제 7 강좌 : RTP 미디어 데이터의 수신

지난 강좌에서는 JMF 에서 정의한 RTP 관련 패키지들을 이용하기 위하여 프로세서의 설정 방법과 출력 데이터를 획득하는 방법, 데이터 전송을 위한 RTP 포맷 변환 작업, 통신의 구현 방법상의 문제들에 관하여 살펴보았으며, 데이터를 전송하기 위한 실제적인 프로그램 작업들의 구현에 관하여 알아보았다. 또한 웹 브라우져를 통하여 애플릿을 이용한 클라이언트와의 통신 과정에서 나타나는 로컬 시스템에서의 보안 문제의 해결을 위하여 서명된 애플릿을 사용하는 방법을 구현하여 클라이언트의 자원을 이용하는 방법을 논의하였다. 이번 강좌에서는 지난 강좌에 이어 RTP 미디어 데이터의 수신을 위한 직접적인 구현 방법들과 이를 클라이언트 측에서 재생하고 데이터를 표현하는 부분에 관하여 알아보고, 수신중의 데이터 포맷의 변환에 대응하는 방법, 전송 및 수신시의 데이터 발생량을 제어하고 수신 버퍼의 버퍼링을 제어하는 방법과 비디오 데이터의 화질을 제어하는 방법등을 알아보기로 한다. 강좌에 대한 문의나 관련 프로그램은 아래의 홈페이지를 참조하기 바란다.

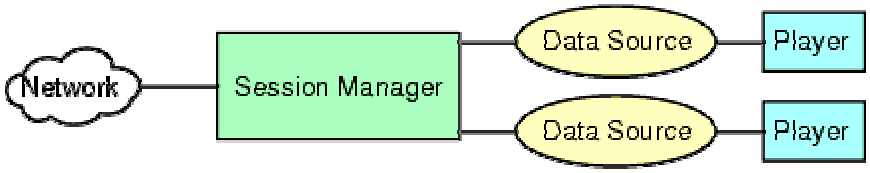
이재훈 전임연구원
삼성테크윈 정밀기기연구소 (Samsung Techwin.)
kingseft@samsung.co.kr
http://myhome.naver.com/kingseft

RTP 미디어 데이터의 수신과 클라이언트측의 표현방법

최근 들어 JMF 에서는 리눅스에서 구동이 가능한 JMF 배포판을 발표하면서 JMF 에 대한 관심이 더욱 커지고 있다. 또한 새롭게 발표된 RTP 패키지들을 통하여 보다 안정적인 RTP 통신을 구현하기 위한 노력이 이루어지고 있으며 향상된 자바 사운드의 기능을 추가하여 이전 버전에 비하여 많은 부분이 기능적인 개선점이 나타나게 되었다. 또한 JMF 는 향후 Video Conference 를 위한 패키지로의 발전을 예상한다고 Sun 에서 발표한 바 있다.

지난 강좌의 내용에서는 RTP 미디어 데이터의 송신과 더불어서 이번 강좌에서는 RTP 데이타 수신을 하는 방법과 그에 따르는 몇가지 부가적인 설명을 하고자 한다. JMF 에서 현재까지 가장 문제가 되고 있고, 또한 JMF-Interest 에서도 가장 큰 이슈로 제시되고 있는 문제가 바로 RTP 를 이용한 데이타 송/수신 문제이다. 현재 넷스케이프 및 인터넷 익스플로러에서도 RTP 부분에서 몇몇 버그를 나타내고 있고, 또한 Sun 에서 공식적으로 버그로 인정한 몇몇 문제점들도 있고 JVM 에서도 RTP 에서 정의한 미디어 데이터의 완전한

지원을 구현하지 못하고 있는 실정이다. 그러나 JMF 의 RTP 패키지들은 현재까지고 계속해서 업그레이드가 되고 있으며, 강좌가 진행되는 현재 시점에서도 베타 테스트 버전이 발표되고 있는 상황이다. 독자들도 주기적으로 JMF 의 RTP 패키지의 새로운 변경 메쏘드들과 구현에 관하여 검토하기를 바란다. 먼저 간략한 RTP 수신을 위한 개념부터 알아보기 위해 [그림 1]을 살펴보자.



[그림 1] RTP 수신과정의 데이터 흐름

[그림 1]에서 각각 별도의 독립적인 플레이어들이 네트워크를 통해 전달된 데이터를 컨트롤하는 세션 메니져를 통해 수신된 각 스트림들을 데이터 소스로 분리한 후에 클라이언트측의 데이터 재생과 저장 및 또 다른 목적을 위해 이용되어진다. 이러한 각각의 수신 스트림에 대하여 Manager 의 createPlayer() 메쏘드를 통하여 Palyer 의 입력 데이터 소스로 이용함으로서 클라이언트측의 데이터 표현을 구현할 수 있다. 이러한 방법으로는 크게 2 가지의 방식으로 구분이 된다.

[클라이언트측의 수신 데이터 표현 방법]
방법 1: URL 위치 정보와 프로토콜 정보를 이용하여 RTP 세션의 파라미터를 갖는 MediaLocator 를 Manager 의 createPlayer() 메쏘드의 인자로서 MediaLocator 정보를 넘겨줌으로서 Player 를 생성하는 방법

방법 2: 입력 스트림으로부터 데이터 소스를 획득하고, 이것을 Manager 의 createPlayer() 메쏘드의 입력 인자인 데이터 소스 정보로 넘김으로서 각각의 ReceiveStream 에 대한 Player 를 생성하는 방법

위의 각 방법들은 나름대로의 장/단점이 존재한다. 여기서 주의할점은 [방법 1]과 같이 MediaLocator 를 이용하여 플레이어를 생성하는 경우에는 단지 세션 내부에서 검색된 여러 개의 RTP 스트림중에서 단지 첫번째 RTP 스트림만을 이용할 수 있으며 나머지 다른 스트림들의 검출 기능을 구현할 수 없다는 것이다. 만약 독자들이 하나의 세션 내에서도 여러개의 다중 RTP 스트림을 검출하고 이를 클라이언트 측에서 재생하기를 원한다면 Session Manager 를 이용하여 각각의 ReceiveStream 에 대한 Palyer 를 생성해야 한다. 대부분의 실제 구현에서는 [방법 1]은 주로 테스트 용으로 많이 이용하며 다중 스트림의 클라이언트측의 재생을 위해서는 반드시 [방법 2]를 이용해야한다. 먼저 [방법 1]에 대한

구체적인 구현방법에 관하여 알아보자.

RTP 세션에 대한 하나의 플레이어 생성방법

MediaLocator 를 이용하여 RTP 세션에 대한 플레이어를 생성할때에는 Manager 는 단지 세션에서 검출된 첫번째 스트림에 대해서만 플레이어를 생성한다는 것과 그외의 다른 모든 스트림들은 인식을 하지 못한다는 것에 주의해야 한다. 이렇게 해서 생성된 플레이어는 세션에서 데이터가 검출되면 RealizeCompleteEvent 이벤트를 발생시킨다. 우리가 프로그래밍을 할 때에는 바로 이 RealizeCompleteEvent 이벤트가 발생되기를 기다렸다가 이 이벤트를 검출하여 데이터가 수신측으로 도착했는지의 여부와 플레이어가 데이터를 재생할 수 있는지의 여부를 판단할 수 있다. 이 과정을 통하여 이벤트가 발생되면 우리는 Visual Component 와 Control Component 를 획득할 수 있다. 여기서 주의할점은 RTP 스트림에 대한 Player 는 세션내에서 데이터가 검출되기 전까지는 결코 realize 되지 않는다는 것이다. 이같은 제약때문에 이전에 우리가 살펴보았던 Manager 의 createRealizedPalyer() 메쏘드는 세션에서 수신된 데이터의 재생을 위해 이용할 수 없다. 강제적으로 이 메쏘드를 이용하는 경우 데이터 도달시까지 어떤 플레이어도 생성되지 않으며 데이터 수신이 없다면 무한 루프에 빠질 위험이 있다. 이 부분에서 한가지 유용한 정보는 플레이어를 통하여 RTP 에 특화된 컨트롤 정보로서 전반적인 세션 통계정보를 나타내는 RTPControl 을 Export 시킬수있으며 세션 메니져에게 동적 Payload 를 등록시키기 위해 이용되어 진다는 것이다. 이 부분에 대해서는 뒤에서 좀더 자세하게 다루기로 하자. 위에서 언급한 [방법 1]을 구현하기 위하여 [리스트 1]을 살펴보도록 하자.

[리스트 1] RTP 세션에 대한 플레이어의 생성

```
String url= "rtp://224.144.251.104:49150/audio/1";
MediaLocator mrl= new MediaLocator(url);

if (mrl == null) {
   System.err.println("Can't build MRL for RTP");
   return false;
}

try {
   player = Manager.createPlayer(mrl);
} catch (NoPlayerException e) {
   System.err.println("Error:" + e);
```

```
   return false;
} catch (MalformedURLException e) {
   System.err.println("Error:" + e);
   return false;
} catch (IOException e) {
   System.err.println("Error:" + e);
   return false;
}

if (player != null) {
   if (this.player == null) {
      this.player = player;
      player.addControllerListener(this);
      player.realize();
    }
}
```

[리스트 1]은 위에서 언급한 RTP 수신 방법중 MediaLocator 정보를 이용하는 [방법 1]에 속한다. 먼저 MediaLocator 정보를 얻기위하여 위치 정보와 프로토콜 정보를 나타내는 URL 정보를 지정해 주었다. URL 정보 표현에는 수신을 위한 IP 주소와 미디어 데이터의 타입 , 그리고 TTL 값을 지정해 주었다. 이 URL 정보를 통하여 MediaLocator 가 생성되고 Manager 의 createPlayer() 의 인자로서 MediaLocator 를 넘겨주어 플레이어를 생성하게된다. 안정적으로 생성된 플레이어는 이후 Listener 를 추가하는 addControllerListener() 를 이용하고, realize() 메쏘드를 통하여 실제 이용가능한 플레이어 상태로 상태정보가 옮겨진다. ControllerListener 에 대한 이벤트 처리는 controllerUpdate() 메쏘드를 통하여 이용되어지는데 이는 뒤에서 다시 설명하기로 한다.

수신측의 스트림의 포맷변환에 대한 처리

한가지 실제적인 상황을 생각해보자. 만약 수신되는 데이터 스트림의 포맷이 송신측의 요구에 의해 변경되어 다른 포맷으로 수신측에 전달된다면 이를 어떻게 처리해야 할까? 수신측에서 플레이어는 payload 포맷이 변경되는 경우 FormatChangeEvent 이벤트를 발생시킨다. 재미있는 사실은 위에서 소개한 [방법 1] 즉, MediaLocator 를 통해서 생성된 플레이어는 이러한 payload 변환에 대하여 프로그래머의 개입없이 자동적으로 처리를 해준다는 점이다. 그러나 실제 프로그래밍의 측면에서는 주로 [방법 2]로 구현하기 때문에

payload의 변환에 대한 이벤트 처리를 위해서 새로운 포맷을 처리하기 위해서 기존의 플레이어를 제거하고 그에 관련된 각종 Component를 모두 제거해야 하며, 변경된 payload를 적용하기 위하여 새로운 플레이어를 생성해야 한다는 것이다. 이를 구현하기 위해서는 플레이어는 FormatChangeEvent 이벤트를 검출할 수 있어야 하며 그 결과로 기존에 이용하던 플레이어를 삭제하고 이에 관련된 Visual / Control Component 역시 변경되었는지를 검사한 후에 삭제하고 모두 다시 생성해 주어야 한다. 이에 대한 구현은 [리스트 2]로 나타내었다.

[리스트 2] Payload 변환이벤트에 대한 처리

```
public synchronized void controllerUpdate(ControllerEvent ce) {
    if (ce instanceof FormatChangeEvent) {
        Dimension vSize = new Dimension(320,0);
        Component oldVisualComp = visualComp;
        if ((visualComp = player.getVisualComponent()) != null) {
            if (oldVisualComp != visualComp) {
                if (oldVisualComp != null) {
                    oldVisualComp.remove(zoomMenu);
                }
                framePanel.remove(oldVisualComp);
                vSize = visualComp.getPreferredSize();
                vSize.width = (int)(vSize.width * defaultScale);
                vSize.height = (int)(vSize.height * defaultScale);
                framePanel.add(visualComp);
                visualComp.setBounds(0,
                                     0,
                                     vSize.width,
                                     vSize.height);
                addPopupMenu(visualComp);
            }
        }
        Component oldComp = controlComp;
        controlComp = player.getControlPanelComponent();
        if (controlComp != null)
        {
            if (oldComp != controlComp)
            {
                framePanel.remove(oldComp);
```

```
                framePanel.add(controlComp);
                if (controlComp != null) {
                    int prefHeight = controlComp
                                    .getPreferredSize()
                                    .height;
                    controlComp.setBounds(0,
                                          vSize.height,
                                          vSize.width,
                                          prefHeight);
                }
            }
        }
    }
}
```

[리스트 2]를 살펴보자. 이 프로그램은 [리스트 1]에서 생성된 플레이어의 addControllerListener() 메쏘드를 통하여 이벤트가 처리되는 부분으로서 구현 부분은 controllerUpdate() 메쏘드 내부에서 전부 구현이 된다. Payload 변환을 인식하는 이벤트인 FormatChangeEvent 이벤트를 통하여 플레이어의 Visual / Control Component 를 획득한다. 이를 위해서는 각각 플레이어의 getVisualComponent(),getControlPanelComponent() 메쏘드가 이용된다. 이 부분에서는 payload 가 변경된 경우 기존의 컴포넌트들과 새로 획득된 컴포넌트들을 비교해서 다른경우 기존의 컴포넌트들을 삭제하고 새롭게 컴포넌트를 부착시키는 과정을 나타내었다.

세션내의 각 새로운 스트림에 대한 Player 의 생성 방법

앞에서도 언급했지만, 실제적인 프로그래밍 관점에서 RTP 수신을 위해서는 [방법 2]를 주로 이용한다. 일단 하나의 Session 이 만들어 지고 나면, 생성된 세션 내부에는 여러 종류의 스트림이 존재할 수 있다. 이렇게 하나의 세션안에서 모든 ReceiveStreams 들에 대한 플레이어를 생성하기 위해서는, 각각의 스트림에 대해서 별도로 플레이어를 생성해 주어야 하며, 각각의 플레이어에 대한 이벤트 처리 역시 독립적으로 구현을 해주어야 한다. 즉, 개별적인 스트림에 대해 각각 하나씩의 player 를 만들어 주어야 한다. 새로운 스트림이 생성되었을때 세션 메니져는 NewReceiveStreamEvent 이벤트를 발생시키게되고, 그러면 우리는 이러한 이벤트의 처리를 위해서 public void update(ReceiveStreamEvent event) 함수를 정의해야 한다. 일반적으로 이벤트 처리는 ReceiveStreamListener 로서

등록을 해야하고, 각각의 새로운 스트림에 대해서 플레이어를 생성해야 한다. player 의 생성을 위해서는 ReceiveStream 으로부터 DataSource 를 얻어와야 하고, 이렇게 얻어진 DataSource 를 Manager.createPlayer()의 인자로 넘겨 주어야 한다.

특정한 하나의 Session 내에서 각각의 새로운 receive stream 에 대해 player 를 생성하기 위해서는 아래의 단계를 거쳐야 한다.

[새로운 스트림에 대한 수신단의 처리과정]
1. Setup RTP Session 부분의 단계는 아래의 절차을 따라야 한다.
a. SessionManager 를 생성한다.
   예를들어서 com.sun.media.rtp.RTPSessionMgr 의 인스턴스를 만든다.
b. RTPSessionMgr 의 addReceiveStreamListener()를 호출하여 리스너를 등록한다.
c. RTPSessionMgr 의 initSession()을 호출하여서 RTP session 을 초기화 한다.
d. RTPSessionMgr 의 startSession()을 호출하여서 RTP session 을 시작한다.

2. ReceiveStreamListener 의 update() 메쏘드에서 NewReceiveStreamEvent 에 대한 이벤트 처리함수를 작성해야 한다. 이 이벤트는 새로운 데이타 스트림이 검출 되었을때를 알려준다.

3. NewReceiveStreamEvent 가 검출되면, getReceiveStream 을 호출해서 NewReceiveStreamEvent 로부터 ReceiveStream 을 획득한다.

4. getDataSource 를 호출해서 ReceiveStream 으로부터 RTP DataSource 를 획득한다. 이 DataSource 는 RTP specific format 을 가지는 PushBufferDataSource 이다. 예를들자면 , DVI 오디오 player 에 대한 부호화 인코딩은 DVI_RTP 가 되는것이다.

5. 위에서 얻은 DataSource 를 Manager.createPlayer 로 넘겨서 player 를 생성하도록 한다. player 가 성공적으로 생성되기 위해서는, 특정한 RTP 포맷화된 데이타에 대한 Decoding 과 Depacketin 을 위한 plug-in 이 설치되어있어야 한다. 독자들도 이것을 반드시 확인해 보기 바란다.

실제로 위에서 언급한 방법을 그대로 따라 코딩을 한다면 RTP 수신단의 구현에는 큰 어려움이 없을것이다. 이제 앞에서 설명한 부분에 대한 실제 프로그램 구현 방법에 관하여 알아보자. [리스트 3]을 확인해본다.

[리스트 3] update 함수에대한 ReceiveStreamEvent 처리
```
public void update( ReceiveStreamEvent event){
```

```
SessionManager source =(SessionManager)event.getSource();
Player newplayer = null;
if (event instanceof NewReceiveStreamEvent){
      try{
        ReceiveStream stream = ((NewReceiveStreamEvent)event).getReceiveStream();
        DataSource dsource = stream.getDataSource();
        newplayer = Manager.createPlayer(dsource);
      }catch (Exception e){
        System.err.println("RTPPlayerApplet Exception " + e.getMessage());
        e.printStackTrace();
      }
      if (newplayer == null){
          return;
      }
      if (source == videomgr){
        if (videoplayer == null){
              videoplayer = newplayer;
              newplayer.addControllerListener(this);
              newplayer.start();
         }
        else{
            if (playerlist != null)
               playerlist.addElement((Object)newplayer);
               new PlayerWindow(newplayer);
            }
      }
    if (source == audiomgr){
        if (playerlist != null)
              playerlist.addElement((Object)newplayer);
              new PlayerWindow(newplayer);
         }
    }
```

[리스트 3]에서는 update() 메쏘드 내부의 각종 이벤트 처리 과정중 NewReceiveStream 이벤트 처리를 구현한 것이다. 이 이벤트가 발생되면 getReceiveStream() 메쏘드를 통하여

수신된 스트림을 획득하고, 획득한 스트림으로부터 getDataSource()를 호출하여 이용가능한 데이터 소스를 얻어 이를 Manager 의 createPlayer() 메쏘드의 인자로 넘겨줌으로서 새로운 플레이어를 생성하게 된다. 다음으로 데이터 소스의 종류를 결정해야 한다. 본 구현에서는 새롭게 수신된 데이터 소스가 비디오 데이터 소스인 경우와 오디오 데이터소스인 경우를 구별하여 각각 플레이어를 독립적으로 생성하게 하였다. update() 메쏘드는 인자로서 ReceiveStreamEvent 이벤트를 전달 받는다. 내부적인 구현에서는 이렇게 전달받은 이벤트의 소스를 구하기 위하여 getSource() 메쏘드를 통하여 세션 메니져를 획득하게 된다. 다음으로 할 작업은 이 이벤트가 NewReceiveStreamEvent 이벤트인가를 판단하는 과정이다. 이 과정을 통하여 새로운 스트림이 전달되면 getReceiveStream() 메쏘드를 통하여 수신측에 전달되는 스트림을 획득하고 이렇게 수신된 스트림으로부터 getDataSource() 메쏘드를 이용하여 플레이어의 생성을 위한 데이터 소스를 얻는 과정이다. 또한 이 부분에서는 이벤트의 소스인 세션 메니져가 비디오 수신을 위한 세션 메니져인지 혹은 오디오 수신을 위한 세션 메니져인가를 판별하여 각각 독립적으로 처리를 하였다. 이전 강좌에서도 언급을 하였지만, RTP 통신은 오디오 및 비디오 데이터에 대한 독립적인 송신과 수신을 하여야 하며, 이를 위하여 각각의 데이터 소스에 따라 세션 메니져 역시 개별적으로 구현해야 한다는 것에 주의하기 바란다.

다음으로는 앞에서 언급한 Receive Steram 에서 새로운 스트림의 출현에 대한 처리과정을 구현한 프로그램을 살펴보자. [리스트 4] 를 살펴보자. 이 부분에서 다루고 있는 ReceiveStream 이벤트와 그 내부 이벤트인 NewReceiveStreamEvent 및 ReceiveStream 의 획득과 획득된 스트림으로부터 데이터 소스의 추출과 이를 통한 플레이어의 생성 과정등의 일련의 흐름을 이해하여야 한다.

[리스트 4] NewReceiveStreamEvents 에 대한 처리 부분

```
public void update( ReceiveStreamEvent event)
{
    Player newplayer = null;
    RTPPlayerWindow playerWindow = null;
    SessionManager source = (SessionManager)event.getSource();
    if (event instanceof NewReceiveStreamEvent)
    {
        String cname = "Java Media Player";
        ReceiveStream stream = null;
        try
        {
            stream =((NewReceiveStreamEvent)event).getReceiveStream();
```

```
            Participant part = stream.getParticipant();
            if (part != null) cname = part.getCNAME();
            DataSource dsource = stream.getDataSource();
            newplayer = Manager.createPlayer(dsource);
             System.out.println("created player " + newplayer);
        } catch (Exception e) {
             System.err.println("NewReceiveStreamEvent exception " + e.getMessage());
             return;
        }
        if (newplayer == null) return;
        playerlist.addElement(newplayer);
        newplayer.addControllerListener(this);
        playerWindow = new RTPPlayerWindow( newplayer, cname);
        }
    }
}
```

[리스트 4]의 update() 메쏘드의 NewReceiveStreamEvent 에 대한 이벤트 처리과정을 살펴보자. 먼저 getReceiveStream()을 통하여 수신되는 스트림을 획득할 수 있으며, 이렇게 수신된 스트림은 세션에 참여하여 통신을 수행하는 Participant 정보를 갖게된다. 우리는 이 부분에서 getParticipant() 메쏘드를 통하여 세션에 참여한 사람의 정보를 획득할 수 있다. 다음은 수신 스트림의 getDataSource() 메쏘드를 통하여 플레이어를 생성하기 위한 데이터 스트림을 확보하는 과정이다. 이후에는 스트림으로부터 데이터 소스를 얻고 이 데이터 소스를 Manager의 createPlayer의 인자로 넘겨 플레이어를 생성하게된다. 마지막으로 생성된 플레이어들도 역시 각각의 이벤트 처리를 위하여 addControllerListener 를 호출하여 이벤트 처리 등록을 시켜준다.

원격 송신지의 RTP payload 변경에 대한 대응방법

이번에는 프로그램 작성시 많은 프로그래머들이 대부분 간과하고 넘어가는 부분에 대하여 살펴보자. 만약 송신측에서 RTP payload의 변경, 즉 송신 데이터의 포맷을 변경한다면 수신단에는 반드시 이 부분에 대한 처리를 해주어야 한다. 세션 메니져에서는 자동적으로 payload의 변환에 대한 처리를 하는 기능이 없기 때문이다. 이때 ReceiveStream 은 RemotePayloadChangeEvent 이벤트를 발생하게된다. 그러므로 payload가 변경되는 경우 현재까지 이용하던 플레이어는 이렇게 새로운 포맷을 다룰수가 없게되며, 이러한 새로운

payload 를 이용하려고 한다면 JMF에서는 Exception을 발생하게된다. 이러한 문제점의 해결을 위해서 우리가 작성하는 ReceiveStram Listener는 반드시 RemotePayloadChangeEvent 이벤트에 대한 처리 작업을 추가시켜 주어야 한다. RemotePayloadChangeEvent 이벤트가 검출되면 다음과 같은 작업이 필요하다. [리스트 5]에서는 이러한 과정을 구현하였다.

[리스트 5] RemotePayloadChangeEvent 처리

```
public void update(ReceiveStreamEvent event) {
  if (event instanceof RemotePayloadChangeEvent) {
       if (newplayer != null) {
          newplayer.stop();
          newplayer.removeControllerListener(listener);
          newplayer.close();
         try {
              rtpsource.connect();
              newplayer = Manager.createPlayer(rtpsource);
              if (newplayer == null) {
                 System.err.println("Could not create player");
                 return;
               }
               newplayer.addControllerListener(listener);
               newplayer.realize();
          } catch (Exception e) {
               System.err.println("could not create player");
           }
        }
     }
 }
```

[RemotePayloadChangeEvent 처리 과정]

1. 현재 존재하는 플레이를 Close 시킨다.
2. 이렇게 제거된 플레이어에서 이용하던 모든 리스너를 제거한다.
3. 동일한 RTP 데이터 소스를 이용하여 새로운 플레이어를 생성한다.
4. 새롭게 생성된 플레이어로부터 Visual/ Control Component를 획득한다.
5. 필요한 이벤트 리스너를 등록시키고 이벤트 처리를 진행한다.

[리스트 5]에서는 RemotePayloadChangeEvent 이벤트의 검출을 통하여 먼저 이전에 구동되고

있던 플레이어의 상태 정보를 검사한 후에 stop() 메쏘드를 통하여 플레이어의 구동을 중지시키고 removeControllerListener() 메쏘드를 통하여 플레이어에 부착되었던 리스너를 제거하여야 하며 마지막으로 close() 메쏘드를 통하여 플레이어를 종료해야 한다. 또한 많은 독자들이 실수를 하는 부분이 한가지 있다. 바로 connect() 메쏘드를 통하여 획득한 데이터 소스를 다시 연결시켜 준후에 플레이어를 생성시켜야 한다는 점이다. 이 과정을 수행한 후에 플레이어에 리스너를 등록시키고 realize() 메쏘드를 통하여 사용가능한 안정한 플레이어를 획득한다.

입력 RTP 스트림에 대한 버퍼링 제어

수신되는 데이터를 어떻게 효율적으로 제어할 수 있을까? JMF의 RTP 에서는 입력되는 데이터에 따라서 버퍼링 제어 작업이 가능하다. 이 작업은 Session Manager에서 생성되는 BufferControl 을 통하여 RTP 수신 버퍼를 제어하는 방법이다. 이 BufferControl 을 통하여 2개의 파라미터를 제어할 수 있는데 이것은 각각 버퍼의 길이와 버퍼 임계값의 제어이다. 버퍼의 길이는 수신단에 의해 유지되는 수신 버퍼의 크기를 말하며 버퍼 임계값은 데이터를 Push / Pull 하기 이전에 컨트롤에 의해 버퍼링 되어지는 최소 데이터 량을 표시한다. 그러므로 수신 되어지는 모든 데이터가 유효한 것이 아니라 수신 데이터의 크기가 이 버퍼 임계값 이상이 되었을 때에만 유효한 데이터로서 이용이 가능해진다는 것이다. 만약 버퍼화된 전체 데이터의 크기가 이 버퍼 임계값보다 작다면 데이터는 이 버퍼 임계값에 도달할 때까지 다시 버퍼링 되어진다. 버퍼의 길이와 버퍼 임계값은 모두 밀리세컨드 단위로 지정이 되며, 오디오 패킷의 갯수나 비디오 프레임임의 갯수는 입력되는 스트림의 포맷에 따라 달라진다. 각각의 수신되는 스트림은 버퍼 길이와 최소 버퍼 임계값에 대한 디폴트 값과 최대값을 갖게된다. 세션에 대한 BufferControl을 얻기 위해서는 Session Manager의 getControl() 메쏘드를 통하여 획득할 수 있으며, BufferControl에 대한 getControlComponent()를 호출함으로서 GUI Component를 얻을 수 있다. [리스트 6]을 통해 구현에 필요한 사양에 대하여 알아보자.

[리스트 6] 수신 버퍼 컨트롤 얻기

```
import javax.media.control.BufferControl;
Button monitorb, closeb, bufferb;
mypanel.add(bufferb = new Button("Buffer Control for RTP Session"),"South");
bufferb.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        buffercontrol();
    }
```

```
});
public void buffercontrol() {
  if (manager != null) {
    BufferControl b =
      (BufferControl) manager.getControl ("javax.media.control.BufferControl");
           Component c = null;
           if (b != null)
                      c = b.getControlComponent();
           mypanel.remove(bufferb);
           mypanel.add(c, "South");
           parent.pack();
           parent.show();
       }
}
```

[리스트 6]의 구현처럼 BufferControl 을 위해서 먼저 javax.media.control.BufferControl 를 통하여 BufferControl 을 사용할 것을 알린다. 버퍼 컨트롤을 위해서는 먼저 버퍼 컨트롤이 정의되어 있는 패키지의 사용을 알려주어야 한다. 실제적인 구현을 위하여 버퍼 기능의 선택을 위한 버튼을 구현하고 버튼 이벤트 처리를 위한 ActionListener()를 부착시키었다. 사용자에 의한 버튼의 입력 이벤트가 발생하면 buffercontrol() 메쏘드를 호출하고 실제 버퍼 컨트롤 기능은 buffercontrol() 메쏘드의 내부에서 구현된다. 버퍼 컨트롤을 얻기 위하여 BufferControl 컨트롤을 선언하고 manager의 getControl() 메쏘드를 통하여 버퍼 컨트롤을 획득하게 된다. 또한 버퍼 컨트롤에서 제공하는 컴포넌트를 얻고자 하는 경우 버퍼 컨트롤의 getControlComponent() 메쏘드를 호출하여 획득할 수 있다.

수신 프로그램의 구현

이번에는 몇가지 수신 프로그램의 예제를 살펴보면서 RTP 수신 구현부분에 대하여 알아보도록 하자. JMF의 영상 음성 기반의 통신 이외에도 Socket, UDP, TCP/IP 기반의 다른 통신 프로그램들을 결합하여 채팅창이나 화이트보드등을 여러분들이 작성한 프로그램에 결합 시키는 것도 가능하다. 먼저 비디오 스트림 데이터의 수신 과정을 살펴보도록 하자.

비디오/ 오디오 수신 부분의 구현

이번에는 직접적인 구현된 프로그램들을 살펴보며 앞서서 언급한 이론들을 어떻게 프로그래밍으로 구현하는가에 대하여 알아보도록 하자. 먼저 비디오의 수신 과정을 위해서는 다음과 같이 구현하였다.

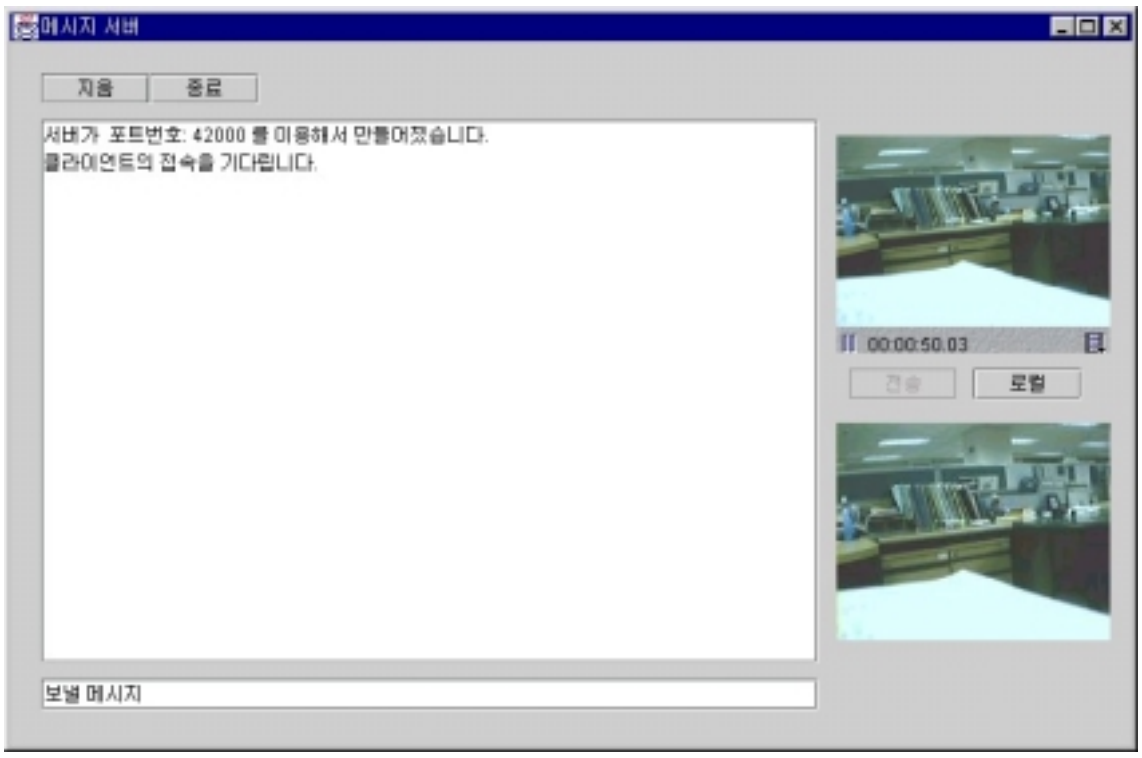
[리스트 7] 비디오 수신 클래스의 생성자 부분

```
public VideoReceiver(){
  playerlist = null;
  loweredbevel = BorderFactory.createLoweredBevelBorder();
  raisedbevel = BorderFactory.createRaisedBevelBorder();
  panel = new JPanel();
  panel.setLayout(new BorderLayout());
  panel.setBorder(BorderFactory.createCompoundBorder(raisedbevel, loweredbevel));

  address = "224.0.0.1;
  port = 42050;
  media = "video";
  StartSessionManager(address, port, "video");
  if (videomgr == null){
      System.err.println("null video manager ");
      return;
  }
  panel.setPreferredSize(new Dimension(250,200));
  add(panel);
}
```

[리스트 7]은 비디오 수신을 위한 클래스의 생성자 부분으로서 먼저 비디오의 플레이어 리스트를 벡터형태로 구성을 하고 화면의 디자인을 구성한다. RTP 수신을 위한 3가지 파라미터인 수신을 위한 주소, 수신포트와 오디오/비디오의 구별을 위한 인자값을 정의한 후에 StartSessionManager() 메쏘드를 정의하여 이를 호출하였다. 주의 할 점은 포트 번호를 지정할 때에는 반드시 짝수로 지정을 해주어야 지정한 포트 번호 + 1 번호로 컨트롤 데이터를 위한 포트가 함께 지정되어 질 수 있다는 점이다. 또한 현재 자신의 시스템에서 이미 사용중인 포트를 지정하는 오류를 피해야 한다. 또한 [리스트 7]에서 지정한 IP 주소인 224.0.0.1 과 같이 송/수신 시에 멀티 캐스팅 IP 그룹인 IP 클래스 D의 주소를 이용하면 세션에 참여하는 모든 참여자들과의 데이터 송수신이 가능하여 진다. 만약 특정 참여자와의 통신만을 구현하고자 한다면 수신측에서는 송신측의 IP 주소를 바로 지정함으로서 유니 캐스트 통신 형태로서 세션을 수립하고 통신을 수행하게 된다. [그림 1]은 이러한 JMF 수신

프로그램과 일반적인 채팅 프로그램을 혼합한 구현 예이다.



[그림 1] JMF 의 RTP 통신부분과 채팅 창의 결합부분

[리스트 8]은 [리스트 7]에서 새롭게 작성한 StartSessionManager 의 실제 구현 부분이다. StartSessionManger 클래스는 SessionManager 클래스를 이용하며, 실제 모든 RTP 수신 작업을 위한 초기화 작업을 구현한다.

[리스트 8] 세션 메니져 클래스의 구현

```
private SessionManager StartSessionManager
(String destaddrstr, int port, String media)
{
    SessionManager mymgr = new RTPSessionMgr();
    if (media.equals("video")) videomgr = mymgr;
    if (mymgr == null) return null;
    mymgr.addReceiveStreamListener(this);

   String cname = mymgr.generateCNAME();
   String username = "jmf-user";
   SessionAddress localaddr = new SessionAddress();
```

```
try{
    destaddr = InetAddress.getByName(destaddrstr);
 }catch (UnknownHostException e){
    System.err.println("inetaddress " + e.getMessage());
    e.printStackTrace();
 }
SessionAddress sessaddr = new SessionAddress(destaddr, port, destaddr, port+1);
SourceDescription[] userdesclist = new SourceDescription[4];
int i;
for(i=0; i< userdesclist.length;i++){
     if (i == 0){
          userdesclist[i] = new SourceDescription
          (SourceDescription.SOURCE_DESC_EMAIL, "jmf-user@sun.com", 1, false);
           continue;
      }
     if (i == 1){
         userdesclist[i] = new
         SourceDescription(SourceDescription.SOURCE_DESC_NAME, username,1, false);
         continue;
     }
    if ( i == 2){
       userdesclist[i] = new SourceDescription
                (SourceDescription.SOURCE_DESC_CNAME, cname, 1, false);
       continue;
    }
    if (i == 3){
        userdesclist[i] = new SourceDescription
           (SourceDescription.SOURCE_DESC_TOOL, "JMF RTP Player v2.0", 1, false);
         continue;
     }
} // end of for
try{
  mymgr.initSession(localaddr, mymgr.generateSSRC(), userdesclist, 0.05, 0.25);
  mymgr.startSession(sessaddr,1,null);
}catch (SessionManagerException e){
     System.err.println("RTPPlayerApplet: RTPSM Exception " + e.getMessage());
```

```
        e.printStackTrace();
        return null;
   }catch (IOException e){
        System.err.println("RTPPlayerApplet: IO Exception " + e.getMessage());
        e.printStackTrace();
        return null;
    }
    return mymgr;
  }
}
```

[리스트 8]을 통하여 먼저 RTPSessionMgr()을 통하여 세션 메니져를 생성하고, 인자로 넘어온 미디어 타입에 따라서 비디오 메니져 혹은 오디오 메니져를 생성하는 부분을 구현하였으며, User 정보를 넣어주는 부분과 initSession, StartSession을 통하여 전체적인 세션의 수신 부분이 구현이 된다. [리스트 9] 에서는 세션을 통해 전달되는 데이터 송수신의 시작과 중단 기능을 구현한 것이다.

[리스트 9] 데이터 수신의 시작과 중단

```
public void start(){
 if (videoplayer != null){
    videoplayer.start();
 }
 if (playerlist == null)
    return;

  for (int i =0; i < playerlist.size(); i++){
        Player player = (Player)playerlist.elementAt(i);
        if (player != null) new PlayerWindow(player);
   }
} // start() end

public void stop() {
   if (videoplayer != null){
        videoplayer.close();
        System.out.println(" 수신이 정지되었슴!! ");
    }
```

```
    if (playerlist == null) return;
    String reason = "Shutdown VideoPlayer";
   for (int i =0; i < playerlist.size(); i++){
        Player player = (Player)playerlist.elementAt(i);
        if (player != null){
            player.close();
        }
   }
   if (videomgr != null) {
      videomgr.closeSession(reason);
      videoplayer = null;
      videomgr = null;
   }
}
```

[리스트 9]에서는 플레이어의 메쏘드인 start(), stop() 기능과 유사한 기능을 구현하였으며, 특히 stop() 메쏘드 내에서 플레이어의 close() 함수 호출후에 비디오 및 오디오 세션 자체를 close() 시키는것에 주의하여야 한다.

RTP 송수신 구현의 팁

이번에는 RTP 송수신 과정에서 많은 량의 데이터를 요구하는 비디오 스트림에 대한 효율적 관리를 위하여 몇가지 팁을 소개하고자 한다. 송수신되는 비디오의 크기는 비디오의 캡쳐 포맷과 비디오의 압축률에 따라 그 데이터의 크기가 변하게된다. 이 부분에서는 프로그램에서 직접 비디오 캡쳐 소스의 크기를 선택적으로 획득하는 방법과 JPEG의 압축률을 변경하여 선택적으로 전송하는 방법을 구현하여 본다. 먼저 [리스트 10]을 살펴보자.

[리스트 10] 비디오 이미지의 캡쳐 크기 선택

```
public class CaptureSizeDialog extends JDialog implements ActionListener{
  public DataSource source ;
  public FormatControl[] formatControls = null;
  public Format[] formats = null;
  public Format format = null;
  public Choice  list = new Choice();
```

```
  public JButton button = new JButton("OK");

  public CaptureSizeDialog (Frame panel) {
    super(panel, true);
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            cancel();
        }
    });
    button.addActionListener(this);
    init ();
  }
  public void putDataSource(DataSource ds){
        source = ds;
  }
  public void actionPerformed ( ActionEvent event ) {
        String          strCmd;
        strCmd = event.getActionCommand ();
        if ( strCmd.equals("OK") ) {
        open(list.getSelectedIndex());
    }
 }
 private void init () {
    this.setTitle ( "원하는 캡쳐사이즈를 선택하세요" );
    this.getContentPane().setLayout ( new BorderLayout() );
    this.getContentPane().add ( list, BorderLayout.CENTER );
    this.getContentPane().add ( button, BorderLayout.SOUTH );
    pack ();
    setVisible(false);
  }

public void add(String str){
        list.addItem(str);
  }
  public void setVisible(boolean show) {
      super.setVisible(show);
```

```
  }
  public void open(int index) {
    formatControls = ((CaptureDevice)source).getFormatControls();
    System.out.println(formatControls.length);
    formats = formatControls[0].getSupportedFormats();
    System.out.println(formats.length);
    format = formats[index];
    formatControls[0].setFormat(format);
    System.out.println("format-"+format);
    System.out.println("format control 에서 뽑은 format-
          +(formatControls[0].getFormat()).toString());
    setVisible(false);
  }
 public void cancel() {
       setVisible(false);
  }
}
```

[리스트 10]은 비디오 캡쳐 드라이버에서 얻어지는 여러가지 비디오 캡쳐 이미지의 크기를 선택하여 이를 우리가 사용하는 캡쳐드라이버로 지정해 주는 부분을 구현한 것이다. 전체적인 구현은 다이얼로그 클래스를 상속받아 다이얼로그 기반의 프로그램으로 구성되어 있으며 여러분들이 눈여겨 볼 부분은 바로 open() 메쏘드와 putDataSource() 메쏘드이다. 이 클래스는 JDialog 를 상속받은 다이얼로그 기반의 클래스이고 버튼 처리를 위하여 ActionLIstener 를 구현하였다. 생성자에서는 부모 프레임을 인자로 받아들이며 윈도우 리스너와 버튼리스너를 추가하느 과정을 수행하며, init() 메쏘드를 통하여 생성한 다이얼로그의 속성을 변경하는 과정을 수행한다. add() 메쏘드는 캡쳐 데이터 소스를 획득할때마다 리스트 형태로 추가를 하게되며, 사용자의 버튼 입력에 대응하여 open() 메쏘드를 호출하게 된다. 실제의 중요한 구현은 putDataSource() 메쏘드와 open() 메쏘드에서 구현이 된다. putDataSource()를 통하여 획득한 데이터 소스를 입력받는 과정을 거치고, open() 메쏘드에서는 사용자가 선택한 캡쳐 데이터 소스의 인덱스 정보를 이용하여 putDataSource() 메쏘드에서 획득된 캡쳐 장치의 formatControls 정보를 획득하고 이렇게 획득된 컨트롤에서 getSupportedForamts() 메쏘드를 통하여 현재의 컨트롤에서 지원되는 포맷 정보를 얻게된다. 다음으로 사용자가 선택한 인덱스 번호에 따라서 formatControls 의 setForamt() 메쏘드를 통하여 실제적인 포맷 변환이 이루어 지게된다. [리스트 11]은 위의 [리스트 10]을 실제로 이용하는 부분이다.

[리스트 11] 캡쳐 이미지소스의 크기 지정

```
JFrame testFrame = new JFrame();
CaptureSizeDialog d = new CaptureSizeDialog(testFrame);
d.setVisible(false);

for( i=0; i<nCount; i++ ) {
  infoCaptureDeviceVideo = (CaptureDeviceInfo)MediaDevice.elementAt(i);
  arrFormats = infoCaptureDeviceVideo.getFormats();
  System.out.println("비디오 캡쳐의 지원포맷은 : " + arrFormats.length);
  for( j=0; j<arrFormats.length; j++ ) {
       if ( arrFormats[j] instanceof VideoFormat ) {
          VideoDevice.addElement(infoCaptureDeviceVideo);
          System.out.println("Find Video Capture Device");
          Dimension size = ((VideoFormat)arrFormats[j]).getSize();
          String encoding = ((VideoFormat)arrFormats[j]).getEncoding();
          float frameRate = ((VideoFormat)arrFormats[j]).getFrameRate();
          String frame = String.valueOf(frameRate);
          System.out.println(size.width + " " + size.height
                      + " " + encoding + " " + frame);
        // 캡쳐 크기 선택박스에 추가
        d.add(size.width + " " + size.height + " " + encoding + " " + frame);
   }
}
}

try {
  dsVideo = Manager.createDataSource(infoCaptureDeviceVideo.getLocator());
  d.putDataSource(dsVideo);
  d.setVisible(true);
} catch (NoDataSourceException e) {
  System.out.println("Video 파트 : NoDataSourceException !!!");
} catch (IOException e) {
  System.out.println("Video 파트 : IOException !!!! ");
}
```

[리스트 11]은 이미 익숙한 코드일 것이다. 이 프로그램에서는 기본적으로 시스템의 비디오

캡쳐 장치를 찾고, 비디오 캡쳐 데이터 소스를 찾을때마다 그것을 [리스트 10]의 다이얼로그 박스에 추가하게 된다. 이렇게 추가되는 데이터 소스들은 캡쳐 다이얼로그 박스에서 인덱스의 번호를 가지며 리스트를 형성하고, 사용자의 선택에 의하여 원하는 크기로 비디오의 캡쳐 크기가 정해지게된다. 생성된 비디오 캡쳐 데이터 소스는 캡쳐 다이얼로그 박스의 putDataSource() 함수에 의하여 캡쳐 다이얼로그에 전해지게 되고, 이후 open() 함수의 내부에서 데이터 소스의 getFormatControls() 을 통하여 formatControls 을 획득한 후에 얻어진 다양한 포맷 컨트롤의 첫번째 요소에서 getSupportedFormats() 을 이용하여 formats 정보를 획득한다. 이 formats 정보에서 사용자가 선택한 인덱스 번호를 입력한후 다시 formatControls 의 setFormat()을 이용하여 최종적으로 포맷 지정을 완료함으로서 사용자 정의의 캡쳐 데이터 소스 크기를 조정하여 송수신시의 데이터 발생량을 줄일 수 있다.

JPEG_RTP 의 압축률 조정을 통한 데이터 발생량의 조절

RTP 송수신에서 발생되는 비디오 데이터의 크기를 효과적으로 줄일 수 있는 또 하나의 방법은 바로 비디오 압축방식의 선택과 더불어 압축율의 조정을 그 예로 들 수 있다. RTP 에서 이용하는 비디오 압축 방식으로는 H.263 과 MPEG, JPEG 등이 있으나 본 예제에서는 JPEG_RTP 와 압축률을 통한 데이터 발생량의 조절에 관하여 알아보기로 하자. 먼저 JPEG_RTP 로 영상을 전송하기 위하여 비디오 포맷을 변경해야 할것이다. [리스트 12]를 살펴보도록 하자.

[리스트 12] RTP 전송을 위한 비디오 포맷변경

```
try {
  RTPprocessorVideo = Manager.createProcessor(srcVideo);
} catch ( NoProcessorException e ) {
  System.out.println("Video: No Processor Exception");
} catch ( IOException e ) {
  System.out.println("Video: IO Exception ");
}
RTPprocessorVideo.configure();
while(!(RTPprocessorVideo.getState() == Processor.Configured)) {}
TrackControl track2Video[] = RTPprocessorVideo.getTrackControls();
for( i=0; i<track2Video.length; i++ ) {
   if( track2Video[i].getFormat() instanceof VideoFormat ) {
        if( ((FormatControl)track2Video[i]).setFormat(new
```

```
            VideoFormat(VideoFormat.JPEG_RTP)) == null ) {
               track2Video[i].setEnabled(false);
             } else {track2Video[i].setEnabled(true);
            }
        }
    }
    RTPprocessorVideo.setContentDescriptor( new
        ContentDescriptor(ContentDescriptor.RAW));
    RTPprocessorVideo.addControllerListener(this);
    RTPprocessorVideo.realize();
```

[리스트 12]에서는 먼저 비디오 전송을 위한 프로세서를 생성시키고, configure() 를 호출하여 프로세서의 초기화를 실행하게된다. 프로세서가 완전히 초기화가 된 것을 확인한 후에 프로세서의 getTrackControls() 를 통하여 현재 프로세서가 이용하는 데이터 소스의 트랙별 컨트롤 정보를 획득할 수 있다. 이 트랙정보들 중에서 트랙의 포맷이 비디오 포맷인 경우만을 선별하고 그때의 비디오 포맷을 VideoFormat.JPEG_RTP 로 변경하는 것이다. 이 부분에서 주의할 것은 데이터 소스의 포맷 변경은 반드시 프로세서가 configured 된 후에 실행하며 실행후에는 다시 realize()를 통하여 프로세서를 사용 가능한 상태로 만들어야 한다는 것이다.

다음으로 JPEG_RTP 로 변경한 비디오 포맷을 RTP 로 전송하는 경우에 전송되는 압축률을 조정하는 방법에 관하여 알아보자. 이 부분은 실제로 독자들이 압축률과 수신되는 화면의 화질을 비교해 가면서 검증을 해야 할것이다. 보다 좋은 화질을 얻기 위해서는 압축률을 낮추어야 하지만 그에 따라 발생되는 데이터량은 더 커지게된다. 특히 JPEG 이나 H.261 의 경우 압축률의 커지게 되면 수신되는 화면에서는 극심한 블록킹 현상(Blockking Effect) 이 발생하여 화면에서 마치 모자이크와 같은 현상이 나타나고, 반대로 수신측에서의 우수한 화질을 얻기위해서는 압축률을 낮추기 때문에 전송되야할 데이터량이 크게 증가하는 문제가 있다. 이러한 이유로 이 부분은 독자들이 직접 여러 가지 실험적인 압축률을 대입해가면서 자신의 환경에 가장 적합한 값을 찾기를 바란다. [리스트 13]을 살펴보자.

[리스트 13] JPEG 의 압축률 조절방법

```
void setJPEGQuality(Player p, float val) {
  Control cs[] = p.getControls();
  QualityControl qc = null;
  VideoFormat jpegFmt = new VideoFormat(VideoFormat.JPEG);
  for (int i = 0; i < cs.length; i++) {
```

```
    if (cs[i] instanceof QualityControl && cs[i] instanceof Owned) {
        Object owner = ((Owned)cs[i]).getOwner();
        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);
                  System.err.println("- Setting quality to " +val + " on " + qc);
                  break;
                }
             }
        }
        if (qc != null)
          break;
      }
    }
}
```

[리스트 13]에서는 JPEG의 압축률을 조절하기 위한 구현 부분이다. 이 메쏘드의 인자로 현재 구동중인 플레이어와 목표로 하는 압축률이 필요하다. 먼저 인자로 넘어온 플레이어의 getControls() 를 이용하여 현재 플레이어에서 사용하는 모든 컨트롤들의 정보를 배열로 얻어온다. 이 전체 컨트롤 배열에서 각 컨트롤들이 QualityControl 인가를 판별하고, 그 컨트롤의 소유자 정보가 Codec 이라면 실제로 우리가 이용가능한 QualityControl 이 되는것이다. 이러한 정보중에서 다시 JPEG 코덱인가를 확인한후에 실제적인 setQuality() 메쏘드를 통하여 압축률을 조정하게된다. 특히 이 부분에서 주의할 점은 이 setJPEGQuality() 메쏘드를 언제 호출해야 하는가 즉, 호출 시점이 문제가 된다. 실제적으로 이 메쏘드의 호출은 플레이어가 완전히 사용가능하게 된 후 , 즉, RealizeCompleteEvent 가 발생한 후에 호출을 해야 한다는 것이다. 그러므로 독자들이 압축률 조절을 위해서는 setJPEGQuality(RTPprocessorVideo, 0.5f); 와 같은 메쏘드 호출을 RealizeCompleteEvent 이벤트 처리시에 해주어야 한다는 것이다.

수신 데이터의 저장

이번에는 RTP 를 통하여 전송된 스트림 데이터를 저장하는 방법에 관하여 알아보자. 이전

강좌들에서도 잠시 언급한바 있지만, 데이터의 저장을 위해서는 DataSink 가 이용되어 진다. 먼저 [리스트 14]를 살펴보자.

[리스트 14] 수신 스트림 데이터의 저장

```
MediaLocator saveLocation = new MediaLocator("file://test.mpg");
DataSink Save_DataSink = null;
try{
   Save_DataSink = Manager.createDataSink
                (mediaProcessor.getDataOutput(), saveLocation);
   Save_DataSink.open();
   Save_DataSink.start();
}catch(NoDataSinkException ndse){
  ndse.printStackTrace();
  System.out.println("NoDataSinkException...");
  System.exit(0);
}catch(IOException ioe){
  ioe.printStackTrace();
  System.out.println("IOException...");
  System.exit(0);
}catch(Exception e){
  e.printStackTrace();
  System.out.println("Exception...");
  System.exit(0);
}
System.out.println("Save_DataSink Open and Starting...");

if (Save_DataSink != null)
{
  Save_DataSink.close();
  Save_DataSink = null;
}
System.out.println("Save_DataSink Stop...");
```

[리스트 14]의 구현은 이미 독자들에게도 익숙한 구현이라고 생각된다. 데이터의 저장을 위해서 DataSink 를 이용하고, 실제적인 데이터의 저장을 위해서는 MediaLocator 를 통하여 file://test.mpg 와 같이 저장을 위한 파일의 이름을 명시하였다. 먼저 프로세서의

getDataOutput() 메쏘드를 통하여 우리가 가공한 프로세서의 출력 데이터 소스를 획득하고, saveLocation 을 통하여 저장될 MediaLocator 를 명시하여 Manager 의 createDataSink() 메쏘드를 통하여 DataSink 를 생성한다. 이렇게 생성된 DataSink 는 이제 open() 메쏘드와 start() 메쏘드를 통하여 로컬 시스템의 파일로 저장이 되며 파일의 종료를 위해서 close() 메쏘드를 통하여 DataSink 를 닫음으로서 파일의 저장이 종료된다.

이번 강좌에서는 RTP 통신의 수신측의 구현에 관하여 알아보았다. 수신을 위한 기본적인 두가지의 구현 방법에 관하여 논의하였으며, 수신시의 버퍼링과 전송되는 데이터의 포맷이 변경될때의 대응방법 및 네트워크상의 전송 데이터량을 조절하기 위한 영상 캡쳐 데이터 소스의 변경과 영상 압축코덱의 압축률을 조정하는 방법에 관하여 논의하였다. 다음 강좌에서는 RTP 를 이용한 응용 프로그램과 RTP 를 이용하는 네트워크 환경에 대하여 알아보도록 하고, 또한 JMF 와 비교되는 몇몇 다른 구현들에 관해서도 알아보도록 한다.

[용어설명]

1. Player Component
JMF 에서의 Player Component는 Visual Component와 Control Component로 나누어 지며, 각각 재생되는 데이터의 실제 화면 및 화면 하단부의 컨트롤패널부분으로 구분된다. 각각의 컴포넌트들은 getVisualComponent() 메쏘드와 getControlPanelComponent() 를 통하여 획득되어 지며, 사용자가 정의한 Panel등에 하나의 부속 컴포넌트로 부착될 수 있다.

2. Receive Buffer
RTP 데이터을 수신할 때 수신단에서 설정되어지는 버퍼의 크기와 최소 버퍼크기량을 결정하여, RTP 데이터 수신을 조절하는 컨트롤이며 javax.media.control.BufferControl 를 통하여 정의된다.

3. Blocking Effect
JPEG, MPEG, H.261등의 DCT(Discrete Cosine Transform) 기반의 영상 압축/복원 코덱에서는 압축률을 높임에 따라서 주파수 변환된 영상의 고주파 부분을 제거함으로서 복원시 화면에 모자이크 블럭 같은 현상이 발생한다. 이러한 현상의 해결을 위하여 수신측에서 Loop Filter , Low Pass Filter 등을 이용하기도 한다.