4

Audio and Video in Depth

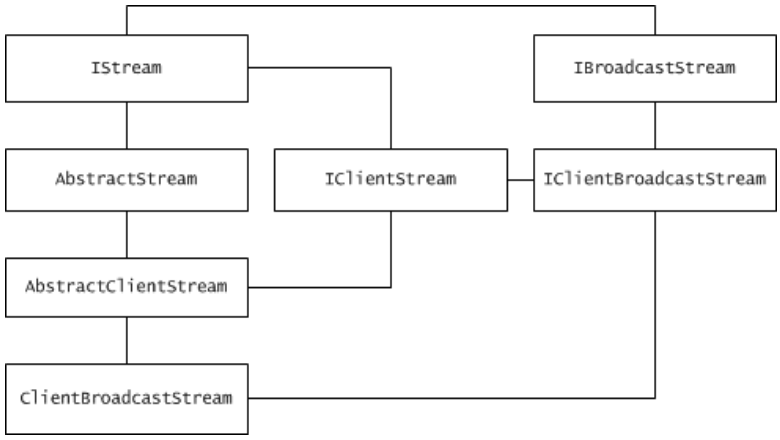
It could certainly be said that streaming media makes up the majority of what goes on, on the Internet. From your favorite video portal to internet radio, the things we watch or listen to are almost all streamed using a media server of some sort. Although many of these providers use a commercial product like FMS or Apple's Darwin streaming server, a few of them stream with Red5. As an aside, the Red5 development team is aware that Facebook and Yahoo labs have used Red5 and may have it deployed in their datacenters.

In this chapter we will cover:

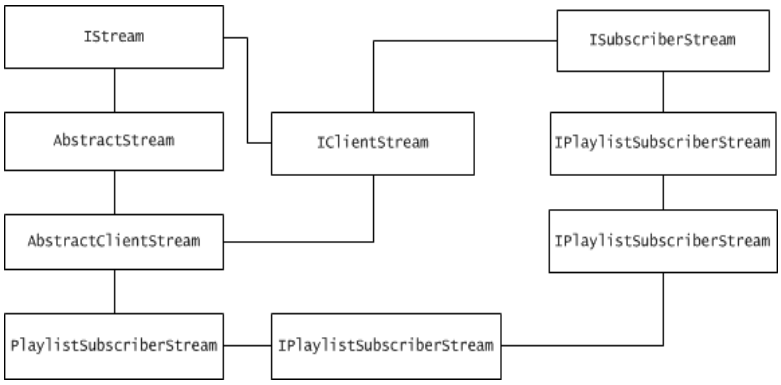
- Managing streams
- Stream listeners
- Supported codecs
- Supported file types
- Bypassing firewalls

Managing streams

The act of managing streams on a media server means to publish, play, pause, stop, and delete in most cases. Since Red5 is completely open to whatever we can create, management of our streams may take on a multitude of variations based on these concepts. Stream management is handled via different classes depending upon where the management function is performed. On the client-side when using Flash, management of streams is handled in ActionScript via the NetStream class. The NetStream class is very well documented across the Internet and in Adobe literature, so herein we will only cover the features applicable to Red5 streaming. Red5 provides several classes for stream management on the server-side, two very important classes are the ClientBroadcastStream and PlaylistSubscriberStream both of which descend from the IStream interface.



The diagram above shows the relationship with the IStream interface for publishers. As a point for reference, most classes with an “I” prefix will consist of an interface in the Red5 codebase.



This diagram shows the relationship with the IStream interface for subscribers.

What's in a name?

A stream name is used as a locator for data either currently being streamed or which will be streamed at a future time. Data streaming in this instance means any digital information, whether it be audio, video, text, images, or anything else which can be encoded into bytes and sent over the network. Stream names must be unique per scope, we cannot have two streams with the name of “mystream” for instance. The name selected for our live stream can pretty much be anything which has a string representation and may be a valid file name on the servers operating system. This essentially means

letters, numbers, and a small set of punctuation characters; the underscore "_" and dash "-" are usually safe in terms of cross-platform support. For prerecorded streams which are often referred to as video on demand (VOD), the stream name would be the name used for the recorded file (i.e. mystream.flv).

Naming collisions may be prevented in several ways, such as querying the application for the current list of stream names, by adding a time value suffix, or using a sever controlled sequence number. To return a list of stream names for a given scope name, we can utilize this method.

```
public ArrayCollection<String> getScopeStreamList(String
scopeName) {
     ArrayCollection<String> streams = new
ArrayCollection<String>();
     IScope target = null;
     if (scopeName == null) {
        target = Red5.getConnectionLocal().getScope();
        scopeName = target.getName();
     } else {
        target = ScopeUtils.resolveScope(scope, scopeName);
     }
     List<String> streamNames =
getBroadcastStreamNames(target);
     for (String name : streamNames) {
        streams.add(name);
     }
     return streams;
  }
```

The method returns a collection of all the stream names for a scope and it also contains a little bit of error prevention as well. The getScopeStreamList method would be best placed within our scope handler or application adapter class. Client-side ActionScript used to request the stream list may look like this:

```
nc.call("getScopeStreamList", new Responder(onStreamList),
"streams");
```

This NetConnection call uses a Responder to handle the callback from the server which will contain the list of streams currently broadcasting live in the "streams" scope. The client handler method may be implemented as shown below.

```
public function onStreamList(list:ArrayCollection):void {
  trace("Streams: " + list.length);
  if (list.length > 0) {
```

```
        var stream:String = list.removeItemAt(0) as String;
        trace("Getting " + stream + " from list");
    }
}
```

When working with streams in Red5, it is important to remember that some methods do not return what we might expect. The ClientBroadcastStream class is the default stream implementation and it exposes two methods for working with stream names.

1. getPublishedName – Returns what would normally be expected as the stream name, a name such as “live” or “mystream”.
2. getName – This method on the other hand, returns an internal name which resembles an identifier more than it would a regular stream name. An example return from this method looks like this: 15ccb8f2-2a07-467a-a02a-ff49769fba85.

Publishing in detail

Publishing, is the act of sending our content to a server for subscribers to consume or to record to a file. Of course by content we mean pretty much anything that can be converted into bytes using codecs, as covered later on in this Chapter. Red5 supports several different types of publishing by default:

- Live – Analogous to experiencing something in real-time, this stream is not pre-recorded. An example of live stream would be a video teleconference between to office locations or a video phone call.
- Video on demand – Similar to watching a television show or movie from a DVD. This media occurred in the past and is stored in a compatible digital format which may be played back upon request. VOD content may be fast forwarded, rewound, or played from any available position in the file.
- Playlist - A list of streams or video files that have been grouped together as one stream. The next item in the group is played after the previous one is complete. This may be set to repeat without client input.

Publishing a live stream

Live stream publishing is a very important capability for a media server to support. This feature allows us to broadcast anything in near real-time to as many viewers as we can support with our installation. Since Flash Player version 6, it has been incredibly simple to publish content. Following these five steps gives us the ability to stream to the world.

1. Simply select our audio and video sources

```
var mic:Microphone = Microphone.getMicrophone();
var camera:Camera = Camera.getCamera();
```

2. Connect to a media server using NetConnection

```
var nc:NetConnection = new NetConnection();
nc.connect("rtmp://localhost/myapp");
```

3. Create a NetStream from the NetConnection

```
var ns:NetStream = new NetStream(nc);
```

4. Attach the audio and video source to the NetStream

```
ns.attachAudio(mic);
ns.attachCamera(camera);
```

5. Publish the stream

```
ns.publish("mystream", "live");
```

To stop publishing our stream we simply issue this command.

```
nc.publish(false);
```

Providing the best quality stream possible for our subscribers is a matter of our available bandwidth and the encoder settings that we select, however in most cases the default values that Flash Player selects are sufficient. If we have plenty of bandwidth available and would prefer high quality using standard dimensions, this is how we would configure our camera.

```
camera.setMode(320, 240, 23)
camera.setQuality(0, 90);
```

The default values for setMode are a width of 160 pixels, a height of 120 pixels, and 15 frames per second. Be aware that the Flash Player will not exceed the maximum dimensions allowed by the camera hardware, if our values are too high they will be automatically reset to the highest available value. For setQuality the default values are 0 for quality and 16384 for bandwidth. The valid range for quality is from 0 to 100, in which 1 represents the lowest quality with maximum compression and 100 is highest quality with no compression. If quality is set to 0, picture quality will be varied to prevent exceeding available bandwidth. The bandwidth property is the amount of bandwidth that the video may utilize in byte per second. Setting bandwidth to zero will cause the Flash Player to consume as much bandwidth possible to maintain the quality selected.

A third option to when publishing live is the key frame interval, or the number of frames which are to be sent without being manipulated based on previous frames by the compression algorithm. Valid values are from 1 which means every frame is a key frame to 48 which means every 47 frames there is one full key frame. The lower the interval value, the higher our bandwidth requirements will be since we will be sending a smaller number of compressed frames. In our example code we specified an fps value of 23 so a key frame interval of 7 should provide good quality with regard to our situation.

```
camera.setKeyFrameInterval(7);
```

When publishing a live stream using a Flash-based client, we only have the ability to select from two of the seven available audio codecs. The selection must be performed prior to attaching the microphone to the NetStream. If we want to use Speex we must specify it as shown in the example, if we do not specify Speex then the default which is NellyMoser will be selected for us.

```
//get the audio source
var mic:Microphone = Microphone.getMicrophone();
//select Speex audio encoding
mic.codec = SoundCodec.SPEEX;
//set to the highest quality
mic.encodeQuality = 10;
//attach the microphone to our NetStream
ns.attachAudio(mic);
```

Remember that we are not limited only to audio and video when publishing, we could just as easily publish events via a shared object or a data frame call instead. An excellent example of this kind of publishing is the “shared ball” demo, which could be considered a multiple publisher application.

Publishing a VOD stream

Even more simple than publishing a live broadcast stream, is providing prerecorded media files for subscribers. The default way to accomplish this is to create a “streams” directory within our application directory and place our media files within it. The second way is to create the media file during our live publish, by indicating to the server that we want to record our stream.

```
nc.publish(“mystream”, “record”);
```

This request will create a file named “mystream.flv” in the applications “streams” directory using the default Red5 streaming classes. While the stream is being published it may also be requested as a live stream by subscribers without interfering with the recording processes.

There are times when we may want to append to a previously recorded media file for our stream, this is accomplished by sending the append type in the NetStream publish.

```
nc.publish(“mystream”, “append”);
```

It is important to understand that the type of publish selected will determine the handling of any existing media files for the stream. The following publish rules apply:

- If “live” is specified, no file is created.
- If “record” is selected the file will be created if it does not exist.
- If “live” or “record” are specified and a file already exists for the stream, the file will be deleted.
- If “append” is requested and the file exists, the file will be appended to.
- If “append” is selected and no file exists, the type will be switched internally on the server-side to “record”

Recording media on a Red5 can only be in two forms by default, but this does not prevent us from creating our own media writer implementations for other media types like MP3. Both of the recorded media types are written to an FLV container file and may consist of stream data encoded with any of the supported codecs which are compatible with this container format. Normally the media files will contain Sorenson video with NellyMoser audio, but other codecs are available such as h.264 or On2 VP6 video using existing media encoders. The Flash Media Live Encoder (FMLE) is one such encoder that will work with Red5, but the Adobe's license disallows its use with anything other than FMS.

Playlists

When we have a set of media files to play back as a group, we should use a play list. Play lists may be created on the client-side or server-side. To create a server-side play list in our application, we need to collect the names of the target media files and then decide upon the conditions for our play lists creation. Play lists are usually created when a particular “room” or scope is created so that the content related with its name or description is played back to subscribers in a preset order. Play list streams that are created on the server-side are treated as live streams, in that they do not have a known

fixed duration and seek operations would return unexpected results beyond the first play item.

Using a method which creates a play list stream in a room, we can reuse the code for any or all of our rooms. This method could be placed in our ApplicationAdapter to create the play list.

```
private void createPlayList(IScope room) {
  //get a list of all the currently publishing streams
  List<String> streamNames = getBroadcastStreamNames(room);
  //look for our playlist stream in the list
  if (!streamNames.contains("myplaylist")) {
    IServerStream serverStream =
StreamUtils.createServerStream(room, "myplaylist");
    SimplePlayItem item = new SimplePlayItem();
    item.setName("stream1");
    SimplePlayItem item2 = new SimplePlayItem();
    item2.setName("stream2");
    serverStream.addItem(item);
    serverStream.addItem(item2);
    serverStream.start();
    serverStream.setRepeat(true);
  }
  streamNames.clear();
}
```

If we call our createPlayList method from the application scope, “myapp” in this example we could play the stream on the client-side with this method.

```
ns.play(“myplaylist”);
```

Since we set repeat to true, the videos will loop over and over until we stop the stream or restart the server. The play items specified in the example would exist within our application at the following locations on the hard drive.

```
C:\red5\webapps\myapp\streams\stream1.flv
C:\red5\webapps\myapp\streams\stream2.flv
```

Subscribing to a stream

Subscribing to a stream consists of the act of consuming or playing a stream. While there are a multitude of options we can configure when playing a stream, most of the defaults

are good enough for generalized usage. We will again be utilizing the NetStream class to accomplish our tasks, but in this section we will be a streaming media consumer instead of a producer of the media.

To publish a stream we had to attach our audio and video sources to a NetStream. For subscription it is a bit simpler in that we need only attach the NetStream to a Video object and then place that object in the display list. As with much of the stuff in Flash, this is not the whole story because we can certainly do a lot more with our incoming data than we will cover here such as volume control.

```
var video:Video = new Video();
video.attachVideo(ns);
ns.play("mystream");
addChild(video);
```

In the block of code, we have created a Video object and attached our NetStream to it. We then requested playback of our stream and added the Video object to the display list, which will allow it to be rendered in the Flash Player. Now that our stream is playing, we have additional functions available on the NetStream to control the streamed data.

- pause – Pauses playback until we indicate that we want playback to proceed.
- resume – This causes the pause to be canceled and resumes playback.
- togglePause – A short-cut method to switch between pause or resume.
- seek – Moves the play head to the key frame closest to the specified time. The time value supplied to seek is offset from the start of the stream.
- receiveAudio – Indicates whether or not we will to play the audio data in the stream. Setting this to false, basically mutes the audio.
- receiveVideo – Indicates whether or not we will render the video images contained in the stream.
- send – Sends an AMF data object to all the stream subscribers. This is intended to be used only by a publisher, but in Red5 it has some functionality available to subscribers.
- close – Closes all the communication over the NetStream, effectively stopping playback.

One last thing to consider when attempting to obtain the best experience for our subscribers involves tweaking two properties of the NetStream.

- bufferLength – The amount in seconds of data that we will buffer for playback. This value should be higher for slow connections and lower for

consumers with more available bandwidth. A good starting value is around 5 seconds.

- bufferTime – The time in seconds to delay handling of buffered stream data. For high bandwidth networks playing back live streams, a value of 0.1 is often possible and provides outstanding responsiveness. In most environments this should be set at 1 to 2 seconds.

Stream listeners

Stream listeners provide low level access to a stream and its individual data packets via notifications. Having this level of access to a streams data packets provides a means to manipulate the data in any way we can imagine, anything from creating image snapshots of the live stream to transcoding the audio into another format. The only limitation is that we have a codec that is capable of decoding the streams raw byte data.

To create a stream listener we need only implement one interface and one method, quite easy when compared to other media server solutions. The following listener simply prints the type of packet received to the console.

```
public class MyStreamListener implements IStreamListener {
    public void packetReceived(IBroadcastStream stream,
IStreamPacket packet) {
        byte dataType = packet.getDataType();
        if (dataType == Constants.TYPE_AUDIO_DATA) {
            System.out.println("Audio packet received");
        } else if (dataType == Constants.TYPE_VIDEO_DATA) {
            System.out.println("Video packet received");
        }
    }
}
```

While this code example may not seem useful, it would help during debugging to inform us that audio and video data is being streamed. For our class to receive the packet notifications on a stream, it is advised that a listener be added when the streamBroadcastStart method of our application is called as shown below.

```
@Override public void streamBroadcastStart(IBroadcastStream
stream) {
    //adds a listener to the stream that just started
    stream.addStreamListener(new MyStreamListener());
}
```

A best practice in a standard application would be to keep track of our listeners and then remove them when the broadcasting stream is finished. This is the way it could theoretically be done.

```
private IStreamListener myListener = null;

@Override public void streamBroadcastStart(IBroadcastStream
stream) {
    //create our listener
    myListener = new MyStreamListener();
    //adds a listener to the stream that just started
    stream.addStreamListener(myListener);
}

@Override public void streamBroadcastClose(IBroadcastStream
stream) {
    //removes our listener
    stream.removeStreamListener(myListener);
}
```

To create a more generalized stream listener which would write a single frame snapshot of our streams video to disk. With which we could then process into a PNG or JPG image with another process or tool like Xuggler or FFMpeg. The following example creates an FLV which contains only one frame, a “key frame” of video for our conversion.

```
public void packetReceived(IBroadcastStream stream,
IStreamPacket packet) {
  //we are only interested in video data
  if (packet instanceof VideoData) {
    //cast the packet into a video object
    VideoData videoData = (VideoData) packet;
    //we are only interested in the first keyframe
    if (vd.getFrameType() == VideoData.FrameType.KEYFRAME) {
      //remove this listener since we only want one snapshot
      stream.removeStreamListener(this);
      IoBuffer buffer = packet.getData();
      //keep track of the video data size
      int size = buffer.limit();
       //create a byte array to hold the video bytes
      byte[] blockData = new byte[size];
      //read the video image data into our byte array
      buffer.get(blockData, 0, size);
      try {
        //create file output handler
        FileOutputStream fos = new
FileOutputStream(stream.getPublishedName() + ".flv");
        //write FLV header
```

```
            byte[] hdr =
{'F','L','V',0x01,0x01,0x00,0x00,0x00,0x09,0x00,0x00,0x00,0x0
0,0x09,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
            //write the size of our video image
            hdr[14] = (byte) ((size & 0xff0000) >> 2);
            hdr[15] = (byte) ((size & 0xff00) >> 1);
            hdr[16] = (byte) ((size & 0xff));
            //write the flv header to the file
            fos.write(hdr, 0, hdr.length);
            //write the keyframe
            fos.write(blockData, 0, size);
            fos.close();
          } catch (Exception e) {
            e.printStackTrace();
          }
        }
      }
    }
```

A good point to make about the previous example is that it will work with video data encoded in any of the Flash supported video codecs that are permitted to be stored within an FLV container file. Even more complicated code could possibly decode the video data and manipulate it, such as adding a company watermark.

Supported codecs

A codec, which is short for “coder-decoder” is a computer program that can accept a signal or digital data and encode and/or decode it into a usable format. An example of this would be speaking into your microphone and having an application use an MP3 codec to convert your analog speech into bits and bytes. The recorded voice would then be playable on any device which supports the MP3 codec.

Codecs come is two primary forms named “lossy” and “lossless”. Lossy codecs achieve compression by reducing the quality of the content. It is possible that content could be compressed with a lossy codec at such a high level, that it would no longer resemble the source material; so a balance must be maintained to produce acceptable quality. On the other hand, using a lossless codec we would retain all the source data collected and thus we would lose any of original quality. The main reason to use lossy versus lossless is to reduce the storage requirements for the encoded source material.

All the of the codecs detailed here in may be streamed from Red5 server, but some of them may not be rendered by the Flash Player; this is outside the control of the server. Red5 does not provide encoding or decoding of the media by default, but this does not prevent us from creating such services.

Video

The latest versions of the Flash Player support a decent number of video codecs, some of these codecs are available as open source. Since some of the codecs are available for free, this means that we could theoretically write our own publishing and viewing applications; this has been done by others in the community.

- Jpeg – Very little information is available to the public about the support of this codec for video in Flash. The name associated with the codec identifier eludes to its possible similarity to image format of the same name.
- Screen Video – There are two versions of this codec, originally created by the Macromedia corporation for sharing screen shots. The first version "ScreenVideo" uses a simple BGR image difference routine is available in several programming languages on the Internet, including ActionScript. The "ScreenVideo2" codec was used in the commercial Breeze product and very little is known about its details.
- Sorenson – A variant of the International Telecommunications Union-Telecommunication Standardization Sector (ITU-T) h.263 video codec. This codec is also often referred by the FLV acronym or FLV1 in the ffmpeg tool. Sorenson adds the following features beyond the h.263 standard: difference frames, greater arbitrary dimensions (65535 pixels), unrestricted motion vector turned on by default, and a deblocking flag. This codec is the most commonly used codec in Flash video on the Internet today.
- VP6 – The On2 corporation's proprietary high quality video codec. There are various Flash player supported versions of this low bitrate codec such as VP6, VP6A, VP6E, VP6F, and VP6S. This codec provides higher quality video than Sorenson at a lower bitrate. It was chosen over h.264 by Macromedia for several reasons back in 2005 such as quality at a given bandwidth, portability to non Intel architectures, code size in Flash Player, stability, and overall performance.
- AVC / h.264 – Otherwise know as MPEG-4, this is the standard codec for next generation video on the Internet. Currently the AVC codec provides the best quality video at the lowest bitrate than any other codec supported by Flash Player. This codec is specified in ISO 14496-10 documentation. Flash player supports the following profiles: Base (BP), Main (MP), and High (HiP). One downside to using this superior codec is that the license is difficult for the average user to comprehend. The license terms should be understood before we implement this codec in our applications.

Codec Id	Codec Name	First player

		version support
1	Jpeg	
2	Sorenson Spark	6
3	Screen Video	6
4	VP6, VP6E, VP6F	8
5	VP6A, VP6S	9.0.115.0
6	Screen Video 2	8
7	AVC / h.264	9.0.115.0

Audio

Standard voice audio is usually encoded at 8000 samples per second (8 kHz), music or any other audio will normally be sampled at 11 kHz, 22 kHz, or 44 kHz. CD quality is commonly stated as 44.1 kHz and we all know this is very good quality sound, further proved by the popularity of compact discs. Keep in mind that higher our sample rate, the greater our bandwidth requirements will be times the number of consumers of our stream; this can get very expensive. The selection of audio codec should be made based on our target bandwidth and quality.

- PCM – This codec represents the magnitude of an analog audio waveform with a finite range of values. The audio wave is sampled at fixed regular intervals. Digital telephone systems, compact discs, and computers have been using this format since the 1980's. PCM audio data is by default, not compressed.
- ADPCM – The adaptive delta pulse-code modulation codec is based on differential pulse code modulation (DPCM) which itself is based on PCM. Adaptive in the case of this codec indicates that the quantization size is varied which reduces the bandwidth needed for streaming.
- NellyMoser Asao – Proprietary lossy codec produced by the Nellymoser corporation. This codec is good for speech and general audio content. Flash support three sample rates for this codec, roughly equivalent in quality to telephone, radio, or compact disc.
- MP3 – The very common audio format used in all our audio devices and players. This patented codec was created by the Moving Picture Experts

Group (MPEG) in the early 1990's and was approved as an ISO standard in 1991. This codec handles two channel audio and can sample from 16 kHz to 48 kHz. As with many of the codecs outlined here, this one is also lossy.

- G.711 – Uses logarithmic PCM to sample voice frequencies at 8 kHz. This codec is available in two flavors: μ-law and a-law. Whether we use μ-law or a-law is based on the region of the world where it will be used. In North America and Japan μ-law is used and a-law is used in everywhere else. This codec is reserved for internal use by Adobe at the time of this writing.
- AAC – The advanced audio coding codec is a lossy codec at the heart of the MPEG-4 specifications, it was designed to replace MP3. This codec can handle up to 48 channel audio and sample from 8 kHz to 96 kHz. The output produced by this codec is of similar quality to MP3 but at half of the bitrate, an audio clip encoded in AAC at 64 Kbit/s will sound the same as an MP3 encoded at 128 Kbit/s. Currently the Flash player supports the following AAC profiles: AAC Main, AAC LC or SBR. Many more details about this codec are specified in the ISO 14496-3 specification document. One last thing to note is that Flash player is capable of playing back multichannel AAC audio, but it is re-sampled down to two channels at a sample rate of 44.1 kHz.
- Speex – A patent free lossy codec designed specifically for speech. This codec is available as open source and supports features that are not present in proprietary offerings such as: variable bitrate (VBR), intensity stereo encoding, and integration of different sample rates in the same stream.

Codec Id	Codec Name	First player version support
0	Uncompressed PCM (big endian)	6
1	ADPCM	6
2	MP3	6
3	Uncompressed PCM (little endian)	6
4	Nelly Moser 16kHz	6
5	Nelly Moser 8kHz	6
6	Nelly Moser 5.5 kHz	6
7	G.711 a-Law	Internal use only

8	G.711 μ-Law	Internal use only
9	Reserved by Adobe	NA
10	AAC	9.0.115.0
11	Speex	10
15	MP3 8kHz	

The easiest way available for creating an audio publisher, is to use ActionScript. When selecting the audio codec to use between the two available to ActionScript, we should know the bandwidth requirements for NellyMoser and Speex. Speex bandwidth requirements are controlled with the encodeQuality property of the Microphone class, values from 0 to 10 are allowed and are shown with their bandwidth values below.

Quality	**Bandwidth (kbps)**
0	3.95
1	5.75
2	7.75
3	9.8
4	12.8
5	16.8
6	20.6
7	23.8
8	27.8
9	34.2
10	42.2

NellyMoser bandwidth is controlled via the rate property of the Microphone class. The rate property signifies the sample rate at which the audio is captured. The following table shows the NellyMoser bandwidth requirements for each supported sample rate.

Sample rate	Bandwidth (kbps)
5	11.03
8 (default)	16
11	22.05
22	44.1
44	88.2

There is also an additional rate of 16 kHz which was made available with the release of Flash Player version 10 and Adobe AIR version 1.5.

Supported file types

Red5 server may be used to stream a plethora of media file types right out-of-the-box. The server originally only supported Flash Video (FLV) and MPEG Layer 3 (MP3) files, but has since been extended to stream other popular formats. The current list of media file formats is listed below:

- FLV – Container file for Flash media files, created by Macromedia in 2002 and supported by all versions of Flash Player starting with version 6. Data is stored in this container in a manner similar to an SWF file. Currently this is the only container that Red5 will save to by default.
- MP3 – The de facto standard for audio-only files on the Internet. This format is well known and supported by a great number of applications and hardware devices available today.
- MP4 – Extension for general MPEG-4 files, containing any of the content encoding formats supported by this ISO container type. Audio, video, and text content is stored in the file as “tracks” and has no fixed limit. Normally the content contained within this file would be AVC video with AAC audio, but there are many more types that may be in a file simultaneously.
- M4A – This extension is used to signify an audio-only MPEG-4 file. Consider this similar to MP3 but of higher quality.

- AAC – Container for audio-only files encoded with AAC, essentially the same as an M4A file.
- F4V – The latest container format available for Flash media files which is not based in any way on the FLV container type. This container is based on the MPEG-4 standard and was first supported by Flash Player in version 9 update 3. The older codecs ScreenVideo, Sorenson, VP6, ADPCM, and NellyMoser supported by FLV are not available for this container.
- MOV – The Quicktime container format is the basis of the MPEG-4 file format. This format was created in 1991 by the Apple corporation and is still utilized by their commercial software products. Not all of the codecs used in this container are supported by Flash Player.
- 3GP / 3G2 – File container type used for mobile media created by the 3rd Generation Partnership Project (3GPP). The RFC 3839 contains details for this format. This format is an extension of the MPEG-4 standard format and is targeted for mobile devices. Keep in mind that the majority of 3GP files recorded on mobile phones are not supported by Flash Player due to their use of the AMR codec for audio.

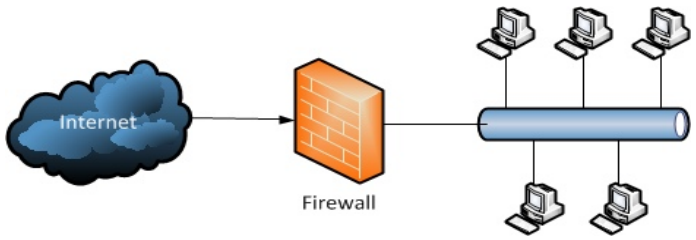
Other file types and formats, may also be streamed with via plug-ins such as Nullsoft Video (NSV).

Extension	Mime type
.flv	video/x-flv
.f4v	video/mp4
.mp4	video/mp4
.mp3	audio/mpeg
.m4a	audio/mp4
.aac	audio/x-aac
.mov	video/quicktime
.3gp	video/3gpp or audio/3gpp
.3g2	video/3gpp2 or audio/3gpp2

Bypassing firewalls

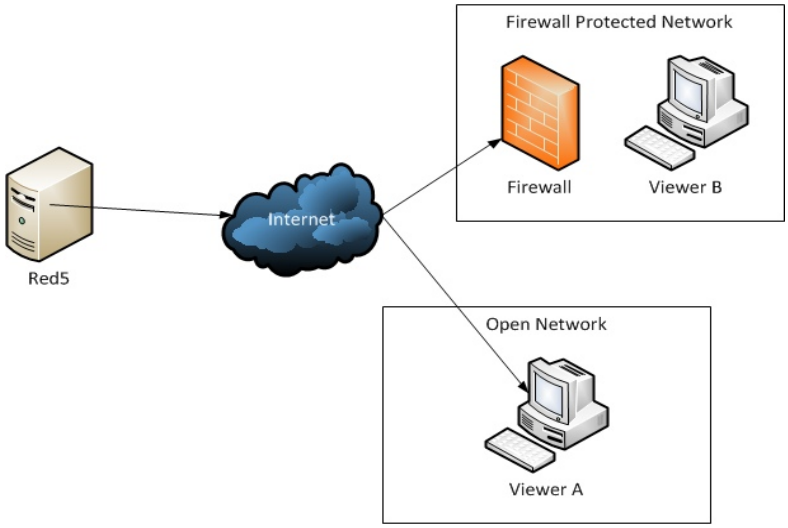
Most of us are familiar with the use of firewalls on the Internet, in the simplest terms they are used to prevent unauthorized network access. There are a lot of different types of firewalls available today, but we will focus on how they affect our Red5 application. The passage of our streams through firewalls is very important for viewership. A great many institutions and corporations block well known applications based on the port that they utilize, but they normally leave the web traffic ports wide open and this is our way in.

The method used by Red5 to traverse these restrictions is to encapsulate RTMP binary data into a text based format, in the most broad sense this is known as tunneling.



RTMPT

Real-time Media Protocol Tunneling or RTMPT for short, is the means with which RTMP data is transmitted via HTTP. Using HTTP as the “transport” for our streams data, we will be able to pass though the majority of firewalls due to its popularity. Since “surfing” the web is extremely common and expected, the HTTP port (80) is usually open by default. When using the RTMPT protocol in Flash Player, the player will act in a similar manner to a browser by making requests and accepting responses. Action Message Format (AMF) data originating from the player will be encapsulated in XML and then be submitted via a POST request. The servers response will also be constructed in XML and sent back to the player. While the content of these requests and responses is technically text data, we may not be able to decode it as we would a document.



This diagram shows a network connection being blocked by a firewall for “Viewer B” and unblocked for “Viewer A”. To initiate a request using RTMPT we simply replace the “rtmp” with “rtmpt” in the protocol position of the resource locator of our NetConnection. Everything else from creating NetStreams, attaching sources, or playing media is exactly the same syntax-wise as when we use the RTMP protocol.

```
nc.connect(“rtmpt://localhost/myapp”);
```

To handle RTMPT requests on the server, our application will need either a dedicated JEE server instance with the RTMPT servlet attached or just the RTMPT servlet configured in our web.xml file.

Summary

In this chapter, we covered a lot of details about streaming media using Red5 server. Using Red5 to stream enables us to do a lot of things with live and pre-recorded media such as conduct video conferences, play slide shows, audio chat, and play music files. We are limited only by our capabilities to write applications and produce content.

We also learned about:

- Naming our streams
- Publishing a live stream

- Providing VOD streaming
- Creating playlists
- Streaming audio files
- Subscribing to streams
- Creating our own stream listeners
- The codecs supported by Red5
- Supported media files
- How to bypass firewalls

In Chapter 5, our focus will be on Shared Objects. We will learn the difference between client-side and server-side shared objects in addition to their persistence options. Shared object listeners will also be covered in detail.