

Introduction

The K Desktop Environment – KDE – continues to mature as one of the major forces in the Open Source / Free Software world. As we draw near to the end of the KDE 3 cycle, KDE's contributors continue to evaluate the mounting challenges for the UNIX desktop. Multimedia is and will continue to be an important part of any major desktop suite. As such, in looking towards KDE 4, it is important to examine the current state of multimedia within KDE and our challenges for the next major phase of KDE development.

Here we highlight some of the past accomplishments of KDE in this area, some of the limitations that have been encountered and will present a roadmap for the future of multimedia within KDE.

This is neither the starting point nor the ending point for these explorations. Discussions on the future of multimedia in KDE 4 started in earnest in early 2003 and built up to a meeting of KDE developers at KDE's developers' conference in 2004. aKademy was the first KDE conference to feature a multimedia track with presentations from several of the major projects in the UNIX multimedia landscape. It also allowed for the largest meeting of KDE multimedia developers to date. It was during this time that the future of KDE 4's multimedia architecture was taken from a jumbled collection of ideas and began progressing towards the framework that we present here. However, it should be borne in mind that KDE 4's multimedia framework is still a work in progress. KDE 4 is targeted for release in late 2006. As such the content presented here should be viewed as one of the incremental steps towards the future of KDE multimedia – a roadmap, rather than a schematic.

History of Multimedia in KDE

With the release of KDE 2.0 KDE became the first UNIX desktop to ship with a standard multimedia framework: aRts. As an audio-only framework, aRts provided several things:

- A common framework for audio decoding and playback
- A means of allowing multiple programs to playback sounds simultaneously (which at the time was impossible with most sound cards)
- A framework for building audio synthesis applications

The first of these was more or less accomplished. During the KDE 2 and KDE 3 development cycles a number of multimedia applications that made use of the aRts framework came about. Perhaps more interesting, multimedia elements were integrated into other parts of the desktop including the browser, events notification, games and a variety of other applications.

Naturally many of these sources of sound were concurrent – for example it was and is typical for audio notifications to be used during music playback. The aRts sound server made this possible.

While aRts was originally designed to be a synthesis framework, it largely lacked the critical mass of application support for that goal to be achieved. The extent to which this was used was seen via the sound processing elements – most obviously volume control, but also equalization, reverb and similar – which were used by some media players that made use of aRts as a backend.

All of this naturally represented a step forward in multimedia on the Linux desktop, however over the course of aRts' more than 5 year lifespan, some of its limitations became clear. Some of these limitations were by design: the most obvious of these was the inability to reasonably deal with video's increasingly prevalent role on the Open Source desktop. While there were some patchwork efforts to bolt video decoding onto aRts via a Xine *PlayObject*, video simply was not in the scope of the original aRts design.

aRts also had more tightly coupled sound server and decoding components than are typical with more recent frameworks. As sound hardware, sound drivers and kernel-level sound mixing have improved, it has become less and less necessary to require a sound daemon to be present to make use of the multimedia framework.

In addition to the limitations that came from the original scope of aRts' design, during the second half of the KDE 3 series aRts began to suffer from a lack of active maintenance. As such, there have been a number of annoyances that have gone unfixed for the duration of aRts' lifetime.

During this time the rest of the Open Source multimedia community had not been at a standstill, and it has become clear that when looking to the future of multimedia in KDE that the most reasonable path to take will involve reusing the efforts of other members of the broader multimedia community. A brief discussion of the more visible community efforts follows.

A Survey of Open Source Multimedia Frameworks

In looking towards the future of KDE multimedia, as mentioned previously, it has become clear that the most pragmatic way forward involves reusing the work of other members of the multimedia community. Here we present a very brief survey of the more notable efforts and the relevance of those within KDE's future.

MPlayer

MPlayer is probably the most widely known of the Open Source multimedia efforts. It is a largely static playing application. While some integration has

been possible in playback applications, its design primarily as an application rather than a framework component makes it, despite its maturity, generally not suitable for pervasive use in a variety of desktop applications.

Xine

Together with MPlayer, Xine is one of the most recognized playback applications in the Open Source multimedia world. While again the primary focus is on playback, Xine does maintain a library level abstraction between its interface and playback components. This abstraction, Xine's library, was used during the KDE 3 cycle to enhance the playback abilities of aRts to cover the expanded range of formats supported by Xine. Even so, Xine is still primarily geared towards playback applications. This is a limiting, though not wholly disqualifying factor in considering the role of Xine in future KDE multimedia software.

Helix

Real Networks offers the most recent framework to be thrown into the KDE multimedia mix: Helix. Helix is primarily a playback framework. Developers from Real have in the last year become active in KDE multimedia discussions and the Helix framework may be suitable as one of the playback-only backends for KDEMM.

GStreamer

GStreamer is probably the most widely known of the full-fledged multimedia frameworks and has been designed specifically to meet the needs of desktop multimedia. While originally developed with GNOME in mind and written in C using Glib, it has gained an increasing amount of mindshare in the wider Open Source desktop community in the last few years. GStreamer has specifically cooperated with KDE developers for more than two years and is at this point used by a large number of applications both for GNOME and KDE.

NMM

NMM, or Network Multimedia Middleware, is one of the relative newcomers in the world of Open Source multimedia. In a relatively short while it has grown from being a research project at Saarland University to a reasonably feature-complete network-transparent multimedia framework. Like GStreamer, it was designed from the ground up as a framework for development of a wide range of multimedia applications. The NMM developers have also made a concerted effort in the last year to increase their level of visibility within the KDE and broader Open Source community.

Other Frameworks

KDE is also open to cooperation with other multimedia projects provided that they provide the capabilities necessary to implement the major features of the KDEMM framework and are willing to provide said implementation.

High Level Goals for Multimedia in KDE 4

KDE needs to better integrate into existing multimedia solutions for UNIX. As a developer you should be able to “natively” use multimedia functions in the KDE API. Writing a media player should be both straightforward and completely covered by the KDE APIs. Furthermore, KDE needs to work on many platforms (and with a GPL'ed Qt 4 available on Windows we will likely see a port of KDE to that platform as well) and selecting one media framework to do the task could result in a lock-in situation such as we had with aRts during KDE 2 and KDE 3. KDE needs to be largely independent of the media framework that is used underneath – even if the framework breaks binary compatibility.

The API does not aim to be a general Linux solution to multimedia. It aims to be the solution merely for KDE developers. Also, the framework cannot guarantee the low latencies that are often required for pro-audio applications. As such, you may be able to write a video editor or audio editor, but probably not a synthesizer or sequencer – notably MIDI support is lacking from the current framework design.

As a user of KDE the innerworkings of the framework will be largely transparent; ideally the KDE multimedia framework should simply work the way you expect it. Additionally, KDE multimedia applications should also be able to integrate into a pro-audio environment seamlessly.

Application Development With KDEMM

Simple Video Player Example

So let's take a look at how KDEMM will be used once it's finished. First we'll take a look at how one would write a simple video player – if you remember the Kaboodle application – this would be what we're trying to write.

First we assume that the GUI has been created. As that is standard KDE programming, it's beyond the scope of our current task. On opening a URL a new *MediaObject* is instantiated with the given URL. If the media data provides a video signal we want to show a *VideoWidget*.

```

void SimpleMediaPlayer::openAndPlayURL( const KURL& url )
{
    m_media = new MediaObject( url );
    // handle errors (e.g. if the URL wasn't valid)
    if( m_media->hasVideo() )
    {
        m_video = new VideoWidget( m_mainwindow );
        m_vpath = new VideoPath( m_video );
        m_media->addVideoPath( m_vpath );
    }
    connect( m_media, SIGNAL( finished( MediaObject* ) ), this,
            SLOT( mediaObjectFinished( MediaObject* ) ) );
    m_positionSlider->setMediaObject( m_media );
    m_media->play();
}

```

Let's go through those lines: first a new *MediaObject* is created which is always coupled to a URL. The URL needs to point to media data – if not the constructed *MediaObject* cannot be used and the error handler would need to let us know that an error condition had been hit.

Next, the program needs to know whether the media data includes video data. If so a new widget for showing the video is created and placed into the main window. To actually send the video data from the *MediaObject* to the *VideoWidget* a *VideoPath* is needed (more on that below). The *VideoPath* then outputs to the *VideoWidget* and gets its data from the *MediaObject*.

Then a Qt signal/slot connection is used to execute the *mediaObjectFinished* function when the *MediaObject* has finished playing. This function simply needs to do some cleanup.

```

void SimpleMediaPlayer::mediaObjectFinished( MediaObject* media )
{
    if( m_media != media )
        return;
    delete m_media; m_media = 0;
    delete m_vpath; m_vpath = 0;
    delete m_video; m_video = 0;
}

```

The *m_positionSlider* is a widget that shows the position and can be used to seek in the media. This slider has already been created and put into the GUI. Now all it needs is a *MediaObject* to work with.

In the end the *MediaObject* is instructed to start playing.

You may wonder where the audio data is going. For convenience, the *MediaObject* automatically has an *AudioPath* and an *AudioOutput* (the type of output – OSS, ALSA, Jack, whatever – is determined by the system settings) so that most of the time there's no need for special calls to initialize the audio output. What works for audio does not work for video, though. There's no way

to automatically create a VideoOutput. That's why we initialize it explicitly here.

Crossfading Example

```
MediaPlayer::MediaPlayer()
{
    m_audioPath1 = new AudioPath();
    m_audioPath2 = new AudioPath();
    m_fader1 = new Fader( m_audioPath1 );
    m_fader2 = new Fader( m_audioPath2 );
    setupGUI();
}

void MediaPlayer::play( const KURL& url )
{
    m_media = new MediaObject( url );
    // handle errors (e.g. if the URL wasn't valid)
    m_media->setAboutToFinishTime( 1500 );
    connect( m_media, SIGNAL( aboutToFinish( MediaObject*,
        int ) ), this, SLOT( xfade( MediaObject*, int ) ) );
    connect( m_media, SIGNAL( finished( MediaObject* ) ), this,
        SLOT( mediaObjectFinished( MediaObject* ) ) );
    m_positionSlider->setMediaObject( m_media );
    m_media->addAudioPath( m_audioPath1 );
    m_fader1->fadeIn( 1500 );
    m_media->play();
}

void MediaPlayer::xfade( MediaObject* media, int msec )
{
    swap( m_audioPath1, m_audioPath2 );
    swap( m_fader1, m_fader2 );
    KURL url = takeNextURLFromPlaylist();
    play( url );
    m_fader2->fadeOut( media->remainingTime() );
}
```

Let's look at the additions compared to the first code sample. In the constructor two *AudioPaths* are created (which use the default *AudioOutput* if none is specified). Then an audio effect is created and inserted into the *AudioPath*. Note that the *Fader* effect differs from a volume control in that it does a real continuous volume change while a volume control only changes the scale factor between audio chunks.

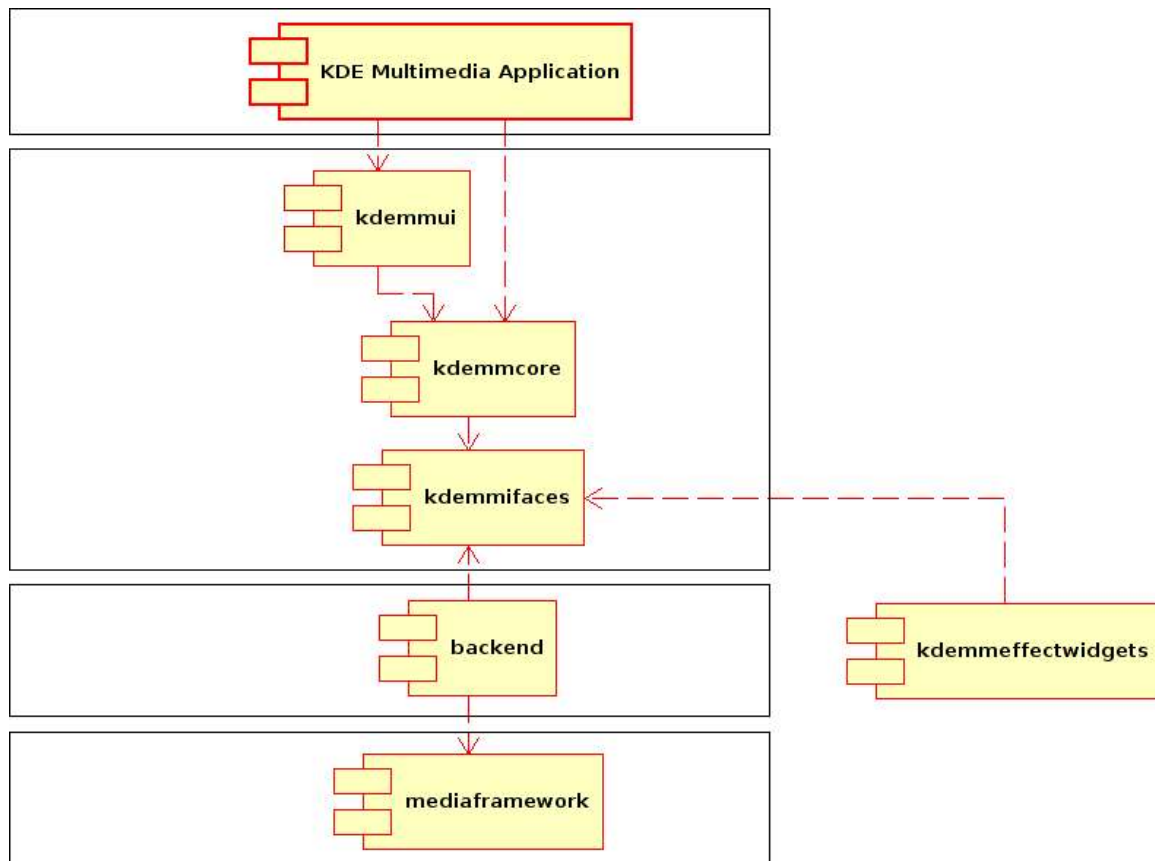
When a new *MediaObject* is created we ask it to signal us 1500 milliseconds before it has finished playing. The signal is then connected to the *xfade()* function which starts the crossfading process. Also, one *AudioPath* is assigned to the *MediaObject* and the corresponding fader effect is set to fade in during these 1500 milliseconds.

When *xfade()* is called it swaps the pointers to the *AudioPaths* and *Faders*, grabs the next URL from the playlist, calls *play()* on that and tells the *Fader* effect of the currently playing *MediaObject* to fade out over 1500 milliseconds.

Architectural Overview

There are basically three parts to KDEMM: *kdemmcore*, *kdemmui* and *kdemmifaces*. The applications will use the core and UI parts while the glue code that binds a media framework to KDEMM will implement the interfaces in *kdemmifaces*. This allows for:

1. KDEMM applications without GUI
2. Exchangeable multimedia frameworks
3. Binary compatible additions / changes to the API



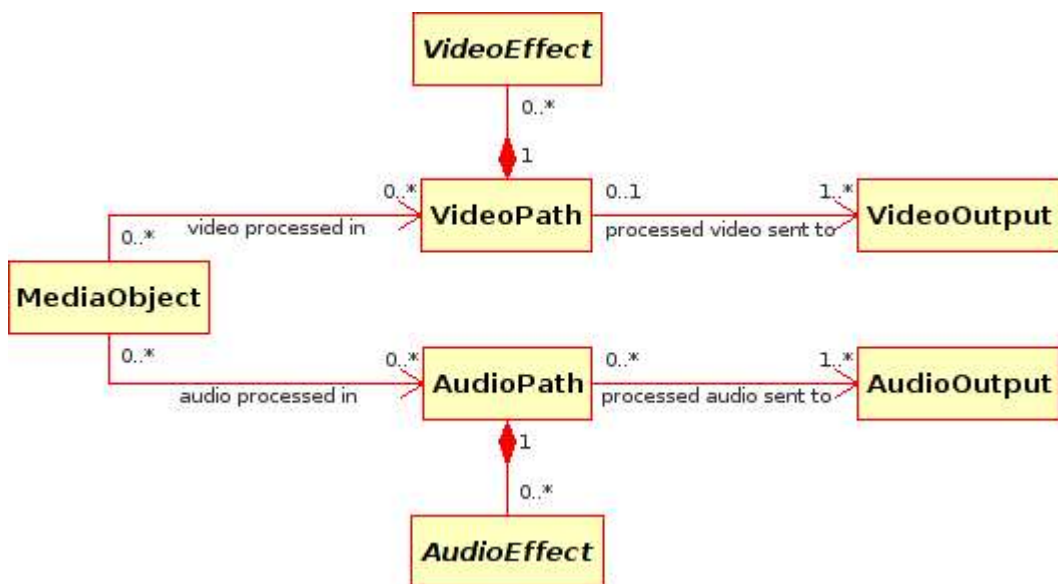
The most interesting part of KDEMM is *kdemmcore*. This is the API that you are going to be using most of the time when writing a media application with KDEMM. The central class is the *MediaObject*. An instance of this object represents a media file / stream that has the standard operations like play, pause, stop, seek and a few signals that indicate the state of the object.

The *MediaObject* is connected to an *AudioPath* and a *VideoPath*. Those paths

are an abstraction of a media data processing path. Every path can hold effects that map to some effect in the underlying media framework. For media frameworks this maps to pipelines or flow graphs of processing nodes / elements. This abstraction makes using effects easier for the user of the API and also makes it easier for backend developers to implement the *kdemmifaces*. The effects in the path either have special interfaces or provide ways of showing a GUI so the user of an application can set the effect parameters. This is a little tricky at times since we neither know the effects that are going to be there nor do we know what media framework we'll be using.

The *AudioPath* finally is connected to an *AudioOutput* which can be a file, soundserver, another computer on the network or just standard OSS or ALSA.

The *VideoPath* is connected to the *VideoOutput* that also has different kinds of implementations. Probably the most important implementation is the *VideoWidget* which provides the standard operations you want to have for a video (ratio, zoom, fullscreen, Xv parameters if applicable).



It is important to keep in mind that the “connections” talked about earlier do not actually form media data streams by themselves. The “connections” are simply an abstraction for the elements that handle the processing in the backends.

Conclusions

KDE 4's multimedia direction will lay the groundwork for the next generation of multimedia and multimedia enabled applications on the KDE desktop. This step will influence a number of those involved in the KDE community.

For users, the new framework will represent a step forward in the

pervasiveness of multimedia on the KDE desktop. While the details of the framework are not of concern to casual users, the benefits of a desktop with richer multimedia capabilities will be visible to all. This also represents a step away from the aRts framework, which has gained some infamy with KDE users in recent years.

For application developers the new framework will make possible wide-spread richer multimedia capabilities, via familiar (i.e. KDE style) and generally easy work with APIs. KDE developers should be able to easily add multimedia components to their existing applications. Additionally we hope to make powerful multimedia capabilities available to the next generation of KDE developers in hopes that this will inspire them to create new applications that go beyond the current breadth of multimedia in KDE.

And of course, this opens up a new line of development for those at the core of KDE's multimedia team and for those that are working in multimedia frameworks in general for the UNIX desktop. While the roadmap is becoming clear – many of the core parts of the KDE 4 multimedia architecture have begun to shape up conceptually – the work of implementing many of these bits is still left to be done. As this step in KDE's multimedia architecture also represents turning towards the community of current multimedia developers on UNIX platforms, this also opens up new lines for cooperation between a variety of groups concerned with the future of multimedia on UNIX.

As these components fall into place over the next year and a half – roughly the amount of time that we expect prior to a KDE 4 release – the final shape of the KDEMM should become more clear. However, this paper, and the accompanying presentation, should serve as a reasonable sketch of things that are on the way.