The KDE Multimedia Architecture

Jeff Tranter
tranter@kde.org

Abstract

This paper and presentation will give an overview of the new multimedia architecture introduced in KDE 2.0. The emphasis will be on aRts, the Analog Real-Time Synthesizer that provides the foundation for multimedia on KDE. The presentation will describe the overall architecture of aRts, give details of the key aRts components such as structures, streams, interfaces, instruments, and the sound server. It will briefly cover the aRts application programming interface, and discuss the current state of multimedia in KDE and some of the future plans.

1 Introduction

This paper presents an overview of the multimedia architecture provided by KDE, with an emphasis on aRts, the Analog Real-Time Synthesizer.

I first give some background on the history of aRts and multimedia on KDE. I then go over the KDE multimedia architecture and key concepts. I will give an example of the code for an aRts effects module, and show how it can be used from `artsbuilder`, the interactive sound development tool for aRts.

I'll briefly review each of the KDE multimedia application programming interfaces (APIs), showing a few coding examples along the way. I'll also discuss strategies for porting existing multimedia applications to aRts.

Finally, I'll review the current status of multimedia on KDE, outline the plans for the future, and provide some references to more information.

The material being presented assumes familiarity with C++ and the basic concepts behind digital audio. A few of the examples assume some knowledge of developing for KDE as well.

While much of the discussion focuses on Linux, keep in mind that KDE runs on several other operating systems, in some cases with a reduced level functionality in the case of multimedia. It should also be noted that aRts itself is portable and can be built and used without KDE.

All of the aRts libraries (which is the majority of the code) are released under the GNU Lesser General Public License. This allows the libraries to be used for non-free or non-open source applications if desired. Some of the programs, like the sound server, are released under the GNU General Public License.

2 Overview and History

KDE is the *K Desktop Environment*, a free, open source graphical desktop for Linux and several other Unix-compatible operating systems.

KDE version 1 had very limited multimedia support. For KDE 2 it was recognized that multimedia had become important on the desktop and a more powerful infrastructure was needed. The developers took a look at what already existed, and came across the aRts project being developed by Stefan Westerfeld.

Since 1997 Stefan had been working on a a real-time, modular system for sound synthesis. It had evolved to include graphical elements built using KDE and ran primarily on Linux. The project at this stage was quite complete, with a CORBA-based protocol, dozens of audio effects modules, a graphical module editing tool, C and C++ APIs, documentation, utilities, and a mailing list and web site with a small group of developers.

Impressed by aRts, the KDE team decided to use it as the basis for the multimedia architecture for KDE 2.0. With this decision the pace of aRts de-

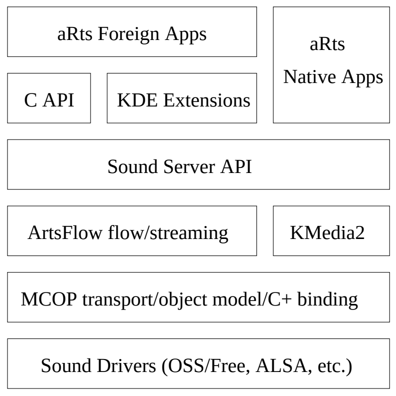

Figure 1: aRts Block Diagram

velopment accelerated. Significant new effort took place, most notably the replacement of the CORBA code with an entirely new subsystem, MCOP, optimized for multimedia. The KDE applications were adapted to use the new sound system, and the sound server was integrated into the control center. Version 0.4 of aRts was included in the KDE 2.0 release.

Work continues on aRts, improving performance and adding new functionality. It should be noted that even though aRts is now a core component of KDE, it can be used without KDE, and is also being used for applications that go beyond traditional multimedia. The project has also attracted some interest from the GNOME team, opening up the possibility that it may someday become the standard multimedia architecture for both of the major desktop environments.

3 aRts Architecture

Figure 1 is a high level block diagram showing the structure of aRts. At the the top of the diagram are the multimedia applications. Those written to use aRts directly ("Native Apps") make use of the sound server API and optionally the lower level APIs such as KMedia2 and artsflow. Applications not specifically written for aRts ("Foreign Apps") can still use the system using KDE extensions such as **knotify** and **kaudioplayer**. Applications written in C can use the C API (aRts and KDE are written in C++). As we will see later, there are also facilities to run existing non-aRts applications directly using the `artsdsp` utility.

The sound server API is the main interface into the aRts services. It is built on top of other layers such as the streaming and flow system and MCOP transport.

At the lowest level are the kernel drivers for the sound hardware, which on Linux can be the OSS/-Free, OSS/4Front (commercial), or ALSA drivers.

In the next section we will describe many of the concepts and capabilities of the system in more detail.

4 aRts in Detail

Before looking at the APIs we need to review the major concepts in aRts, starting with the design goals.

4.1 aRts Philosophy and Goals

aRts (note the capitalization) stands for *Analog Real-Time Synthesizer*, but its architecture can be applied to any streaming media, not just sound. The system was designed to meet a number of goals:

4.1.1 Modularity

It is built on small building blocks called *modules*. One can dynamically build up complex audio processing tools by connecting these modules together. Typical modules include sound waveform generators, filters, audio effects, mixers, and playback of digital audio in different file formats.

4.1.2 Portability

The core aRts code is not tied to any GUI toolkit. It is written in C++ and runs on multiple operating systems. The programming interfaces are defined in a language-independent way using IDL.

4.1.3 High Performance

Components of aRts communicate using a lightweight protocol called MCOP (*Multimedia COmmunication Protocol*). It provides network transparency, low latency, and authentication.

4.1.4 Rich Functionality

The system provides high-level multimedia capabilities, simplifying the development of applications. As new capabilities are added, such as support for new file formats, applications can automatically take advantage of these features.

4.2 Sound Server

Typical multimedia applications that use sound directly open and access the hardware devices such as `/dev/dsp` to record and play back audio. Only one application can open this device at a time, however. This causes a problem in a desktop environment where multiple applications may want to use the sound resources concurrently. For example, the user may have associated sounds with window manager events, a mail client may play a sound to notify of new mail, and the user may want to listen to an MP3 file. The solution to this is to use a centralized sound server which accepts requests from multiple applications, mixes the audio together, and sends it to the sound hardware. In aRts, the sound server is `artsd`, and one and only only copy of `artsd` is normally executing when a user is running KDE.

Using a central sound server also makes it easier to support network transparency, where the sound server can be on a different host from where the application is running. This means, for example, that if you redirect the X11 display for an application, the sound can also be routed to the machine where the display, mouse, and keyboard reside.

The sound server has a number of options which can be adjusted (either from the command line or the KDE sound server control panel) but typically as a user you don't have to be aware of it — it starts up as part of the KDE desktop.

A nice feature of `artsd` is that, on systems that support it, it can run with real-time scheduling priority so that clicks or dropouts can generally be avoided. It also has parameters that can be adjusted to balance the tradeoff between CPU usage, response time, and probability of dropouts.

The sound server will also suspend itself if idle for a configurable period of time or from a command. Since it frees up the sound device when suspended, this is one way to run legacy applications (but generally only as a last resort; we'll discuss this later).

4.3 Modules and Ports

Modules in aRts are small self-contained components that do one thing. Their interfaces are defined in IDL. Modules have input and output ports through which multimedia streams can pass. Like C++ classes, they can have attributes and can implement methods.

aRts comes with several dozen predefined modules for performing functions such as mixing and arithmetic functions, delays, effects (tremolo, reverb, echo), filters, waveform synthesis, and MIDI. It is straightforward to develop new modules. We will look at an example of a module later.

4.4 Structures

Modules can be connected together dynamically to build up *structures*. The `artsbuilder` application lets you graphically build up structures, save them as data files, and send them to the sound server to be executed. A typical structure could implement an effect such as reverb by using modules for delay, filtering, and mixing of audio streams. A screen shot of `artsbuilder` is shown in Figure 4.

4.5 Busses

Busses allow one to dynamically connect modules and structures using named connections. Data streams are sent out *uplinks* and come in on *downlinks*. All signals on a bus are added (mixed) together and are passed as two (stereo) channels. Busses make it easy to dynamically hook effects in and out of the audio streaming system.

4.6 GUI Elements

Some effects require user interaction. A mixer, for example, is not very useful unless a user can adjust the levels. Effects could also be primarily output elements like a level meter or spectrum analyzer. Ideally this could be done by the modules themselves rather than having to hard code the user interface into an application. This is what GUI elements provide — common widgets such as sliders and knobs and an output window to contain them.

Since they are modules, the GUI elements are defined in IDL, and in theory the elements could be implemented using different toolkits (e.g. Qt, Gtk+). Then, with a tool such as `artsbuilder`, you could dynamically create your application. One can envision a mixer application which is customized with the number of mixer controls, level meters, effects, and equalizers that the end user wants in his virtual recording studio. All this could be done by a non-programmer without writing any code.

GUI elements are currently at an experimental stage in aRts.

4.7 MCOP

MCOP is the multimedia communication protocol used by aRts — you can think of it as a lightweight version of CORBA or DCOM optimized for streaming multimedia data. You define interfaces in IDL (*Interface Definition Language*). From the IDL interface definition the MCOP IDL compiler will generate skeleton code (currently just C++ is supported) which provides the base functionality. You derive from the skeleton class to implement your specific functionality.

MCOP takes care of handling network transparency and security. We'll examine this in more detail when we look at an example of an aRts module.

4.8 MIDI

As well as working with synchronous streams of audio samples, aRts has facilities for working with asynchronous MIDI messages.

MIDI stands for Musical Instrument Digital Interface and is a standard for event-based messaging for electronic music devices. A typical MIDI event could indicate that a key on a music keyboard had been pressed, which key it was, and how hard it was hit. The full MIDI specification defines a hardware level protocol for devices (like music keyboards), the software message protocol, and a file format.

aRts lets you send and receive MIDI messages from MIDI devices, as well as software-based MIDI applications. A MIDI Manager handles connecting the different devices. aRts allows you create MIDI instruments, including mapped instruments.

MIDI support in aRts is relatively new and in rapid development, so check the latest version in CVS if you want to work with it.

4.9 Example: an aRts Module

Let's take a quick look at a real aRts module. We'll use a module that already exists so you can look at the code, but writing a new module is quite straightforward.

A common requirement in audio applications is to mix sound sources together. In aRts, an audio stream is just a sequence of floating point numbers representing sound levels sampled at moments in time. To mix two streams together we merely need to arithmetically add them. For our example we want to create a module which accepts two input streams, adds them together, and outputs the result as an outgoing stream.

To create a module we first have to define it in IDL. The module we will look at is called **Synth_ADD** and it can be found in the file `$KDEDIR/kdelibs/-arts/flow/artsflow.idl` (where `$KDEDIR` is the base directory where KDE is installed), the relevant portion of which is shown below in Figure 2

```
/**
 * A module which adds two audio streams
 */
interface Synth_ADD : SynthModule {
  default in audio stream invalue1, invalue2;
  out audio stream outvalue;
};
```

Figure 2: Excerpt from artsflow.idl

At first glance it looks a lot like C++, but IDL in aRts provides a number of conveniences to make development of a module easier. You declare the interface using the `interface` keyword, giving it a name, **Synth_ADD**. We inherit from the existing interface SynthModule, which in this case provides all of the base functionality (attributes and methods) that is needed for any "synthesizer" type module. If you're curious, the **SynthModule** is defined earlier in the same source file. The first line inside the declaration defines two audio input streams and one output audio stream. The `stream` type for audio data is directly understood by aRts. The `default` keyword is a hint to aRts to use these streams by default when connecting the module to other modules.

We also could have defined *attributes* for our module, which are data associated with an instance of the module, and *methods* which are functions associated with the module instance. In our case we have no need for any attributes or methods beyond those that we inherit from the parent **SynthModule**. Interfaces cannot include any procedural code.

Running our IDL file though the `mcopidl` compiler tool produces a header file and a C++ skeleton class. This provides all of the boilerplate code needed in order for our module to work with MCOP. From the input file `artsflow.idl`, the output files will be `artsflow.h` and `artsflow.cc`. The skeleton class is called Synth_ADD_skel.

To implement our module we subclass Synth_ADD_skel. This is where we implement the functionality specific to our module, in this case adding of streams. We do that in the file `synth_add_impl.c`, shown in its entirety (less comments) in Figure 3.

A number of methods are defined by SynthModule. The skeleton class provides default implementations for them. Depending on what your module does you need to override these methods to implement the functionality you need. In our case we need to implement the method `calculateBlock()` which is called by the media streaming system whenever there is data available for the module to process. The method is passed a parameter indicating the number of samples, and the module streams we defined have been mapped into arrays of type float with corresponding names. We simply step through the array, adding the input samples and assigning them to the output array.

```
#include "artsflow.h"
#include "stdsynthmodule.h"

using namespace Arts;

namespace Arts {

class Synth_ADD_impl
  : public Synth_ADD_skel, public StdSynthModule
{
public:
  void calculateBlock(unsigned long samples)
  {
    unsigned long i;
    for(i = 0;i < samples; i++)
      outvalue[i] = invalue1[i] + invalue2[i];
  }
};
REGISTER_IMPLEMENTATION(Synth_ADD_impl);
};
```

Figure 3: Implementation of Synth_ADD Module

The REGISTER_IMPLEMENTATION macro takes care of telling the MCOP object system that we have implemented an interface. Typically we will build our module as a loadable library, and the MCOP dynamic library loading mechanism will be able to find and load our implementation at run time.

To use the module, we can run the `artsbuilder` tool. From the Modules/Synthesis/Arithmetic+Mixing menu item you can see **Arts::Synth_ADD** in the list of modules. For an example of an aRts structure file that uses the **Synth_ADD** module, load the example `example_dtmf1.arts`. This example adds together the signals from two sine wave generators and sends them to the audio output. The tone produced is the same as the one used to signal the digit "1" on a telephone keypad. Figure 4 shows this module being editted from `artsbuilder`.

The example shown later in the section on artsflow illustrates creating a similar connection of modules from application code.

5 aRts APIs

In this section I will give a very quick overview of each of the major aRts APIs, including some ex-

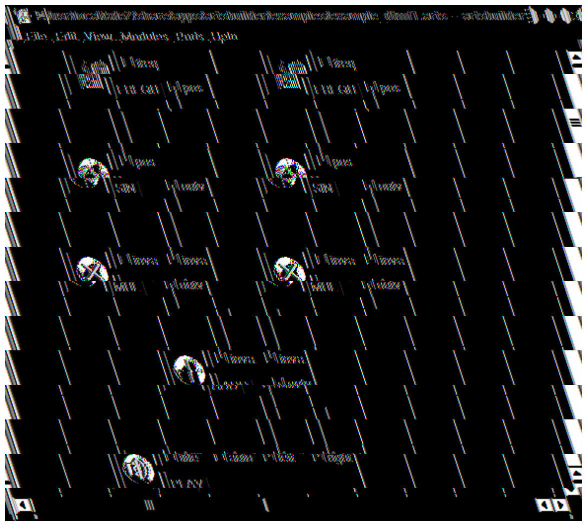

Figure 4: The `artsbuilder` Module Editor

ample programs.

5.1 KNotify

This is part of the more general notification system of KDE. Applications can issue events which can cause actions to occur. In the case of multimedia, you can associate events with sounds. The association is configured using the **System Notifications** panel found in the KDE Control Center under **Look & Feel**. The notifications can be application specific, or common to KDE such as standard dialog boxes. An event can trigger actions to log to a file, play a sound, show a message box, or write to standard error output. Allowing the user to customize the behaviour makes it very flexible, and the library functions provided make it easy for the developer to use.

Events are defined in `eventsrc` files. Standard events common to KDE are defined in the file `$KDEDIR/share/apps/knotify/eventsrc`. Applications-specific events can also be defined, and are stored in an application directory named `$KDEDIR/share/apps/`*`app-name`*`/eventsrc`.

The program in Figure 5 illustrates using the KNotifyClient class to generate some events with associated sounds.

Knotify is very easy to use and configurable by the user, making it suitable for simple applications such

```
#include <kapp.h>
#include <knotifyclient.h>

int main(int argc, char **argv) {
  KApplication *app =
    new KApplication(argc, argv, "Demo");
  KNotifyClient::event("startkde", "Starting.");
  KNotifyClient::event("fatalerror",
    "Fatal error occurred.");
  KNotifyClient::event(KNotifyClient::catastrophe,
    "Something very bad happenned.");
  KNotifyClient::beep("Beep!");
}
```

Figure 5: KnotifyClient Example (knotify.cpp)

```
#include <kapp.h>
#include <kaudioplayer.h>
#include <qpushbutton.h>

int main(int argc, char **argv) {
  KApplication *app =
    new KApplication(argc, argv, "Demo");
  QPushButton button("Honk!", 0);
  // play a sound file (located in standard location)
  KAudioPlayer::play("KDE_Beep_RimShot.wav");
  // create a player for a sound file
  KAudioPlayer player("KDE_Beep_Honk.wav");
  // connect button's signal to player's slot
  cbutton.connect(&button, SIGNAL(clicked()),
    &player, SLOT(play()));
  // make button the main widget and start app
  app->setMainWidget(&button);
  button.show();
  return app->exec();
}
```

Figure 6: KAudioPlayer Example (kaudioplayer.cpp)

as games that only have a requirement to play short sounds.

5.2 KAudioPlayer

This is a simple class that provides audio file playing. The calls return immediately after sending a request to the sound server to play the file in the background. There is no notification when or if the file has been played. In the current implementation, it only indirectly communicates with the aRts sound server, using knotify as a DCOP to MCOP bridge.

You can play a sound file immediately using the static `play()` method. You can also create a **KAudioPlayer** object, and then connect it's `play` slot to another object's signal. Both are illustrated in the code example shown in Figure 6.

The **KAudioPlayer** class is easy to use and does not block the application. It is suitable for applications that only need to play sounds, and the support for signals and slots makes it easy to use with Qt widgets. The disadvantages are that it is somewhat slow, as it uses DCOP, and provides little control or feedback on sound file playing.

5.3 C API

While KDE and aRts are written in C++, it is recognized that many applications are still written in C. The aRts C API was designed to make it easy to write and port plain C applications to the aRts sound server. It provides streaming functionality (sending sample streams to `artsd`), in either blocking or non-blocking modes. For most applications you simply remove the few system calls that deal with your audio device and replace them with the appropriate aRts calls.

Typical usage of the library is as follows:

1. Include the header file `artsc.h`
2. Initialize the API with `arts_init()`
3. Create a stream with `arts_play_stream()`
4. Configure specific parameters with `arts_stream_set()`, if needed
5. Write sampling data to the stream with `arts_write()`
6. Close the stream with `arts_close_stream()`
7. Free the API with `arts_free()`

As a proof of concept Stefan modified the applications mpg123 and Quake to use aRts using the C API.

The example program in Figure 7 shows how to use the C API to play a raw sound file read from standard input.

```
#include <stdio.h>
#include <artsc.h>

int main()
{
  arts_stream_t stream;
  char buffer[8192];
  int bytes;
  int errorcode;

  // initialize
  errorcode = arts_init();
  // create a stream with the appropriate parameters
  stream = arts_play_stream(8000, 8, 1, "demo");
  // read sound data from standard input
  while((bytes = fread(buffer, 1, 8192, stdin)) > 0)
  {
    // write it to the stream
    errorcode = arts_write(stream, buffer, bytes);
  }
  // close stream
  arts_close_stream(stream);
  // free up resources used by library
  arts_free();
  return 0;
}
```

Figure 7: aRts C API Example (artsc.c)

In summary, the C API provides basic sound playback and recording functionality, the ability to control real-time parameters, and is straightforward to use.

5.4 libkmid

Libkmid is the interface for MIDI functions. For simple applications you use the class **KMidSimpleAPI**. The short example in Figure 8 plays a MIDI file.

You can do much more with libkmid, but it requires knowledge of MIDI concepts which are beyond the scope of what can be covered in this short paper.

5.5 aRts Interfaces

These are the native aRts APIs which are not KDE-specific. They are all defined in IDL files (which can be found in `$KDEDIR/include/arts`), from which the C++ header files are generated.

```
#include <iostream>
#include <kapp.h>
#include <libkmid/libkmid.h>

int main(int argc, char **argv) {
  KApplication *app =
    new KApplication(argc, argv, "Demo");
  // initialize the library
  int status = kMidInit();
  // load the MIDI file (adjust path as needed)
  status = kMidLoad("song.mid");
  // start playing the file in the background
  kMidPlay();
  // wait until it finishes
  while (kMidIsPlaying())
    ;
  // release resources
  kMidDestruct();
}
```

Figure 8: libkmid Example (libkmid.cpp)

5.5.1 core.idl

This has basic definitions that form the core of the MCOP functionality, such as the protocol itself, definitions of the objects, the trader, and the flow system.

5.5.2 artsflow.idl

This file contains the definitions for the flow system that is used for connecting audio streams, the definition of **Arts::SynthModule** which is the base for any interfaces that have streams, and a few other useful audio objects.

The example in Figure 10 illustrates connecting together several modules. The program produces two sine waves of different frequencies, mixes them together, scales the result, and sends it to the audio device. The connection of modules is illustrated in figure 9.

Note that in order to keep the example simple, it creates the modules locally and uses the Synth_PLAY module, which writes directly to the sound hardware. A more typical aRts application would use the Synth_AMAN_PLAY module which plays through the sound server, and create the modules in the sound server itself. A slightly longer example which does this is available in the file `artsflow2.cpp`.

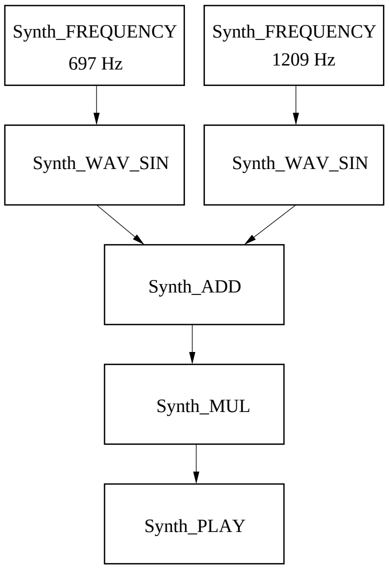

Figure 9: Connection of Modules for artsflow1.cpp Example

5.5.3 kmedia2.idl

This interface defines **Arts::PlayObject**, which can play media of various formats. This is the interface to use if you want to implement a sound file player. It currently supports many formats including WAV, MP3, and Ogg Vorbis.

The example program shown in Figure 11 illustrates using a **PlayObject** to play the sound file named on the command line. It also shows some of the capabilities for querying the object during playback. It would be quite straightforward to extend this into a simple media player program with support for starting, stopping, pausing and seeking.

5.5.4 soundserver.idl

This defines the interface for the system wide sound server implemented by `artsd`. There are currently three interfaces, representing the incremental enhancements to the server: SimpleSoundServer, SoundServer, and SoundServerV2. To provide backwards binary compatibility, interfaces are never changed.

```
#include <arts/artsflow.h>
#include <arts/artsmodules.h>
#include <arts/connect.h>

using namespace Arts;

int main()
{
  Dispatcher dispatcher;
  // create objects
  Synth_FREQUENCY freq1, freq2;
  Synth_WAVE_SIN sin1, sin2;
  Synth_MUL mul;
  Synth_ADD add;
  Synth_PLAY play;
  // setup a 697 Hz sine generator
  // and connect it to the add
  setValue(freq1, "frequency", 697.0);
  connect(freq1, "pos", sin1, "pos");
  connect(sin1, "outvalue", add, "invalue1");
  // setup a 1209 Hz sine generator
  // and connect it to the add
  setValue(freq2, "frequency", 1209.0);
  connect(freq2, "pos", sin2, "pos");
  connect(sin2, "outvalue", add, "invalue2");
  // scale by 0.5 to prevent clipping
  connect(add, "outvalue", mul, "invalue1");
  setValue(mul, "invalue2", 0.5);
  // connect the output to the play module
  connect(mul, "outvalue", play, "invalue_left");
  connect(mul, "outvalue", play, "invalue_right");
  // start all modules
  freq1.start(); freq2.start();
  sin1.start(); sin2.start();
  mul.start(); add.start(); play.start();
  dispatcher.run();
}
```

Figure 10: artsflow Example (artsflow1.cpp)

```
#include <arts/artsflow.h>
#include <arts/kmedia2.h>
#include <arts/connect.h>

using namespace Arts;
using namespace std;

int main(int argc, char **argv)
{
  Dispatcher d;

  // pass a sound file (e.g. mp3) as first argument
  string filename = argv[1];
  // get a reference to the global PlayObject
  // factory
  PlayObjectFactory factory =
    Reference("global:Arts_PlayObjectFactory");
  // Ask factory to create play object
  // for this file
  PlayObject playObject =
    factory.createPlayObject(filename);
  // start playing the file
  playObject.play();
  // wait until play object finishes playing
  do
  {
    sleep(1);
  } while (playObject.state() == posPlaying);
}
```

Figure 11: KMedia2 Example (kmedia2.cpp)

```
#include <arts/soundserver.h>
#include <arts/artsmodules.h>

using namespace Arts;

int main()
{
  Dispatcher dispatcher;
  // get a reference to the server
  SimpleSoundServer server =
    Reference("global:Arts_SoundServer");
  // play a sound file, note that
  // we need the full path
  char dir[PATH_MAX];
  getcwd(dir, PATH_MAX);
  server.play((string)dir + "/" +  "song.wav");
  // create a Reverb effect
  Synth_FREEVERB freeverb;
  freeverb = DynamicCast(
    server.createObject("Arts::Synth_FREEVERB"));
  freeverb.start();
  // insert effect into the output stack
  StereoEffectStack effectstack =
    server.outstack();
  long id = effectstack.insertBottom(freeverb,
    "Freeverb");
  // let it play for a while more
  sleep(10);
  // remove the effect
  effectstack.remove(id);
}
```

Figure 12: Sound Server Example (soundserver.cpp)

The example program in Figure 12 shows some of the sound server features. It illustrates playing a sound file, and inserting an effect into the audio output stream.

5.5.5 artsbuilder.idl

This module defines the basic flow graph functionality, that is, combining simpler objects to create more complex ones. It defines the basic interfaces **Arts::StructureDesc**, **Arts::ModuleDesc** and **Arts::PortDesc** which contain descriptions of a structure, module, and port, respectively.

5.5.6 artsmidi.idl

This module defines basic MIDI functions such as objects that produce MIDI events, the definition of a MIDI event, and the MIDI Manager. This is still somewhat experimental.

5.5.7 artsmodules.idl

This file defines the various filters, oscillators, effects, delays, etc. that can be used for audio signal processing and for building complex instruments and effects.

5.5.8 artsgui.idl

This contains the interfaces for visual objects. At the current time it is still being developed and is subject to change.

5.6 Video

Arhitecturally aRTs can support other streaming media such as video, but this has not yet been implemented. The KDE video player `aktion` is based on the program `xanim`. It does not provide a video API although it can be embedded into other applications using the KDE KPart technology.

6 Porting Applications to KDE

If you have a non-aRts multimedia application, how can you port it to KDE?

A possible solution is to suspend the sound server and then run the application normally. This is not really a solution at all, since if the sound server is busy it cannot be suspended, only killed. Other aRts applications will not be able to use the sound hardware once your application has opened it.

There is a utility called `artsdsp`, included with aRts, which allows most legacy sound applications that talk to the audio devices directly to work properly under aRts. Using the dynamic linker preload mechanism it intercepts system calls to the audio device and redirects them as aRts calls. This works for most applications. Applications written to use the GNOME Enlightenment Sound Daemon (`esd`) will also work in most cases by running `esd` itself under `artsdsp`.

This makes a good short term solution to porting existing applications to KDE. However, it does not allow the application to directly take advantage of all of the power of aRts, such as using modules and multimedia streams other than digital audio. If the application goes beyond simple playing of sound files, it usually makes sense to add native support for aRts to the application.

Using aRts also means that application does not have to do as much work — it can leverage the functions in aRts to handle issues like codecs for different media formats and control of the sound hardware.

When using aRts, you have a number of different APIs to choose from. The decision of which to use depends on a number of factors, including what type of streaming media is used (sound, MIDI, CD audio, etc.), the API features required, and whether it is written in C++. In most cases the choice should be reasonably obvious based on the required features.

For cross-platform portability, applications that need to run on environments other than KDE cannot rely on aRts being present. Using the plug-in paradigm is a good way to support different multimedia environments. Making the plug-in API open and documented (especially for closed source applications) also has the advantage of allowing someone other than the application developer to implement an aRts plug-in.

7 Current Status

The core aRts functionality is considered quite stable and all KDE applications use aRts for sound. Some of the more relevant programs include:

- `noatun` media player
- `artsbuilder` graphical module builder
- `aktion` video player
- `kmid` and `kmidi` MIDI players
- `kmix` mixer
- `kscd` CD player
- and games such as `kpoker` and `ktuberling`

Applications that use **KNotifyClient** (like `konsole`) can play a sound file for ringing the terminal bell.

Some KDE applications that are not yet included in the KDE release (e.g. in **kdenonbeta**) also support aRts, including the `brahms` musical score editor, `kaboodle` embedded player, and `kwave` wave file editor.

Many non-KDE applications are known to work with aRts including the `xmms` MP3 player (with aRts plug-in) and the Real Networks RealPlayer™8.0 (with `artsdsp`; native aRts support is being considered).

8 Future plans

These are some of the enhancements and areas of new development planned for aRts:

- support for multiple sound servers on one machine (like multiple X displays)
- fully threaded aRts

- support for other audio drivers (e.g. SDL, NAS, ESD)
- support for other than 2 sound channels
- enhancements to `artsshell` (e.g. make it a true shell language)
- move to CSL (a new common sound layer intended to be compatible between GNOME and KDE)
- finish implementing support for GUI objects
- support for LADSPA modules (Linux Audio Developer's Simple Plugin API)
- enhancements to `artsbuilder`
- performance optimization
- documentation (manual and `kdoc` source comments)
- gateway between MCOP and DCOP, CORBA, XMLRPC
- C language binding, based on `glib`, with `mcopidl` code generation for C
- support for video streams and codecs

9 Summary and Conclusions

As I see it there are some key factors that have challenged the development of multimedia on Linux:

9.0.1 Lack of hardware device drivers

This is becoming less of an issue now that most hardware vendors are recognizing Linux, especially video cards, but it remains an issue for some leading edge hardware such as the latest sound cards and the more esoteric devices like MP3 players. This is outside the scope of this paper.

9.0.2 Proprietary codecs/file formats

Examples in this category include Apple QuickTime™, RealAudio™, and various Microsoft sound and video file formats. In some cases the issues are not so much technical as legal, such as the intellectual property and patent issues with GIF files, MP3 encoders and DVD players. This also is outside the scope of this paper, but one approach to address this is to encourage vendors to open up their standards, and educate users about the implications of proprietary versus open formats. Users can be given an alternate choice of unencumbered replacements like PNG and Ogg Vorbis.

9.0.3 Lack of standardized libraries

There are few libraries and toolkits for multimedia, and among those there is little agreement on which ones to use. This results in applications which use different and sometimes incompatible libraries, or in many cases the applications have to implement all of the multimedia functionality from the ground up themselves. KDE addresses this with a standardized set of libraries.

9.0.4 Access to shared resources

The best example here is for sound devices. Only one application can use the sound device at one time. A common sound library or sound server can solve this, but all applications have to agree to use it.

The `artsd` sound server addresses this by providing a sound server as part of the desktop architecture. It also holds the potential for providing compatibility with the `esd` sound server in the future, allowing both KDE and GNOME applications to coexist (in fact you can do that today in a less elegant fashion by running `esd` under the `artsdsp` utility).

9.0.5 Rapid pace of change

New file formats such as Ogg Vorbis have the potential to become de facto standards in a short period of time. Applications that don't keep up can rapidly become obsolete. A way of addressing this is to provide a plug-in architecture to allow new codecs to be developed independently of the application. The traditional approach is to support plug-ins in the applications themselves. KDE take this further by providing the plug-ins in the multimedia framework.

In addition, the nature of open source development is such that it encourages a rapid pace of development with quick feedback from users.

9.0.6 Lack of high quality applications

Most of the above issues have acted to limit the number of high quality applications. It is hoped that the new KDE multimedia architecture will make it easier for developers to create killer applications that will put Linux and KDE in the forefront for desktop applications.

10 References

1. *The aRts Handbook* (**http://www.arts-project.org/doc/handbook**)

   Included with KDE, this tries to be the definitive guide for developers. Some sections are still incomplete.

2. *KDE 2.0 Development*, chapter 14.

   This published book has a chapter on multimedia written by Stefan with a lot of good material. As well as in print, it is available on-line at **http://www.andamooka.org** where it is annotated with comments and updates.

3. **http://multimedia.kde.org**

   This is the home page for KDE multimedia.

4. **http://www.arts-project.org**

   This is the home page for the aRts project.

5. **http://sound.condorow.net**

   This is a great site for Linux multimedia applications.

In the KDE CVS source code repository the core functions of aRts are in **kdelibs**. Additional libraries and applications are in **kdemultimedia**. Good API documentation is generated directly from the source comments using the KDE documentation tools.

Most KDE multimedia and aRts development discussions are held on the KDE multimedia and aRts mailing lists. See *The* aRts *Handbook* for details. Occasional meetings are also held on IRC.

This presentation paper, slides for the talk, and expanded versions of the example code listings will be available from my web site at **http://www.pobox.com/~tranter** and at **http://www.ottawalinuxsymposium.org**

11 Acknowledgements

Some of the material in this paper was adapted from *The* aRts *Handbook*, which was written primarily by Stefan Westerfeld and myself. Some of the examples are based on sample programs included with aRts.

12 About The Author

Jeff Tranter has been using Linux since 1992. He is the author of the Linux Sound and CD-ROM HOWTOs and the O'Reilly book *Linux Multimedia Guide*. He has worked in a number of diverse areas of Linux, most recently on the KDE desktop environment focusing on multimedia.