LiveTV
Technical Documentation

Version 1.0

Revision History

Date	Version	Description	Author
2010-01-14	0.01	Initial Draft	Amer, Clement, Darko, Dalibor, Neven, Željko, Nima

Doc. No.:

1. Introduction

1.1 Purpose of this document

The purpose of this document is to provide technical details of software components that this project is made of.

1.2 Document organization

The document is organized as follows:

- Section 1, *Introduction,* describes contents of this guide, used documentation during developing process.
- Section 2, Android recorder, explains technical details about Android recorder application.
- Section 3, *Symbian recorder*, explains technical details about Symbian recorder application.
- Section 4, *Android player*, explains technical details about Android player application.
- Section 5, *Symbian player*, explains technical details about Symbian player application.
- Section 6, *Java player*, explains technical details about Java player application.
- Section 7, *VLC with AMR support*, explains how to add AMR audio codec support to VLC server.
- Section 8, *Streaming server*, explains technical details and configuration of streaming server.
- Section 9, *Studio application*, explains technical details about studio application.

1.3 Intended Audience

The intended audience is:

- Project customer
- Project supervisor
- Project team members
- Users

1.4 Scope

1.5 Definitions and acronyms

1.5.1 Definitions

Keyword	Definitions

1.5.2 Acronyms and abbreviations

Acronym or abbreviation	Definitions
MVC	Model-view-controller
GUI	Graphical user interface
IDE	Intergrated development environment
FFMPEG	Complete, cross-platform solution to record, convert and stream audio and video

RTP	Real-time Transport Protocol

1.6 References

- Installation manual
- User manual

2. Android recorder

2.1 Android operating system

Android is a mobile operating system running on the Linux kernel. It allows developers to write managed code in Java programming language, controlling the device via Java libraries developed by Google.

The central feature of Android operating system is that it allows an application to reuse other application's components, provided those applications permit it. For this to work, the system must be able to start an application process when any part of it are needed, and instantiate the Java classes for that part. Therefore, unlike the applications on most other operating systems, Android applications don't have a single entry point for everything in the application (no main() function, for example). Instead, they are composed of components that the system can instantiate and run as needed. There are four types of those components:

1. Activities - present a visual user interface for one focused endeavor
2. Services - run in the background for an indefinite period of time
3. Broadcast receivers - receive and react to broadcast announcements
4. Content providers - make a specific set of the application's data available to other applications

Recording application provides a user interface for streaming video and audio content to a remote server. Activities are the only component that deals with user interface, and because of that, this application is based on activities. Some basic aspects of the Android activities are described in the following text.

An activity has essentially three states:

- it is active or running when it is in the foreground of the screen
- it is paused if it has lost focus, but is still visible to the user
- it is stopped if it is completely obscured by another activity

If an activity is paused or stopped, the system can drop it from memory either by asking it to finish (calling its finish() method), or simply killing its process. When it is displayed again to the user, it must be completely restarted and restored to its previous state. This means that the application state must be saved every time the application is paused, and restored every time the application is resumed.

As an activity transitions from state to state, it is notified of the change by calls to the following protected methods:

```
void onCreate(Bundle savedInstanceState);

void onStart();

void onRestart();

void onResume();

void onPause();

void onStop();
```

```
void onDestroy();
```

These methods are hooks that can be overridden to do appropriate work when the activity changes states. These methods define a life-cycle of an activity. The following diagram illustrates life-cycle of an activity. The colored ovals are major states the activity can be in. The square rectangles represent the callback methods you can implement to perform operations when the activity transitions between states.

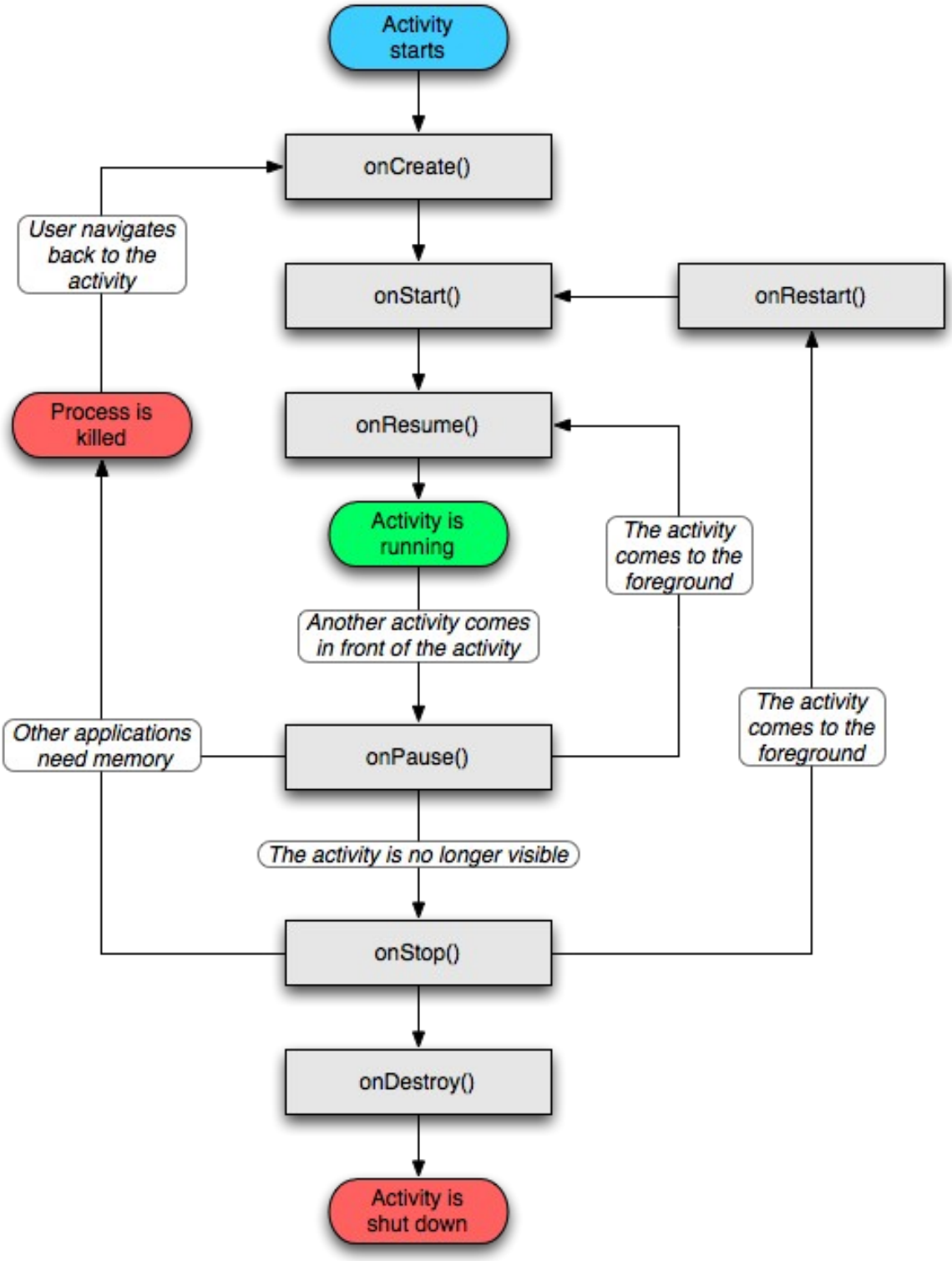

Figure 1: Life-cycle of an Android activity

Besides the source code, Android applications usually contain some additional resources. Most of the resources are defined using XML files. For example, layout of the user interface, animations, constants can all be defined using XML files. Every Android application has a directory for resources (res/) and a directory for assets (assets/). Difference between resources and assets directory is that anything in the resources directory is easily accessible through the R class, which is compiled by Android. Whereas, anything in the assets directory

maintains its raw file format, and is read as a stream of bytes using AssetManager class.

2.2 Streaming

This chapter describes the Android classes used for recording and a way in which streaming was accomplished using these classes. Android SDK defines two classes that can be used for recording audio or video content: AudioRecord and MediaRecorder. AudioRecord class can be used to record raw audio, while MediaRecorder can be used to record audio and video content that can be encoded using a number of different encoders. Neither of those classes directly supports streaming.

AudioRecord can be used to get the raw audio data, and after that this data can be sent over a network using some streaming protocol, like RTP. Problem with the AudioRecord class is that the data must be encoded manually before sending.

MediaRecorder class has the data encoding support, but this class is intended only for storing the recorded data in the phone, or in other words, it isn't intended for streaming recorded data to the remote server. This problem becomes apparent when setting the output location for the recorded data. The output location can be defined in two ways: with a path to the output file or with a file descriptor. Saving data in a file an reading it afterwards in order to stream it over the network would be very inefficient, because accessing memory is a lot faster than accessing a file on the disk. Another problem is that it may not be possible to read from a file, while writing to the same file. This means that the recording would have to be done in stages. Recorder would record in one file for some time, then it would close this file and continue recording in another file so that the first file could be read and streamed over the network. This switch would be repeated in very short periods of time, so that it would look like the recording is continuous. But, in the end, both of these problems would probably cause substantial delay in the streamed content.

Fortunately, there is a better solution to these problems. Android is based on the Linux, and Linux is based on files, almost everything on Linux is a file, including a socket. This allows us to get a file descriptor from a socket and give this to the MediaRecorder as an output file. That way MediaRecorder would write to a socket, and everything that is being recorder would be sent through that socket to some remote server.

Using this solution we can send everything that is recorded to the server. Next problem is that some streaming protocol should be used so the server can understand what it receives. We decided to use RTP protocol for this. But this approach also introduces some new problems. Next paragraphs describe this problems and their solutions in more detail.

First problem was that we needed to get the recorded data before it is sent to the remote server. This can easily be solved by implementing a server on a mobile phone. This way we could send the data to the server on a mobile phone, modify it in any way necessary, and then resend it to the actual remote server. To achieve this, the data was sent through a local socket to the local server socket. A thread read from this local server socket, made required modifications to the data and sent it to the remote server using another socket.

The second problem was that Android records either in 3GPP or MP4 format, neither of which were good for sending using the RTP protocol. These files contain some additional data besides the actual recorded audio and/or video frames. We decided to record in 3GPP file format using H263 video codec and G711 or AMR codecs for audio. G711 codec is implemented in SipDroid project using AudioRecord class and we reused their implementation for recording and encoding recorded data. AMR and H263 were implemented using MediaRecorder class.

3GPP file format consists of atoms, where each atom starts with its size. There are different kinds of atoms in a file, but mdat atom is particularly interesting, because it contains the raw recorded frames, which are necessary for RTP streaming. In order to extract this raw data, all other 3GPP atoms need to be skipped. Once the mdat atom is found, and skipped, the raw data will follow until the recording stops. Additionally, the data obtained in this way cannot be directly sent using RTP. According to the RFC documents which deal with the transfer of the H263 and AMR content using RTP protocol, some additional headers have to be added before the raw video and

audio frames. These headers are described in detail in RFC4629 for H263 codec and in RFC4867 for AMR codec. All data manipulations required before sending this data using RTP protocol is done by the server on the mobile phone. This server extracts the raw recorded frames, adds required headers, forms the RTP packet and sends it to the remote server.

The final problem was that RTP is not implemented in Android, so it should be implemented manually. To avoid doing this, we reused RTP protocol implementation used in SipDroid open source project.

2.3 Recording application

This chapter describes the developed application for Android based mobile phones. Recording application consists of two activities. Main activity is the Recorder activity that controls the recording process and defines the main user interface.

The second activity is the Settings activity that can be used to change different settings of the application. Actual audio and video content recording is done using the classes available in the Android SDK, but SDK doesn't support audio or video streaming. Streaming support is implemented as part of this project. Some classes used for streaming were originally defined in SipDroid open source project, or some of the other open source projects which SipDroid project uses.

2.4 Streaming classes

Streaming is implemented in separate classes so it could easily be reused in similar applications. Every class used for streaming implements StreamSender interface. This interface defines four methods that are used to control the life-cycle of the streaming class. Methods defined in the StreamSender interface are listed here:

```
public abstract void prepare() throws Exception;

public abstract void start();

public abstract void stop();

public abstract void release();
```

Prepare should be called before any other method defined in StreamSender interface, this method is used to initialize the required Android classes that are used for recording. After that, start and stop can be called to start or stop the recording and streaming process. In the end, release should be called to release all resources owned by the object.

Classes VideoStreamSender, AudioStreamSender and RawAudioStreamSender implement the StreamSender interface. All of these classes are abstract, because they also implement Runnable interface and do not define the run method. These classes contain the code required to setup and control the Android classes MediaRecorder and AudioRecord that are used for recording. VideoStreamSender and AudioStreamSender classes use MediaRecorder class to send encoded video or audio content, while the RawAudioStreamSender uses AudioRecord class to get the raw audio bytes. These classes implement Runnable interface because they create a new thread which will execute the code within the run method when the streaming is started or, in other words, when the start method gets called. This thread is stopped when the stop method gets called.

VideoStreamSender, AudioStreamSender and RawAudioStreamSender can be used to easily implement support for streaming using different codecs. Classes that extend those classes should define the code that will process the recorded data and sent it to a remote server. When extending VideoStreamSender or AudioStreamSender, the data is read using the local socket, and when extending RawAudioStreamSender, the data is read from the AudioRecord object. Variables for accessing those objects are defined as protected so they are accessible from the subclasses. VideoStreamSender, AudioStreamSender and RawAudioStreamSender also define getter and setter methods for all of the important parameters that can be used to define audio or video quality, codecs and

information about remote endpoint where the data will be sent.

In the end, three classes are defined that extend abstract classes from previous paragraph and define specific codec streaming logic. These are: RtpH263VideoStreamSender, RtpAMRAudioStreamSender and RtpG711AudioStreamSender. These classes define the code that reads the recorded data, converts it in the format required for streaming and sends properly formatted data to the remote server.

The classes that implement RTP protocol are from the SipDroid open source project. Two classes are used for streaming content using RTP protocol, these are: RtpPacket and RtpSocket. RtpPacket is formed from the data that will be sent and defines the required RTP headers. RtpSocket class is used to send an RtpPacket to a remote server. This class reads the RTP headers and payload data from the RtpPacket and sends it over the network.

The following figure shows the class diagram with the classes described in this chapter.

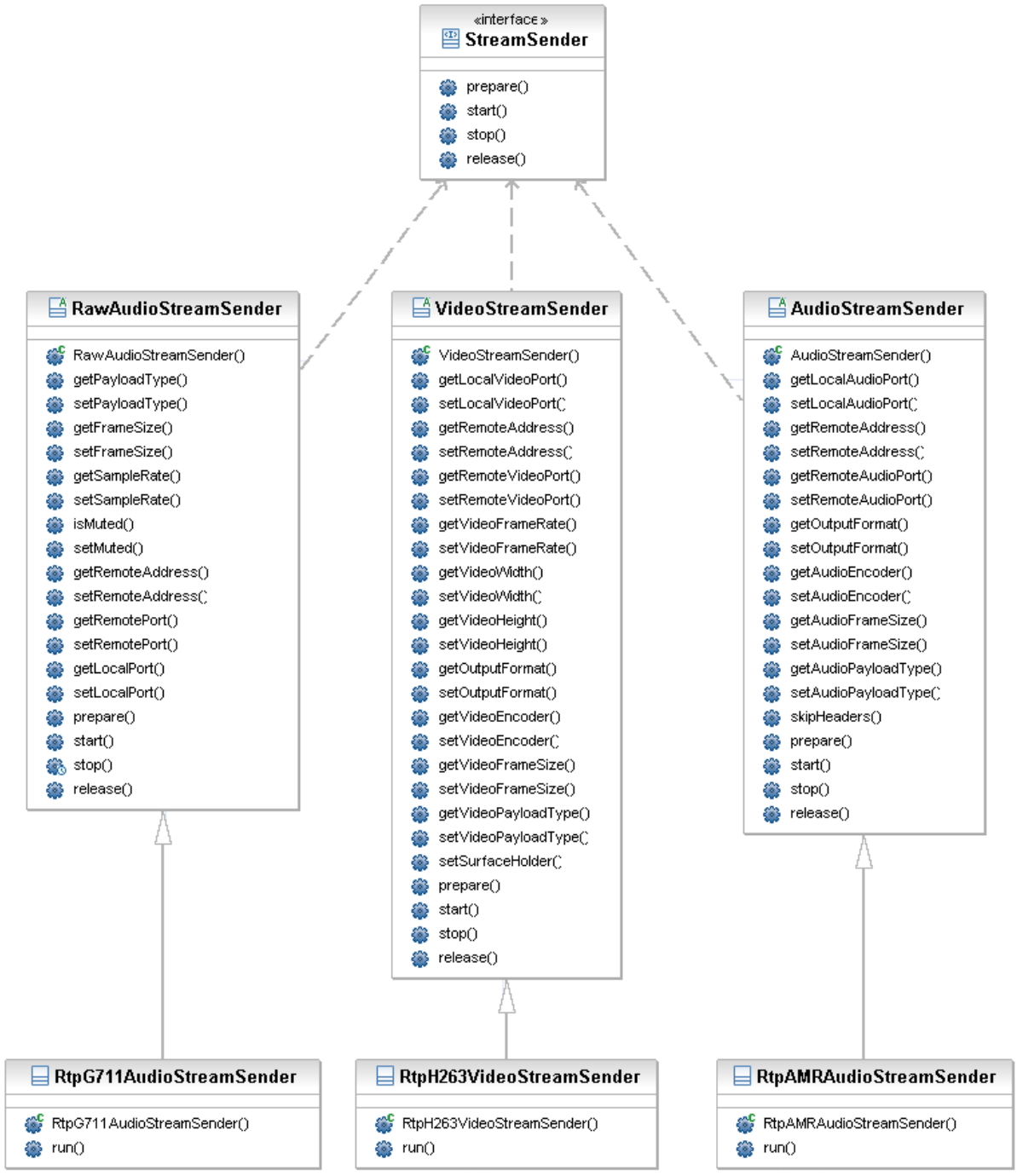

Figure 2 - Class diagram

2.4.1 H263 video

H263 video frames can be of variable length. Because of that, it is necessary to determine where one frames stops and another starts. This can be done by searching for the end of frame marker, which is represented by two consecutive bytes of zeros (00 00). RtpH263VideoStreamSender does this in order to extract the H263 frames from the recorded stream and send them using RTP protocol. Before the video frame is sent, required headers are inserted before the frame as defined in the RFC4629. If the video frame is larger than the MTU size, then it will be split into two or more packets, whose individual sizes will be less than MTU.

2.4.2 G711 audio

G711 encoder used in this project is developed by Jutta Degener and Carsten Bormann from Technische Universitaet Berlin and it is the same encoder that is used in the SipDroid open source project. RtpG711AudioStreamSender reads the data from AudioRecorder object, sends this data to the G711 encoder that encodes this data using G711 codec. This encoded data is afterwards sent using RTP protocol.

2.4.3 AMR audio

AMR audio frames are of constant length, so it is easier to read them than H263 frames. Before the audio data there are some 3GPP headers. RtpAMRAudioStreamSender first skips all of the 3GPP headers, and then just reads the audio frame from the input socket. After the frame is read, required headers are inserted before the frame, and the frame is sent using the RTP protocol. Current implementation of this class has some problems that weren't resolved. When streaming content to the VLC player, VLC prints error: “PTS is out of range”. When manually inspecting sent data using Wireshark trace, it seams that the Android recorder sends the correct data, so it is not currently clear whether this is a bug in the VLC, or in the Android recorder. This should be examined in more detail.

2.5 Recorder activity

The recorder activity is the activity that is displayed to the user when the application is started. This activity defines the graphical user interface of the application and controls the objects responsible for content streaming. This activity first loads user preferences and initializes the objects responsible for streaming based on these preferences. After that, a user can start and stop the streaming via available menu options.

Recorder activity is also responsible for parameters exchange with the server. If this exchange is activated in the settings, it will be done every time when the user attempts to initiate the streaming process. The client application sends username, password and a description of the event that will be recorded to the server, and the server should respond with the audio and video ports to which the corresponding streams should be sent. In case of any errors during this exchange, the client will display the errors messages to the user, and will fall-back to the parameters used in the previous session.

2.6 Settings activity

Settings activity allows the user to view and modify different settings that control various streaming parameters. Android SDK defines classes that can be used for preference handling. Preferences can be defined in two ways: by creating a preferences.xml file or by defining the preferences in the code. We used the XML file approach.

Three types of preferences were used in this application: CheckBoxPreference, EditTextPreference and EditIntegerPreference. Preferences were grouped using categories. CheckBoxPreference and EditTextPreference are built-in Android classes, but the EditIntegerPreference was manually implemented, because there is currently no support for handling integer preferences in the preference screen. This preference was implemented by extending the existing preference, EditTextPreference. Part of the XML file that defines the preferences is listed hereafter:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory android:title="@string/preference_category_streaming">
    <CheckBoxPreference android:key="@string/preference_stream_video"
      android:title="@string/preference_stream_video_title"></CheckBoxPreference>
    <CheckBoxPreference android:key="@string/preference_stream_audio"
      android:title="@string/preference_stream_audio_title"></CheckBoxPreference>
    <CheckBoxPreference android:key="@string/preference_exchange_parameters"
      android:title="@string/preference_exchange_parameters_title">
    </CheckBoxPreference>
  </PreferenceCategory>
                                    . . .
```

3. Symbian recorder

This purpose of this document is to provide the necessary technical documentation regarding the recording application that was developed for Symbian based mobile phones.

The application was developed in Symbian C++ programming languages and therefor follows the basic ANSI C++ together with Symbian C++ modifications. It has specific class connections which are based on the standard Symbian template. Unlike standard C++ programs that start with a normal *main*, Symbian applications have a specific class hierarchy. It is similar to MVC (model-view-controller) in other programming languages but without the model. The structure can be divided into three groups:

1. Classes that communicate with Symbian OS – these classes are part of the standard template and should not be changed. They provide the necessary function to start the application, switch the views and control the memory stack. These classes will not be described into detail.
2. Classes that provide the GUI – these classes are also part of the standard template but are upgraded and changed during the GUI development. Carbide.c++ (IDE for development) provides a graphical variant of GUI development. Developer adds GUI components into the design which are then transformed into source code and added to these classes. Along with these classes will be the UI design file.
3. Classes that provide the needed functionality – these classes are entirely made by the developer. They provide the controller part of the MVC. In this case they provide the recording and streaming part of the application as well as the RTP wrapper class. Without this, the application would only consist from the graphical interface. These classes will, therefor, be described in detail.

The documentation will also be divided in these three groups, with the later being in more detail. The last part will be about the non-class elements of the application (part of the Symbian standard template).

3.1 High level OS classes

These classes are used by the OS itself for launching the application. Although the application it written in C++ it doesn't have the standard main. Instead it has an E32MAIN which is used for application launching. This method then runs the *EikStart::RunApplication()* method with a new application pointer as a parameter. This parameter is of *CRecorderApplication* class which extends the *CAknApplication* class. The later class is the basis for all Symbian applications and all user applications should extend this class.
The first class in this group is, therefor, the *CRecorderApplication* class.

During the creation of an object of this class, a new object of *CRecorderDocument* class is created. This class is the second of the high level OS classes used. The purpose of this class is to create an *AppUI* object which will then be used for container creation. Containers are used for graphical interface display. The only purpose of this second class is, therefor, creation of *AppUI*. The *AppUI* itself is part of the GUI classes.

3.2 GUI classes

The GUI itself is created through the GUI designer which is a part of Carbide.c++. The design made through the GUI designer is then transfered into source code by the IDE itself. The source code from the Designer is integrated in to a total of three classes which are:

- *CRecorderAppUi*
- *CRecorderContainerView*
- *CRecorderContainer*

First is the *CRecorderAppUi* class which was already mentioned. It is used for creating containers and handling various key presses. When a key is pressed, a method in this class is called. This is only for the hardware keys on the device. When a menu key or button is pressed this is transferred to another class which deals with those keypresses. This is standard Symbian template for key handling. Other than creating the containers this class

doesn't have any other purpose. The method which creates the container is *InitializeContainersL()*. An application can have more than one view. In this case, view switching (container switching is implemented in this class. Other than that, it needs to be mentioned that during construction of the AppUi class object, an option is enabled which locks the application in landscape mode only, because the accelerometer in the phone can change that otherwise.

The *CRecorderContainerView* class is the most important class of the entire graphical user interface. During construction, the user interface is created. The creation process isn't in this class. This class only calls a base constructor with a pointer to menu description. The menu description is in another file and is created by the GUI Designer. This class creates the lower containers which are used to display the recorder viewfinder. It is also used to handle menu button presses. For each option in the menu, exist a method which handles the event. The incoming event is switched in the code:

```
switch ( aCommand )
          {      // code to dispatch to the AknView's menu and CBA commands
          case ERecorderContainerViewRecordMenuItemCommand:
                 commandHandled = HandleRecordMenuItemSelectedL( aCommand );
                 break;
          case ERecorderContainerViewSet_IP_AddressMenuItemCommand:
                 commandHandled = HandleSet_IP_AddressMenuItemSelectedL( aCommand );
                 break;
          case ERecorderContainerViewSet_PortsMenuItemCommand:
                 commandHandled = HandleSet_PortsMenuItemSelectedL( aCommand );
                 break;
          case ERecorderContainerViewCredentialsMenuItemCommand:
                 commandHandled = HandleCredentialsMenuItemSelectedL( aCommand );
                 break;
          case ERecorderContainerViewResolutionMenuItemCommand:
                 commandHandled = HandleResolutionMenuItemSelectedL( aCommand );
                 break;
          case ERecorderContainerViewFrame_RateMenuItemCommand:
                 commandHandled = HandleFrame_RateMenuItemSelectedL( aCommand );
                 break;
          default:
                 break;
          }
```

Each event is then handled in its own method. Most of this events are for settings and preferences and are used to change the settings. When a user clicks on a button that is used for that, a query dialogs pops up and asks the user for new settings. This query dialogs are also created by the GUI Designer and are just extended so that they save the new settings in a variable that keeps that setting (for instance, frame rate is saved in a variable in *CRecorderEngine* class).
The *DoActivate()* method of this class is used for creating the lower containers. It exists by default in the standard Symbian template but it was extended so that it also starts the recorder viewfinder (which creates a camera object) and runs the Engine which is the base for recording operations:

```
SetupStatusPaneL();

   if ( iRecorderContainer == NULL )
          {
          iRecorderContainer = CreateContainerL();
          iRecorderContainer->SetMopParent( this );
          AppUi()->AddToStackL( *this, iRecorderContainer );
          }

   iRecorderContainer->Finder()->StartL();
   iEngine = CRecorderEngine::NewL(getRecorderContainer()->Finder()->CameraHandle());
```

The *CRecorderContainer* class is used as a layout for the recorder viewfinder. It is usually used for text and data

displaying but since the application only uses the viewfinder it has been made transparent so that it does not interfere with the viewfinder itself. After the transparency check, the viewfinder is created and applied to the container. The applying is made with *Rect* objects which are passed to the viewfinder. The viewfinder has the *CRecorderContainer* as its *Rect*. Later we can see the code for transparent container:

```
   CWindowGc& gc = SystemGc();
   gc.Clear();

   gc.SetDrawMode( CGraphicsContext::EDrawModeWriteAlpha );
   gc.SetBrushColor( TRgb( 0, 0 ) );
   gc.Clear();
```

These three classes are all added by the IDE when starting a new Symbian project and are used as a template. Other than them, an additional class is added that displays the recorder viewfinder - *CRecorderViewfinder*. There are two ways of displaying a viewfinder to the screen. First is using bitmap images which are projected on the screen. The second is using direct feed from the camera. The first method produces flicker and is not the best possible but all phones support it (earlier versions of Symbian). The second method is much better and gives a slick and smooth viewfinder but has been present only from the 3.2 edition of Symbian OS. Therefor, those two methods have to be interchanged regarding the functionality of the phone. A simple if-else solves this:

```
   if(iFinderDirectL)
          {
          CCoeEnv* coeEnv = CCoeEnv::Static();
          iCamera->StartViewFinderDirectL( coeEnv->WsSession(), *(coeEnv-
>ScreenDevice()), *(DrawableWindow()), rect );
          }
   else
          {
          // Start view finder. On return, finderSize contains the finder size
          iCamera->StartViewFinderBitmapsL(finderSize);

          // Calculate position for finder bitmaps (centered on component)
          // iFinderPosition is the top-left coordinate for bitmap
          iFinderPosition = rect.iTl;
          iFinderPosition.iX += (rect.Size().iWidth - finderSize.iWidth)/2;
          iFinderPosition.iY += (rect.Size().iHeight - finderSize.iHeight)/2;
          }
```

Other than this, the viewfinder class is used for starting the camera. Camera is started in a specific asynchronous way. This process is stared with the *Reserve()* command on the *CCamera* class. After this method is completed, it asynchronously invokes a method in the viewfinder class which then runs the *Prepare()* method. To be able to do this, the viewfinder class must extend the *MCameraObserver* class. Once the camera object is created it can be used in other parts of the application which require the camera (such as recording) via the camera handler. The camera has to be released before that:

```
TInt CRecorderViewfinder::CameraHandle()
   {
   if(iFinderDirectL)
          iCamera->StopViewFinder();

   return iCamera->Handle();
   }
```

3.3 Recording classes

This classes are entirely developer based. In other words, they are not included in the template but have been developed for the recording process. Those classes are:

- *CRecorderEngine*
- *CRecorderSink*
- *CRecorderRTP*

The base class is the *CRecorderEngine* class. It is created during the activation of the viewfinder. The constructor of this class starts with the creation of a *CCMRMediaRecorder* object that is used for recording and is part of the MediaRecorder API of the Symbian OS. A new *CRecorderSink* is also created but that will be explained later. After that the predefined settings are loaded:

```
  iCameraHandle = aCameraHandle;

  iRecorder = CCMRMediaRecorder::NewL();
  iSink = new (ELeave) CRecorderSink;

  TBuf16<16> IPaddress;
  IPaddress = _L16("83.179.12.117");
  destIP.Input(IPaddress);
  destPort = 20000;
  width = 176;
  height = 144;
  frameRate = 15;

  username.Append(_L8("livetv"));
  password.Append(_L8("202020"));

  iState = EStateIdle;
```

There is no other activity in the Engine until the button for start of recording is pressed. The *Record()* method is then called. MediaRecorder API defines a specific way of starting a recording which is implemented in this method. It consists of opening the *CCMRMediaRecorder* object previously defined with the specified audio and video codecs. The MediaRecorder API specifications define a sink which will receive the video and audio frames when they are recorder. We use the *CRecorderSink* object for that:

```
  if (iState < EStateReadyToRecord)
        {
        TMMFMessageDestinationPckg pckg;
        TMMFUidPckg uidPckg( KUidMmfAudioInput );

        iAudioSource = MDataSource::NewSourceL( uidPckg(), KNullDesC8 );
        iRecorder->OpenL(this, iAudioSource, iSink, iCameraHandle, aMimeType,
aAudioType);
        }
```

After that, settings which can be set from the application are enabled (frame rate, resolution, quality...). Unfortunately only the 176x144 resolution is available for h263 format video. It may support higher resolutions but only with a higher quality but this wasn't successfully accomplished. This ends the *OpenL()* method. Before we start the recording we have to start the RTP connection. The RTP object is created in the *CRecorderSink()* class which will be described later. After that we can safely start the recording. Recording is also an asynchronous process which consists of *Prepare()* and *Record()* method calls:

```
  OpenL(aAudioType, aVideoMimeType);

  TCCMRVideoCodingOptions options;
  options.iMinRandomAccessPeriodInSeconds = 10;
  options.iSyncIntervalInPicture = 1;
```

```
   iRecorder->SetVideoCodingOptionsL( options );

   iSink->iRTPEngine->StartRTPConnectionL(destIP.Address(), destPort, username,
password);

   iState = EStatePreparing;
   iRecorder->PrepareL();
```

As it was said, the *CRecorderSink* class is used for receiving the recorded frames of audio and video. These frames have to be encapsulated in RTP and codec headers and then transmitted via RTP class object. The RTP class object is created during the construction of the Sink object:

```
CRecorderSink::CRecorderSink()
: iVideoBufferCounter(0),
  iVideoDataLength(0),
  iAudioBufferCounter(0),
  iAudioDataLength(0)
   {
   iRTPEngine = CRecorderRTP::NewL();
   iRTPEngine->EstablishConnectionL();
   }
```

This also provides a query dialog which gives the user an option to choose which network to use (will be described later). The sole purpose of this class is to receive frames and send them via RTP to the server. This is done in the *WriteBufferL()* method. First we have to find out whether it is an audio or a video frame:

```
   if ( aBuffer->Type() == iVideoCodec )
          {
          …
          }
   else if ( aBuffer->Type() == iAudioCodec )
          {
          …
          }
```

The code for video and audio is different. The video used is h263, and the audio is AMR. Both of these have to be encapsulated according to the RFC documents which deal the transfer of those formats via RTP. These are RFC4629 for h263 and RFC4867 for AMR. The code for encapsulating won't be discussed into detail here. What needs to be mentioned is that there are two levels of encapsulation. First is the RTP header which is used for RTP transfer. It has values such as payload type (103 for video and 108 for audio) and timestamp. More important is the inner header which is used for demuxing the content at the server. These headers are discussed in the RFCs. It is important to keep each RTP packet under the MTU size. Video packets can be bigger that MTU. In that case the video frame has to be divided into two packets and this procedure has to be marked accordingly in the header flags.

Symbian provides the necessary API for RTP transfer. This API is also used in the *CRecorderRTP* class. It is used twice. In other words, separate RTP connections are created for audio and video. Before constructing the RTP connections, an Internet connection has to be created. This connection will be used throughout the project. For video and audio transfer and for communication with the studio:

```
TBool CRecorderRTP::EstablishConnectionL()
   {
   User::LeaveIfError(iSocketServer.Connect(KESockDefaultMessageSlots));
   User::LeaveIfError(iConnection.Open(iSocketServer, KConnectionTypeDefault));

   TCommDbConnPref prefs;
   prefs.SetDialogPreference(ECommDbDialogPrefPrompt);
   prefs.SetDirection(ECommDbConnectionDirectionOutgoing);
   User::LeaveIfError(iConnection.Start(prefs));

   iConnState = ETrue;
```

```
   return ETrue;
   }
```

RTP connection creation is invoked during the start of the recording with the *StartRTPConnectionL()* method. Before we create the connection, the application must communicate with the Studio to receive audio and video ports that will be used for the transfer. After that, two separate RTP connections are created. The connections share the same Internet connection:

```
   // Open RTP session
   aError = iRtpSessionVideo->OpenL(iParam, NULL, &iSocketServer,&iConnection);
   if(aError != KErrNone)
          __LOGNUM1_TOFILE("CRecorderRTP::StartRTPConnectionL() error at iRtpSession-
>OpenL(): %d", aError);

   aError = iRtpSessionAudio->OpenL(iParam, NULL, &iSocketServer,&iConnection);
   if(aError != KErrNone)
          __LOGNUM1_TOFILE("CRecorderRTP::StartRTPConnectionL() error at iRtpSession-
>OpenL(): %d", aError);
```

After this part, standard parameters for each connection are set according to RTP API documentation. The stream ID is used later when transferring the video and audio frames:

```
   iTransmitStreamIdVideo = iRtpSessionVideo->CreateTransmitStreamL(iRtpIdVideo,
aTransmitParams, iSsrcIdVideo);
   iTransmitStreamIdAudio = iRtpSessionAudio->CreateTransmitStreamL(iRtpIdAudio,
aTransmitParams, iSsrcIdAudio);
```

RTP packets are sent in the *SendRtpPacketL()* method. They are sent differently if a video or an audio packet is sent:

```
   if (AV)
          {
          iAudioTimeStamp = aTimeStamp;
          iSendHeader.iTimestamp = iAudioTimeStamp;
          iSendHeader.iPayloadType = 108;

          error = iRtpSessionAudio->SendRtpPacket(iTransmitStreamIdAudio,
iSendHeader, aBuffer);
          }
   else
          {
          iVideoTimeStamp = aTimeStamp;
          iSendHeader.iTimestamp = iVideoTimeStamp;
          iSendHeader.iPayloadType = 103;

          error = iRtpSessionVideo->SendRtpPacket(iTransmitStreamIdVideo,
iSendHeader, aBuffer);
          }
```

Since the application should never receive RTP packets, this method is empty. If an RTP packets arrives, it invokes an error.

3.4 Non-class elements

There are a number of non-class elements in a Symbian C++ project but only the most important will be described here.

The *Recorder.mmp* file is used to define the basic information about the application such as stack and heap size, libraries used... Symbian applications have a thing called *capabilities*. This marks which permissions will the application have on the phone. The permissions range from Internet access to shutter key access. Only some capabilities are available if you aren't a licensed Symbian developer. The recorder application uses the following capabilities:

- NetworkServices
- UserEnvironment

First capability is needed for Internet access and the second for camera and microphone access.

It was already said that the menu creation is done using other files in the project. When we design the UI in the GUI Designer, the IDE creates those files. These are .rss files. The project has three such files: Recorder.rss, Recorder_reg.rss and RecorderContainer.rssi. The menus are in the later file. This part of the file defines the main menu:

```
RESOURCE MENU_PANE r_recorder_container_menu_pane1_menu_pane
   {
   items =
          {
          MENU_ITEM
                 {
                 command = ERecorderContainerViewRecordMenuItemCommand;
                 txt = STR_RecorderContainerView_5;
                 },
          MENU_ITEM
                 {
                 cascade = r_recorder_container_menu_pane2;
                 txt = STR_RecorderContainerView_7;
                 }
          };
   }
```

Other than that, a developer should get to know himself with Symbian development before analyzing this project. This can be done using user guides for Symbian development or various books.

4. Android player

Purpose of this document is to provide the necessary technical documentation regarding the player application that was developed for Android mobile phones.

The application is consist of six classes which are going to be explained in following section

1. Player: main class which is responsible to manage everything.
2. TappableSurfaceView: this class is responsible to handle user requests such as click on the different parts of application.
3. URLHistory: this class is responsible to save different URLs so that user does not need to type all URL paths each time.
4. HistoryItem: this class is used by URLHistory class and it contains one URL item.
5. SocketClient: this class is used for sending UDP message to server and it is implemented because of feedback feature which is not completed in other parts of project.
6. SocketServer: same as socketClient.

4.1 Player

In this class we are controlling everything. On initialization phase user interface components are built in "onCreate" method.

```
public void onCreate(Bundle icicle) {
          super.onCreate(icicle);
          Thread.setDefaultUncaughtExceptionHandler(onBlooey);

          setContentView(R.layout.main);

          surface=(TappableSurfaceView)findViewById(R.id.surface);
          surface.addTapListener(onTap);
          holder=surface.getHolder();
          holder.addCallback(this);
          holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);

          topPanel=findViewById(R.id.top_panel);
          bottomPanel=findViewById(R.id.bottom_panel);

          timeline=(ProgressBar)findViewById(R.id.timeline);

          media=(ImageButton)findViewById(R.id.media);
          media.setOnClickListener(onMedia);

          go=(Button)findViewById(R.id.go);
          go.setOnClickListener(onGo);
          address=(AutoCompleteTextView)findViewById(R.id.address);
          address.addTextChangedListener(addressChangeWatcher);
          history=new
URLHistory(this,android.R.layout.simple_dropdown_item_1line);
          //For setting default addres
          //address.setText("http://www.inventa.com.au/mpeg4%2056kbps176X144PAL.
mpg");
          //address.setText("rtsp://stream.zoovision.com/zootoones/merry_melodie
s_fresh_hare.3gp");


          try {
                File historyFile=new File(getFilesDir(), HISTORY_FILE);

                if (historyFile.exists()) {
```

```
                        BufferedReader in=new BufferedReader(new
InputStreamReader(openFileInput(HISTORY_FILE)));
                        String str;
                        StringBuilder buf=new StringBuilder();

                        while ((str = in.readLine()) != null) {
                               buf.append(str);
                               buf.append("\n");
                        }

                        in.close();
                        history.load(buf.toString());
                }
         }
         catch (Throwable t) {
                Log.e(TAG, "Exception in loading history", t);
                goBlooey(t);
    }

         address.setAdapter(history);
   }
```

There is also method for building feedback menu which is "`onCreateOptionsMenu`".

```
public boolean onCreateOptionsMenu(Menu menu) {
       menu.add(0, 0, 0, "Like!");
       menu.add(0, 1, 0, "Hate!");
       return true;
   }
```

The play method is responsible of playing video using mediaPlayer class.

```
private void playVideo(String url) {
         try {
                media.setEnabled(false);

                if (player==null) {
                       player=new MediaPlayer();
                       player.setScreenOnWhilePlaying(true);
                }
                else {
                       player.stop();
                       player.reset();
                }
                Uri uri=Uri.parse(url);
                player.setDataSource(this,uri);
                player.setDisplay(holder);
                player.setAudioStreamType(AudioManager.STREAM_MUSIC);
                player.setOnPreparedListener(this);
                player.prepareAsync();
                //player.setOnBufferingUpdateListener(this);
                player.setOnCompletionListener(this);
                //address.setText("Yes!");

         }
         catch (Throwable t) {
                Log.e(TAG, "Exception in media prep", t);
                goBlooey(t);
         }
   }
```

There are also some other methods in this class for handling OS operation which are note explained.

4.2 URLHistory

In this class there are two methods for saving URLs and loading saved ones to textbox.

```
void save(Writer out) throws JSONException, IOException {
         JSONStringer json=new JSONStringer().object();

         for (HistoryItem h : spareCopy) {
                h.emit(json);
         }

         out.write(json.endObject().toString());
   }

   void load(String rawJSON) throws JSONException {
         JSONObject json=new JSONObject(rawJSON);

         for (Iterator i=json.keys(); i.hasNext(); ) {
                String key=i.next().toString();
      HistoryItem h=new HistoryItem(key, json.getInt(key));

                spareCopy.add(h);
                add(h);
    }
   }
```

4.3 HistoryItem class

In this class there are two public members: “url” and “count” and two constructors and one method for converting to string format.

```
String url=null;
         int count=1;

         HistoryItem(String url) {
                this.url=url;
         }

         HistoryItem(String url, int count) {
                this.url=url;
                this.count=count;
         }

         public String toString() {
                return(url);
         }

         void emit(JSONStringer json) throws JSONException {
                json.key(url).value(count);
         }
```

4.4 TappableSurfaceView

In this class we just handle touch and tap events.

4.5 SocketClient

In this class we send message over UDP connection. This is implemented in order to send feedback to studio application. In the run method we define one socket and then put information into a packet and send it over the socket. Moreover, they are some lines of code which are writing log in order to test the system.

```
public void run() {
         try {
              // Retrieve the ServerName
              InetAddress serverAddr = InetAddress.getByName(SERVERIP);

              Log.d("UDP", "C: Client Connecting...");
              /* Create new UDP-Socket */
              DatagramSocket socket = new DatagramSocket();

              /* Prepare some data to be sent. */
              byte[] buf = Message.getBytes();

              /* Create UDP-packet with
               * data & destination(url+port) */
              DatagramPacket packet = new DatagramPacket(buf, buf.length,
serverAddr, SERVERPORT);
              Log.d("UDP", "C: Sending: '" + new String(buf) + "'");

              /* Send out the packet */
              socket.send(packet);
              Log.d("UDP", "C: Sent.");
              Log.d("UDP", "C: Client Done.");
              socket.close();
              Log.d("UDP", "C: Client Closed.");
         } catch (Exception e) {
              Log.e("UDP", "C: Client Error", e);
         }
    }
```

4.6 SocketServer

This class can be used in order to receive UDP message from any client. Similar to Socket Client class this class is not getting used in project right now. Sequence of code is very similar to Socket Client class but instead of sending package we are receiving package.

```
public void run() {
         try {
              /* Retrieve the ServerName */
              InetAddress serverAddr = InetAddress.getByName(SERVERIP);

              Log.d("UDP", "S: Server Connecting...");
              /* Create new UDP-Socket */
              DatagramSocket socket = new DatagramSocket(SERVERPORT,
serverAddr);

              /* By magic we know, how much data will be waiting for us */
              byte[] buf = new byte[17];
              /* Prepare a UDP-Packet that can
               * contain the data we want to receive */
              DatagramPacket packet = new DatagramPacket(buf, buf.length);
              Log.d("UDP", "S: Receiving...");

              /* Receive the UDP-Packet */
              socket.receive(packet);
              Log.d("UDP", "S: Received: '" + new String(packet.getData()) +
"'");
              Log.d("UDP", "S: Server Done.");
```

```
            socket.close();
            Log.d("UDP", "C: Server Closed.");
        } catch (Exception e) {
            Log.e("UDP", "S: Server Error", e);
        }
    }
```

5. Symbian player

Purpose of this document is to provide necessary information about symbian player.

The application was developed in Symbian C++ programming languages and follows symbian template. This application is an application which launches Symbians default player which is “Real Player” so this is very simple software and does not need very long documentation.
The only code that is necessary to be explained is the “**HandlePlayMenuItemSelectedL**“ function in “**CPlayerContainerView**” class. Other classes are Symbian’s base template classes which are explained in Symbian recorder documents.

```
TBool CPlayer4ContainerView::HandlePlayMenuItemSelectedL( TInt aCommand )
   {
    aCommand=0;
    RApaLsSession session;
    User::LeaveIfError(session.Connect());
    CleanupClosePushL(session);

    // Wap Browser's constants UId
    //const TInt KWmlBrowserUid = 268458558;
    const TInt KWmlBrowserUid = 0x10008D39;
    TUid uid( TUid::Uid( KWmlBrowserUid ) );;

    TBuf<256> filename;rtsp://
    filename.Copy(_L("rtsp://www.hooraytv.net/livetv.sdp"));
    TFileName aFileName (filename);

    // Runs the default application using the UID.
    TThreadId threadId;

    User::LeaveIfError(session.Connect()); // connect to AppArc server

    session.StartDocument(aFileName, uid, threadId);
    session.Close();

    CleanupStack::PopAndDestroy(); // session

    return ETrue;
    }
```

In this code we provide URL address of our server which is streaming video and then we ask operating system to open this URL. Since it is address of video file automatically it lunches to default player which is Real Player.

6. Java player

6.1 Mobile Media API

Java ME player uses the JSR-135, or Mobile Media API (MMAPI) to play the streamed content. Functionalities of this API can be accessed through the Manager class that can be found in the javax.microedition.media package. This class defined the static method createPlayer that can be used to create a player that will play any of the audio or video formats supported on a mobile phone. This method receives a string that defines the location of the media that should be played. This string should be in the URI syntax. This allows for very easy playback of remote content.

To play any supported remote content, a player needs to be created using createPlayer method with the URI of this content. The createPlayer method returns a Player instance that can be used to control the playback of the media. From this Player instance, a VideoControl and VolumeControl objects can be obtained that can be used to resize the video that is being played, and to control the volume of the audio.

Code required to create a player and start streaming is listed hereafter:

```
Player player = Manager.createPlayer(videoLocation);
player.prefetch();
player.start();
```

First, a new player is created. After that, prefetch is called that will buffer the media that will be played. In the end, the start method is called that will start the playback of the media. The Player object can be in five different states. Change from one state to another can be done by calling various methods defined in the Player class. These states and the methods that can be used to change the player's state are shown in the following figure.

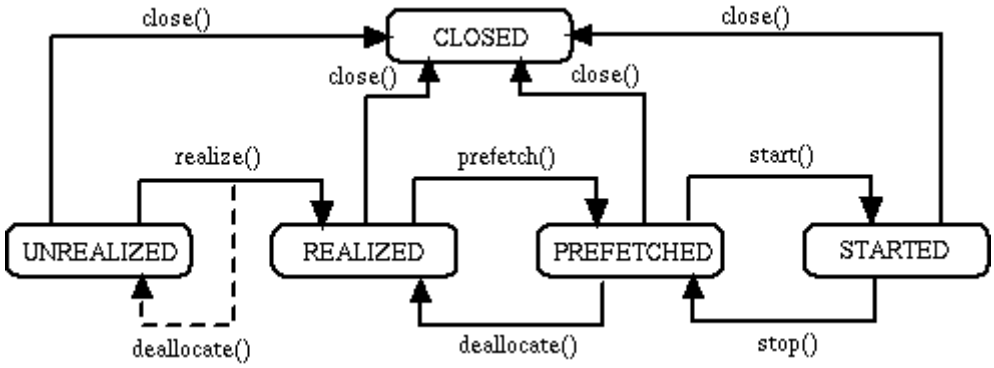

Figure 3: The Player's five states and the state transition methods

States in which an instance of the Player class can be are described in the following text:

- **UNREALIZED** – this is the player's initial state, an UNREALIZED player does not have enough information to acquire all the resources it needs to function.
- **REALIZED** – if the player is in the REALIZED state, it has obtained the information required to acquire the media resources. Realizing a Player can be a resource and time consuming process.
- **PREFETCHED** – once REALIZED, a Player may still need to perform a number of time-consuming tasks, before it is ready to be started. For example, it may need to acquire scarce or exclusive resources, fill buffers with media data, or perform other start-up processing. Calling prefetch on the Player carries out these tasks.
- **STARTED** – once prefetched, a Player can enter the STARTED state by calling the start method. A STARTED Player means the Player is running and processing data. A Player returns to the PREFETCHED state when it stops, because the stop method was invoked, it has reached the end of the media, or its stop time.
- **CLOSED** – calling close on the Player puts it in the CLOSED state. In the CLOSED state, the Player has released most of its resources and must not be used again.

6.2 Implementation

Implementation of the Java ME player consists of four classes: LiveTVPlayer, VideoCanvas, AlertSupport, and FeedbackSupport. These classes will be explained in more detail in this chapter. The following figure shows the class diagram with the listed classes.

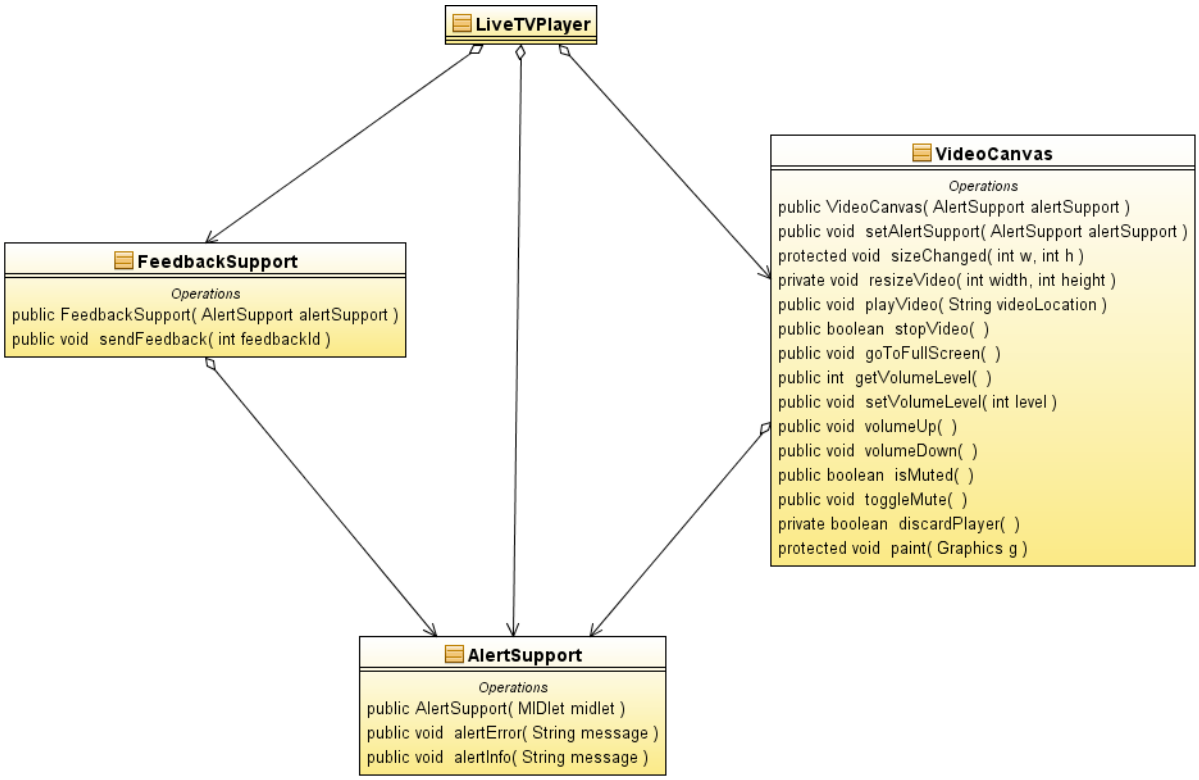

Figure 4: Java ME player classes

LiveTVPlayer is the main class of this application. This class defines the graphical user interface of the application and uses functionalities of the other of the listed classes to implement streaming functionalities. LiveTVPlayer also defines the checkMMSupport that checks for the supported media formats which can be streamed on the phone and prints the results.

VideoCanvas defines all of the functionalities that deal with the streaming process. VideoCanvas class extends the Canvas class that is a GUI component and can be used to draw on it. This enables the VideoCanvas to draw the video that is being played on the underlying Canvas implementation. VideoCanvas defines the methods for resizing the video, muting the audio, increase, and decrease the volume. VideoCanvas also defines the playVideo and stopVideo methods that are used to start and stop the playback of the media content. The playVideo method starts the playback in another thread, because the Player buffers some of the media before the playback starts. This can take some time, so it is done is separate thread so that the application would remain responsive during this time.

AlertSupport class is used for displaying the error or information messages to the user. This class is used in all other classes to allow for centralized alert management. AlertSupport defines two methods: alertError and alertInfo. The first method displays an error message, and the second one displays an informational message to the user.

FeedbackSupport class is used to send the feedback message to the server. This class defines only one method: sendFeedback. This method creates a socket to the remote server, and sends the feedback using this socket. The feedback information is sent by using only one byte. There are currently two constants defined in this class:

```
public static final int I_LIKE_IT_FEEDBACK_ID = 0;
public static final int I_HATE_IT_FEEDBACK_ID = 1;
```

These constants are the values of the bytes that will be sent to the server when the user selects “I like it” or “I hate it” options in the feedback menu. The address of the remote server is currently defined as a constant in the FeedbackSupport class, and is now set to the following value: "socket://localhost:2222". In case the application runs on the emulator, this is the address of the host computer on which the emulator runs. When an application will be used with a real server, this address will have to be replaced with the address of the server that will be used for receiving the feedback.

7. VLC with AMR support

In this project, VLC is used to relay the video and audio stream from the recorder phones to the viewer phones. It is incorporated into the Studio application. For each recorder, a separate VLC server is launched which then receives the stream and relays it to the Darwin Streaming Server.
The main problem of the VLC is the lack of AMR support. AMR is an audio codec designed by the 3GPP group and is used mostly for mobile phone audio streaming. It is very light and requires low bandwidth. Because of licensing issues, VLC lacks support for AMR because FFMPEG which is used for audio/video decoding lacks the support.

To enable the AMR support, VLC recompilation was required. This process is done in three separate parts which come after one another. Those are:

4. libamr-nb and libamr-wb compilation
5. FFMPEG compilation with AMR support
6. VLC compilation with newly compiled FFMPEG

First, the whole compilation process has to be explained before describing the details.
This project uses the Windows version of VLC. In order to recompile the VLC there are two ways of doing it. First is the native way on Windows (using Cygwin or MinGW). Second is recompilation on a Linux machine using i586-mingw32msvc cross compiler (GCC cross compiler, can be found in the repository of various Linux distributions, version used is from Ubuntu 9.10). Official VLC builds are created using cross compilation so this is the best choice.
In order to successfully recompile VLC, additional libraries have to be added into the system. VLC uses these libraries for various decoding, muxing, streaming etc. These libraries can be manually compiled from source or they can be downloaded alredy compiled for a specific version of VLC. The second package is called contribs package. Contribs package already has its own version of FFMPEG which lacks support for AMR. The process of VLC recompilation consists of recompiling FFMPEG with AMR support, replacing the contribs version of FFMPEG with the recompiled one and then recompiling VLC.

The project uses VLC version 0.9.9a. Contribs that are used for this version are 0.9.9.

7.1 Libamr-nb and libamr-wb compilation

In order to integrate AMR into FFMPEG, first the AMR libraries have to be compiled. The source code of the libraries can be downloaded from the following website:

http://www.penguin.cz/~utx/amr

The needed packages are:

- amrnb-7.0.0.2.tar.bz2
- amrwb-7.0.0.3.rat.bz2

After these are downloaded, they have to be configured using the /usr/win32 prefix. We choose this path because we previously extracted the contents of contribs package into this directory. This way, we will add the AMR libraries directly into the contribs libraries (and other contribs files). After configuring and compiling with “make”, the libraries and headers are installed into the contribs directory using:

```
make install-libs
make install-headers
```

The same is done for both libamr-nb and libamr-wb. First is the narrow band and second is the wide band

version of the codec.

7.2 FFMPEG compilation with AMR support

After compiling AMR libraries and headers, FFMPEG has to be recompiled with this newly added AMR support. It is very important to find out the exact version of FFMPEG that is used in the contribs package. After carefully examining the source code and the MINOR and MAJOR version strings it was concluded that the contribs use the r17458 from the FFMPEG SVN repository dated 20090212. This exact version has to be checked out of the repository. The compiling will not go smoothly because some headers in libswscale have multiple definitions at two places. This is a common problem and the necessary patch can be found on the Internet.

After the source code of the FFMPEG has been extracted into a target directory, the compilation process begins. First we have to run the configure script with a great number of parameters:

```
./configure --target-os=mingw32 --enable-memalign-hack
--extra-cflags=-I/usr/win32/include --extra-ldflags=-
L/usr/win32/lib --prefix=/usr/win32 --cross-prefix="i586-
mingw32msvc-" --enable-libfaac --enable-libmp3lame
--enable-postproc --enable-gpl --enable-libamr_nb --enable-
libamr_wb --enable-nonfree --disable-debug -disable-shared
```

After the configure script has finished, we have to run the make command to start the compilation. In case of multi-core CPU, the process can be accelerated using more cores for compilation:

```
make -j 2
```

After the compilation is complete we have a working new version of FFMPEG. Now we have to install the newly created FFMPEG libraries into the contribs. The easiest way of doing this is by simply copying the libavcodec.a and libavformat.a files into the /usr/win32/lib/ directory. This has proven successful. The other way is to run the following command:

```
sudo make install-libs
```

This, however, could end up badly because it will install all FFMPEG libraries and not just the libavcodec which has the AMR support.

7.3 VLC compilation with newly compiled FFMPEG

After we have compiled FFMPEG and copied the libraries, now we have to recompile the VLC against the newly compiled FFMPEG to enable AMR support.
The VLC version that is used is 0.9.9a.
After we have extracted the VLC source code into the target directory we have to run the following configure script:

```
./bootstrap
```

```
PATH=/usr/win32/bin:$PATH
PKG_CONFIG_LIBDIR=/usr/win32/lib/pkgconfig CPPFLAGS="-
I/usr/win32/include -I/usr/win32/include/ebml" LDFLAGS=-
L/usr/win32/lib CC=i586-mingw32msvc-gcc CXX=i586-
mingw32msvc-g++ ./configure --target=i586-minwg32msvc
--host=i586-mingw32msvc --build=i386-linux --disable-gtk
--enable-nls --enable-sdl --with-sdl-config-
path=/usr/win32/bin --enable-ffmpeg --with-ffmpeg-mp3lame
--with-ffmpeg-faac --with-ffmpeg-zlib --enable-faad
--enable-flac --enable-theora --with-wx-config-
path=/usr/win32/bin --with-freetype-config-
path=/usr/win32/bin --with-fribidi-config-
path=/usr/win32/bin --enable-live555 --with-live555-
tree=/usr/win32/live.com --enable-caca --with-caca-config-
path=/usr/win32/bin --with-xml2-config-path=/usr/win32/bin
--with-dvdnav-config-path=/usr/win32/bin --disable-cddax
--disable-vcdx --enable-goom --enable-twolame --enable-
dvdread --enable-debug –disable-zvbi
```

The configure consists of two parts: running bootstrap (VLC internal) and running configure. After running bootstrap we have to manually edit the newly created configure file and add the following lines somwhere (line 1919) in the file:

```
add_extralibs -libamrnb -lm -libamrwb -lm
```

After that we can run the configure script with the parameters above. After the configure is completed we start the compilation:

```
make -j 2
```

After the VLC in successfully compiled we create the Zip package with the following command:

```
make package-win32-zip
```

After that, we have a working copy of VLC with AMR support.

8. Streaming server

The purpose of this document is to provide the necessary technical documentation regarding the streaming server that receives live stream from PC Studio and broadcasts it to mobile and desktop viewers.

The streaming server is a combination of Apple's open-source Darwin Streaming Server (DSS) and VideoLAN (VLC). VLC is integrated in PC Studio and it streams output stream to DSS via RTP streaming. DSS then relays the stream as a broadcast to mobile and desktop viewer clients.

This document is divided into four parts, the VLC component, DSS component, and how they are integrated. Lastly, the Installation and User guide follows.

8.1 VideoLAN (VLC)

VideoLAN is an excellent tool for audio and video playback and streaming. It has a plugin-oriented architecture and in this part of the project, ActiveX plugin exposed by VLC's SDK and is used to relay the output stream coming from PC Studio to the DSS server as an RTP stream.

The ActiveX plugin is integrated in the PC Studio application embedded inside a wrapper class, named DSSRelay. Two functions are implemented in this class to relay either a live stream (coming from Mobile recorder) or an advertisement video. VLC's ActiveX object is set up (hardcoded) to transmit the output RTP stream to the local IP address (assuming that the DSS server is installed and running on the same machine/server). The ports used are 1260 for video and 1262 for audio.

8.2 Darwin Streaming Server (DSS)

Once a desired stream is selected for broadcast in the PC Studio, there needs to be a mechanism to broadcast the chosen stream. For this purpose, Apple's open-source Darwin Streaming Server was selected as it uses standard protocol for broadcasting, i.e. RTSP (real-time streaming protocol). RTSP is widely used to broadcast videos and live media through internet to mobile and desktop viewers.

Another reason to use DSS instead of other streaming servers (such as Real Media Server or Helix streaming server which is the open-source version) is that it is possible to use it in load-balancing mode, i.e. form a cluster of DSS instances which can share the load if the number of viewers increases in future.

DSS has been configured to receive an RTP stream and broadcast it to the mobile and desktop viewers over the RTSP protocol. It runs as a background service and doesn't require user interaction. Only one SDP file is placed in the "Movies" folder of DSS which describes the relay configuration of the stream coming from PC Studio.

8.3 DSS and VLC Integration – The Streaming Server

DSS and VLC are tightly integrated with each other, forming one relay and distribution server, i.e. the Streaming Server in this project. DSS is preconfigured to broadcast the stream coming from VLC ActiveX plugin in PC Studio by the use of an SDP (Session Description Protocol) file, whose contents are given below:

Livetv.sdp

```
v=0

o=- 14913302681658377371 14913302681658377371 IN IP4 lap-
00365
```

```
s=LiveTV

i=N/A

c=IN IP4 127.0.0.1

t=0 0

a=tool:vlc 0.9.9

a=recvonly

a=type:broadcast

a=charset:UTF-8

m=audio 1262 RTP/AVP 96

b=RR:0

a=rtpmap:96 AMR/8000

a=fmtp:96 octet-align=1

m=video 1260 RTP/AVP 97

b=RR:0

a=rtpmap:97 H263-1998/90000
```

This file is placed in the "Movies" folder of DSS (i.e. C:\Program Files\Darwin Streaming Server\Movies) and the access URL of the final broadcasted stream to be used by the desktop and mobile client players is ***"rtsp://www.hooraytv.net/livetv.sdp"*** assuming that the streaming server is running on hooraytv.net host. Furthermore, the DSS and PC Studio should be running on the same server to decrease any intermediate streaming delays and to avoid any introduction of additional bottlenecks in the system.

More details on installation are presented as a guide in the next section of this document, which is then followed by the Usage guide.

9. Studio Application

The purpose of this document is to provide the necessary technical documentation regarding the studio application that receives streams from recorder mobile and send it to the broadcast server (DSS).
The Studio application is divided into four different parts:

- User Interface (AboutBoxLiveTV, StudioApplication, Settings)
- Connection/authentication between the Studio and recorders (ManageConnections, NewRecorderEventArgs, NewRecorderEventHandler, Recorder)
- Video player (Player, VideoPlayer, VideoPlayerManager
- LibVLC wrapper

9.1 User Interface

9.1.1 StudioApplication class

This class is the main class for the User Interface. It was generated automatically by the Visual Studio designer. All the events of it are also managed into it. There is also a function that forces to load the dll of vlc and allow watching faster the first stream played. There are also some functions that manage the new recorder and automatically populate the stream list.

9.1.2 Settings

This class allows to visualize and specify the settings of the system. All of them can be modified.

9.1.3 AboutBoxLiveTV

This class is showing some information about the project, like the member and a description. This data are taken from the AssemblyInfo.cs file of the project.

9.2 Connection/authentication between the Studio and recorders

Every recorder need to be authenticated on the Studio Application before start streaming. The exchange starts from the recorder. It contacts the Studio Application on the IP address and port that are specified in the settings. It sends data, through TCP protocol in the following format “Username|EventPassword|Description”. Then, if the username is unused and the password correct, the studio application send back the video’s port and audio’s port like following “videoport|audioport”. Then, the recorder is automatically added in the list of streams.

9.2.1 ManageConnections

This class allows all the management around new recorders. It calculates the next couple audio and video of port, listen for new recorder, create event when a new one is added.

9.2.2 NewRecorderEventArgs

This class transfers the event and specially the information incoming from the recorder to the listbox of streams.

9.2.3 Recorder

This class manages the TCP pipe between the recorder and the studio application. It's here that the verification of the password's event and the username are done.

9.3 Video Player

The video player use libraries of VLC to play the different streams.

9.3.1 Player

This class is the player indeed. There are functions like play, stop, mute, unmute. There is also a function that allows associating a player that used VLC with a panel into the user interface.

9.3.2 VideoPlayer

This class is the link between the wrapper and the studio application.

9.3.3 VideoPlayerManager

This class allows to manage the set of players running into the application. There are some functions that can modify the comportment of one player and affecting all the other players in contrary. The function UpdateSound allows to mute all players except the one in parameter. UpdateLive allows to modify the border of every player.

9.4 LibVlc wrapper

LibVlc is a library that can be used in .Net project. But a wrapper has to be written in order to use this library. In our project, we used the one that Marx Bitware (support@marxbitware.com). We didn't have to modify it.

List of Tables