An Introduction to Video Streaming

Introduction

This article provides a comprehensive introduction to Video Streaming. The article will be of interest to developers who work with Symbian C++ video streaming applications. The Symbian OS utilizes a new middleware component called the Multimedia Framework (MMF) to perform all video playback (both local and streamed) operations. The **MMF** is based on the **ECOM** plug-in architecture of Symbian OS.

Basic Information

When streaming video and/or audio, the content of the video/audio data is not located on the user's device. It is placed on a streaming server and the user can access the clip on the server with a URL. The data is accessed on the server and buffered incrementally in the most possible continuous way to achieve a smooth playback. Before being able to access the data and start buffering it, you need to get an active wireless connection. This can be achieved using any type of bearer that allows wireless transfer of data on a mobile device (for example, **GPRS**, **HSCSD**, or **EGPRS**). You also need to define an access point, often referred as **IAP**. Depending on your provider, you might need to define other network parameters, such as the proxy. The URL should start with **rtsp://**, which refers to real-time streaming protocol. You need to have a plug-in loaded on the platform that supports this type of protocol in order to achieve video and/or audio real time streaming. The URL ends with a file name and an extension, which corresponds to the clip on the streaming server. However, certain live streaming URLs do not contain a file name and extension.

There are two types of streaming:

- On-demand streaming of video and/or audio clip
- Live streaming

In case of live streaming, it is possible to find a URL that does not have a file extension, and in these cases the plug-in to be used needs to be explicitly specified, because the **MMF** would not know which plug-in to use for opening the stream. This is explained in detail in **Getting Started**.

The user could be given two options:

- Type the streaming URL directly in any Web browser.
- Open a file that contains the streaming URL. It could be any type of container file, such as .ram or .txt. . ram files are often used for storing the URL.

If the option of showing a file to the user is chosen, you need to extract the streaming URL before passing it to the MMF. This is covered in greater length in **Getting Started**. Also note that it is possible to encounter a .ram file containing several URLs. In this case, you need to have a parser to extract each URL individually and pass only one at a time, in a descriptor format, to the Symbian MMF.

Getting Started

The same interface to the MMF, the `CVideoPlayerUtility` class from Symbian OS, is used both for media streaming and for video playback. This means that similarly as for video playback, you need to create an instance of the class `CVideoPlayerUtility` in order to then perform different streaming operations on a clip, such as the basic ones of opening and streaming, as follows:

```
class CRMMVideoRenderer : public CBase, public MVideoPlayerUtilityObserver, public
MVideoLoadingObserver
  {
private:
  CVideoPlayerUtility* iVideoPlayerUtility;
  TInt iPriority;
  TMdaPriorityPreference iPref;
  RWsSession iWindowSession;
  CWsScreenDevice* iScreenDevice;
  RWindow iDisplayWindowhandle;
  TRect iDisplayWindowRect;
```

```
  TRect iClipWindowRect;
  MVideoObserver* iObserver;
  TDesC& iUrl;
  TInt iIapId;
  TDesC8 iMimeType;
  Tuid iControllerUid;
  };

CEikonEnv* eikonEnv = CEikonEnv::Static();
iWindowSession = eikonEnv->WsSession();
iScreenDevice = eikonEnv->ScreenDevice();
iVideoPlayerUtility = CVideoPlayerUtility::NewL(
*this, // Callback observer
iPriority,
iPref,
iWindowSession,
iScreenDevice,
iDisplayWindowhandle,
iDisplayWindowRect,
iClipWindowRect )
```

As you can see from the previous code example, an instance of `CVideoPlayerUtility` is created in exactly the same way and with the same parameters as for the video playback (see An Introduction to Video Playback), with the addition of another observer specific for streaming, the observer class `MVideoLoadingObserver`.

The observer `MVideoLoadingObserver` handles the asynchronous nature of buffering data during streaming.

In addition to deriving from the observer `MVideoLoadingObserver`, you need to turn on the status reporting for connecting, buffering, and re-buffering if you wish to receive information on the progress of buffering and rebuffering. This is done using the method `RegisterForVideoLoadingNotification` as follows:

```
iVideoPlayerUtility->RegisterForVideoLoadingNotification(*this);
```

Now the status will be sent back by the observer class `MRebufferCallback`. The observer `MRebuferCallback` sends two different callback methods. It returns `MrbcRebufferingStarted` to notify the client that buffering is started and `MrbcRebufferingComplete` to notify the client that the buffering is completed and the playback can carry on. The observer `MVideoPlayerUtilityObserver` is used to inform the client when specific events occur in the process of streaming or retrieving individual frames from a video streaming URL, such as the URL opening has been completed or the end of a streaming file has been reached.

After having created an instance of `CVideoPlayerUtility` and registered to the appropriate observers, you need to set up the display window to be used when streaming. Follow the same steps as for local playback, using the method `SetDisplayWindowL()`.

Once the screen display has been set up, you can use the method `OpenUrlL` to load a video stream. One of the parameters passed is the URL of the clip to be loaded (`iUrl`).

```
TRAPD( err, iVideoPlayerUtility->OpenUrlL(iUrl, iIapId, iMimeType, iControllerUid ) );
if ( err != KErrNone )
  {
  iObserver->ErrorMsg(err);
  }
```

In case of a live stream URL that does not contain an extension, you need to pass the value of the parameter `iControllerUid` in order for the MMF to know which plug-in to use. The parameter `iIapId` corresponds to the access point you want to use to get a GPRS connection.

If you do not provide any specific `iIapId`, the default one will be used, because this parameter is optional and is assigned, by default, the value `KUseDefaultIap`. The parameter `iMimeType` is also optional and its default value is `KNullDesC8`.

Tip: The method `OpenUrlL` ends with **L**, which means this method may leave during its execution. Hence, it is recommended to trap the method using Symbian **TRAPD** and handle the error. You can return the error code in the callback method `ErrorMsg` as shown in the previous example. In this case, should the method `OpenUrlL` leave, the application would not get stranded waiting for the `OpenComplete` callback that would never be received, but instead the application will receive the error returned by the method leaving.

If `OpenUrlL` does not leave during execution, you will receive the callback method `OpenComplete`, which indicates that the URL opening phase has been completed. The `OpenComplete` method returns an error code. If the error code is `KErrNone`, the URL opening phase has been completed successfully and you can now call the `Prepare` method to start the preparing phase.

```
iVideoPlayerUtility->Prepare();
```

Once the callback method `MvpuoPrepareComplete` is received from the observer `MVideoPlayerUtilityObserver`, and if no error is returned (`KErrNone`), you can start playing the streaming link by calling the `Play` method.

```
iVideoPlayerUtility->Play();
```

At this stage, the video buffering will start before any frame is shown to the user. The callback method `MrbcRebufferingStarted` is received to indicate the start of the buffering and when the buffering is completed, the callback method `MrbcRebufferingComplete` is received, signaling the end of buffering for now. The streaming starts at this point and the user can watch the video. When the video is rendered, data is consumed from the buffer, while at the same time the stream is replenishing it with the goal of retaining sufficient buffered video to ensure that the user is unaware of any transient communication slowdown or failure. However, when buffering is not fast enough, the displayed video is paused while more data is buffered, and this results in the application receiving the callback method `MrbcRebufferingStarted` to indicate that buffering is about to start and the method `MrbcRebufferingComplete` to indicate that the buffering is completed and the streaming can resume. The amount of time required for buffering during streaming depends on many factors, such as the clip encoding and thus the bandwidth required by this clip to stream in an optimal way, and the network type (that is, HCSD, GPRS, EGPRS) and capacity.

Some plug-ins allow you to adjust the buffering values. You could decide to have less frequent buffering, which would decrease the quality of display, or vice versa, you could choose to have a better quality of display which implies more frequent buffering.

In case of on-demand streaming, once you have reached the end of clip, you will receive the callback method `OpenComplete`. During streaming, you can pause or stop a stream by using the methods `PauseL` and `Stop`, respectively. These are the same methods that are used in video playback.

```
iVideoPlayerUtility->PauseL();
iVideoPlayerUtility->Stop();
```

To continue the streaming, use the same method `PauseL`, which will resume the streaming to the nearest possible frame. Some buffering might occur at this stage before you can hit an entry frame to resume to. In live streaming there is no such concept as pausing or stopping. Nothing prevents you from calling the methods `PauseL` and `Stop` during a live stream but it will result in a disconnection and when trying to **resume** by calling the method `PauseL`, you will pass through the stages of reconnection. Because the `PauseL` and `Stop` methods are not relevant for live streaming, you can choose to never call the `PauseL` and Stop methods in the case of live streaming. If your plug-in supports it, you can check, before the stream begins and before calling the `Play` method, whether the link is live or on-demand streaming. You can use the metadata field **LiveStream** for this if the plug-in you are using provides it. The metadata field **LiveStream** is a `TBool`, which returns `ETrue` if the URL points to a live stream, otherwise it returns `EFalse`.

Other Features

All the methods provided by the MMF class `CVideoPlayerUtility`, described in An Introduction to Video Playback, can also be used for streaming. However, there are exceptions for live streaming. Some of these differences have already been covered (that is, pause and stop are not applicable in live streaming).

Seek Features

Another important difference is the seek feature, which is, of course, not applicable in live streaming. For on-demand streaming you can seek through the clip in the same way as for local playback, using the method `SetPosition`. When reading the metadata, there is a tag called **seekable**, which should always return `EFalse` for live streaming.

Saving a Real-Time Streaming URL

To save the URL of a streaming link, a container needs to be used. A typical example is the .ram file. When using a .ram file, it is necessary to write a .ram MIME type recognizer. The MIME type for .ram files is **application/x-pn-realaudio**, and the VideoPlayer application will need to register this MIME type. To do this, the following code needs to be added to the VideoPlayer UI resource file:

```
RESOURCE AIF_DATA
  {
  datatype_list =
    {
    DATATYPE { priority = EDataTypePriorityHigh ;
    Type = "application/x-pn-realaudio" ;
    }
  }
```

Video Error Handling

The observer `MVideoPlayerUtilityObserver` allows to return any error that could happen during video playback or video streaming as a result of an unsuccessful operation, such as failure to open the clip or streaming link, or access point defined incorrectly for streaming. The method `ErrorMsg()` of the observer `MVideoPlayerUtilityObserver` returns the error code to the client. The error codes returned can be any Symbian system-wide error code or a Symbian MMF-specific error code. The Symbian MMF-specific error codes are prefixed with `KErrMM`. In many cases it is advisable to kill any instance of the plug-in to clean everything before reattempting the operation that failed. When trying to open a clip or stream, you may receive the following error codes:

- `KErrNotSupported`
- `KErrNotFound`
- `ErrNotAvailable`

In most case, the association of the error code returned for a specific scenario is determined by the plug-in. Examples of cases where the error codes do not depend on the plug-in error handling implementation would be an error code returned when manipulating a file on the Symbian file system, such as saving a clip when there is no space left on the disk (`KErrDisKFull`) or renaming a clip with an already existing name (`KErrAlreadyExist`). In other words, in most cases the plug-in decides which error code to return in a specific situation and the MMF passes this error code unchanged to the UI layer. Therefore, when implementing error handling on the UI layer, it is important to understand which error codes the plug-in returns for each scenario of interest to the UI. For example, the plug-in could return the error code `KErrNotSupported` for the case of a non-supported clip format or for other type of errors, or `KErrNotAvailable` if the audio device is not ready to play the audio track. The error code `KErrMMPartialPlayback` could be returned when the clip can only be partially decoded. This means that either the audio or the video can be decoded but not both. This can happen for several reasons, such as the audio track not being of a supported type but the video is, or the audio is supported but the video's resolution is too high to be decoded, and thus only the audio will be played. Once again the plug-in determines whether to give the user the option to play the clip in a partial mode or not at all, in which case the plug-in would return another error code, for example `KErrNotSupported`. Certain error codes are important to know for streaming, such as:

- `ErrNotEnoughBandwidth`
- `ErrInvalidURL`
- `ErrDisconnected`
- `KErrTimedOut`
- `ErrServerTerminated`

- `KErrCouldNotConnect`

The error codes are heavily coupled with the plug-in implementation of the error handling but also, especially in case of streaming, to guide the user during an error while streaming, and the aim should be to cover the following scenarios:

- There is no access point set up on the device so streaming is not possible. In this case a plug-in could return, for example, the error code `KErrAbort`.
- The user is streaming and the connection is lost. In this case `KErrDisconnected` could be returned.
- There is a network problem and hence the user cannot connect. In this case `KErrCouldNotConnect` or `KErrTimedOut` could be returned.
- The user is trying to stream a clip that requires too much bandwidth for the network the user is connected to. In this scenario `KErrNotEnoughBandwitdh` could be returned.
- `KErrInvalidURL` could be returned for the case of a non-supported URL type, such as a URL for IpV6 because older devices only support IpV4.

The error scenarios and codes described above are the type of errors that should be covered on the UI layer. However, it is once again important to know how the plug-in used handles these errors.

Related Examples

- File:LiveStreamingTV.zip

Related Articles

- Mobile Multimedia
- Archived:MDFDevVideo Record Symbian API
- UI composition in Symbian^3 video player applications (Known Issue)
- WriteDeviceData capability needed to use CMMFDevVideoPlay on Nokia E72 (Known Issue)
- Archived:Additional settings needed to run DevVideoRecord on N96 (Known Issue)
- Video Recording and Video Playing APIs
- Video Streaming Made Simple
- Video