀 데이터 소스를 구현하여 비표준적인 데이터를 JMF에서 등록하고 이를 재생하는 방법에 대하여 구현해 보았으며 마지막으로 JMF의 JMStudio를 통하여 RTP 통신을 사용하는 방법에 대해서도 살펴보았다. 다음 강좌에서는 실제적인 RTP통신의 정의와 특징에 대하여 언급하고, JMF에서의 RTP 적용 패키지들과 이벤트처리 구현 부분, 송수신을 위한 클래스 구현 부분에 대하여 언급하고자 한다.