

Phonon and NMM – Multimedia Architectures for the Desktop and Beyond

Marco Lohse, Matthias Kretz, Michael Repplinger, and Bernhard Fuchshumer

16. April 2006

Lizenzbestimmungen

Dieser Beitrag ist unter der Creative Commons Licence <<http://creativecommons.org/licenses/nd-nc/1.0/>> lizenziert.

Zusammenfassung

An important building block of a modern desktop environment is a multimedia architecture. The main requirement for such an architecture is to provide an easy-to-use interface for applications to integrate general multimedia functionality, such as media playback.

Phonon (formerly known as KDEMM) aims at fulfilling this role for the upcoming KDE 4. Internally, the Phonon architecture will be based on available full-featured multimedia frameworks. One back-end will be implemented for NMM, the Network-Integrated Multimedia Middleware. As a unique feature, NMM allows for controlling and connecting local and remote multimedia devices and software components. This kind of network transparency enables new application scenarios for the desktop that extend the traditional ways of interacting with multimedia: Media content can be played out on remote devices seamlessly, handed over between running applications, or presented to all connected users in a network simultaneously and synchronously.

This talk and accompanying paper will present the concepts of both Phonon and NMM, reflect the current state of their integration, and discuss the resulting new features for the KDE user.

1 Introduction

An important building block of a modern desktop environment is a multimedia architecture. The main requirement for such an architecture is to provide an easy-to-use interface for applications to integrate general multimedia functionality, such as media playback.

Phonon (formerly known as KDEMM) aims at fulfilling this role for the upcoming KDE 4. Internally, the Phonon architecture will be based on available full-featured multimedia

frameworks. One backend will be implemented for NMM, the Network-Integrated Multimedia Middleware. As a unique feature, NMM allows for controlling and connecting local and remote multimedia devices and software components. This kind of network transparency enables new application scenarios for the desktop that extend the traditional ways of interacting with multimedia: Media content can be played out on remote devices seamlessly, handed over between running applications, or presented to all connected users in a network simultaneously and synchronously.

2 Outline

In this paper and accompanying talk, we will first present the concepts and design of Phonon, the new multimedia architecture for the upcoming KDE 4. We will also present the current API of this framework and explain how it is used by a KDE developer. Then, we will describe the general design approach of NMM as well as its most important services. Furthermore, the API of NMM will be explained by demonstrating several simple applications. A tool provided is discussed next, which allows for the easy testing of configurations before starting to develop an application. Finally, we will discuss the integration of Phonon and NMM and explain the implementation of the first backend for Phonon using NMM. We will also present the current state of the implementation of this backend and will give an outlook of new features for the KDE user that can only be provided by the NMM backend for Phonon.

3 Phonon

For a long time using multimedia functionality in an application posed a big challenge to most developers. It is high time for the great media frameworks that have emerged to get an easy to use, but still powerful, interface.

3.1 A Short Look Back

KDE has been using aRts as its media framework and multimedia API since KDE 2.0. aRts has been a great framework in many ways, but didn't manage to keep up and is unmaintained by now. In the meantime many good alternatives have come up, including, but not limited to, libxine, gstreamer, NMM and Helix.

3.2 Multimedia Unleashed

With Phonon, KDE developers will be able to write applications with multimedia functionality in a fraction of the time needed with one of the above mentioned media frame-

works/libs. This will facilitate the usage of media capabilities in the KDE desktop and applications.

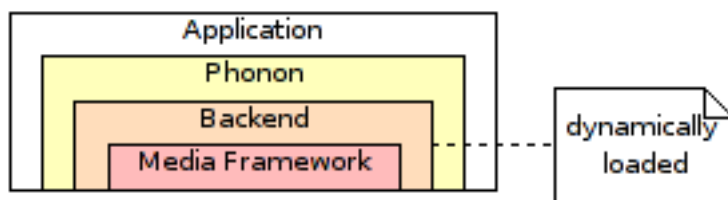
Developing an application with Phonon is like saying what you want and not how you want to do it. This leaves room for the user/administrator to customize things.

3.3 Choice

Users will not be restricted to the choice of soundserver or media framework made by KDE anymore. This is especially interesting for users of applications that require no interference with the sound board. An example scenario would be an application that needs jackd running with low latency on an ALSA device. ALSA dmix has to be disabled for low latency, so KDE either has to (properly) use jackd or not touch the sound board at all. Both options should be available to the user.

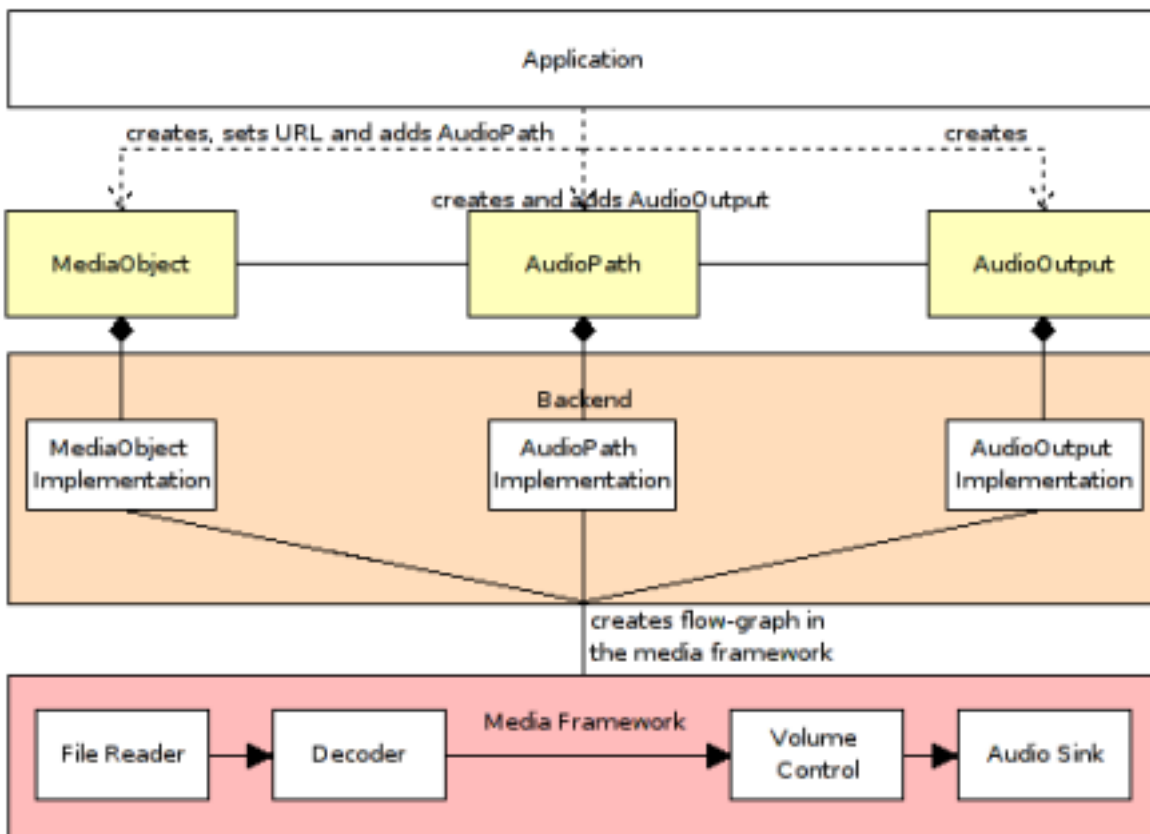
NMM for example provides a higher level of network transparency than any other framework for consideration. KDE would not want to miss the opportunity to provide KDE users with the power of NMM, while recognizing the high quality of other projects.

3.4 Backends



The functionality of Phonon is provided by a backend. This backend is the glue between the Phonon classes an application uses and the media framework. An application does not have access to the backend at all, it is completely wrapped by the Phonon front-end classes.

To implement a backend some abstract classes have to be implemented. As a backend implementor those classes should be treated as a representation of what the application wants.



A starting point to a backend implementation are the three classes `MediaObject`, `AudioPath` and `AudioOutput` that are used for audio playback. Those define a task for the media framework to read and decode a media file, apply a volume control and send it to the sound board.

For development and testing purposes there exists a Fake backend which fakes most of the requested functionality. The following list provides an overview of all classes that must be implemented for a full-featured Phonon backend:

- Classes used for playback:
 - *MediaObject*: controls media reading, decoding and playback
 - *ByteStream*: same as above but media data is provided by the application
 - *AudioOutput*: controls output to an audio device with an additional volume control
 - *AudioDataOutput*: allows the application to get the audio data for further processing like visualizations
 - *VideoWidget*: widget that displays the video signal
 - todo: DVD, TV features
- Classes used for capture:
 - *AvCapture*: controls capture from audio and video devices

- todo: write (and encode) to file
- Classes used for routing and processing:
 - *AudioPath*: defines audio signal routes and allows effect insertion
 - *VideoPath*: defines video signal routes and allows effect insertion
 - *AudioEffect*: selects effect to be applied to the audio signal
 - *VolumeFaderEffect*: like above but with specialized interface
 - *VideoEffect*: selects effect to be applied to the video signal
 - todo: more specialized effect interfaces
 - todo: network streaming

The complete API documentation is available online <<http://developer.kde.org/documentation/library/cvs-api/kdelibs-apidocs/phonon/html/index.html>>.

3.5 Application development

Application development using Phonon is quite easy and looks familiar to most KDE developers as can be seen in the following example. Here, we show how to play back an audio file (if you want more flexibility than the SimplePlayer interface which wraps the construction and connection calls conveniently).

First, you have to instantiate objects of type `MediaObject`, `AudioOutput` and `MediaPath`, which you can pass a `QObject` as parent. (`QObject`s automatically delete their children when they are deleted.)

```
m_audiooutput = new AudioOutput( this );
  m_audiopath = new AudioPath( this );
  m_media = new MediaObject( this );
```

Then, you have to set the category of the audio output. This category is used to determine the default audio output device and for software mixing. A software mixer application will have the possibility to control all outputs of one category at once.

```
m_audiooutput->setCategory( Phonon::MusicCategory );
```

After this, the audiopath is used to tell the backend that the audio data of the media should be sent to the given audio output. Audio path objects serve mainly for routing and processing (effects can also be inserted into the path).

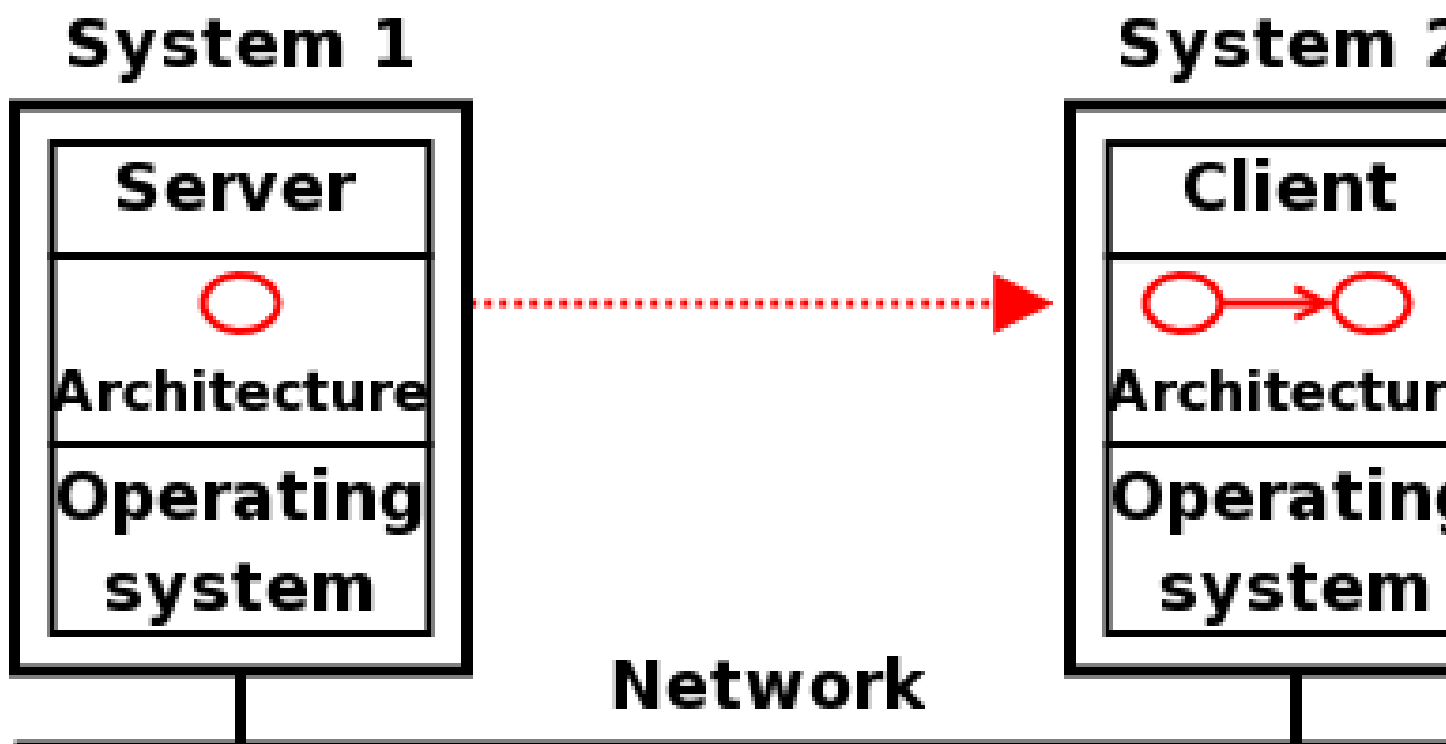
```
m_audiopath->addOutput( m_audiooutput );
  m_media->addAudioPath( m_audioPath );
```

Finally, a URL is passed to the media object which is using the fish protocol. As media frameworks don't understand the fish protocol (internally the backend will fail opening/understanding the URL) the Phonon `MediaObject` class will fall back to using a `ByteStream` on the backend side and stream the data using KIO. This way all KIO URLs can be supported by Phonon transparently. In the last line, the setup is complete and playback is started.

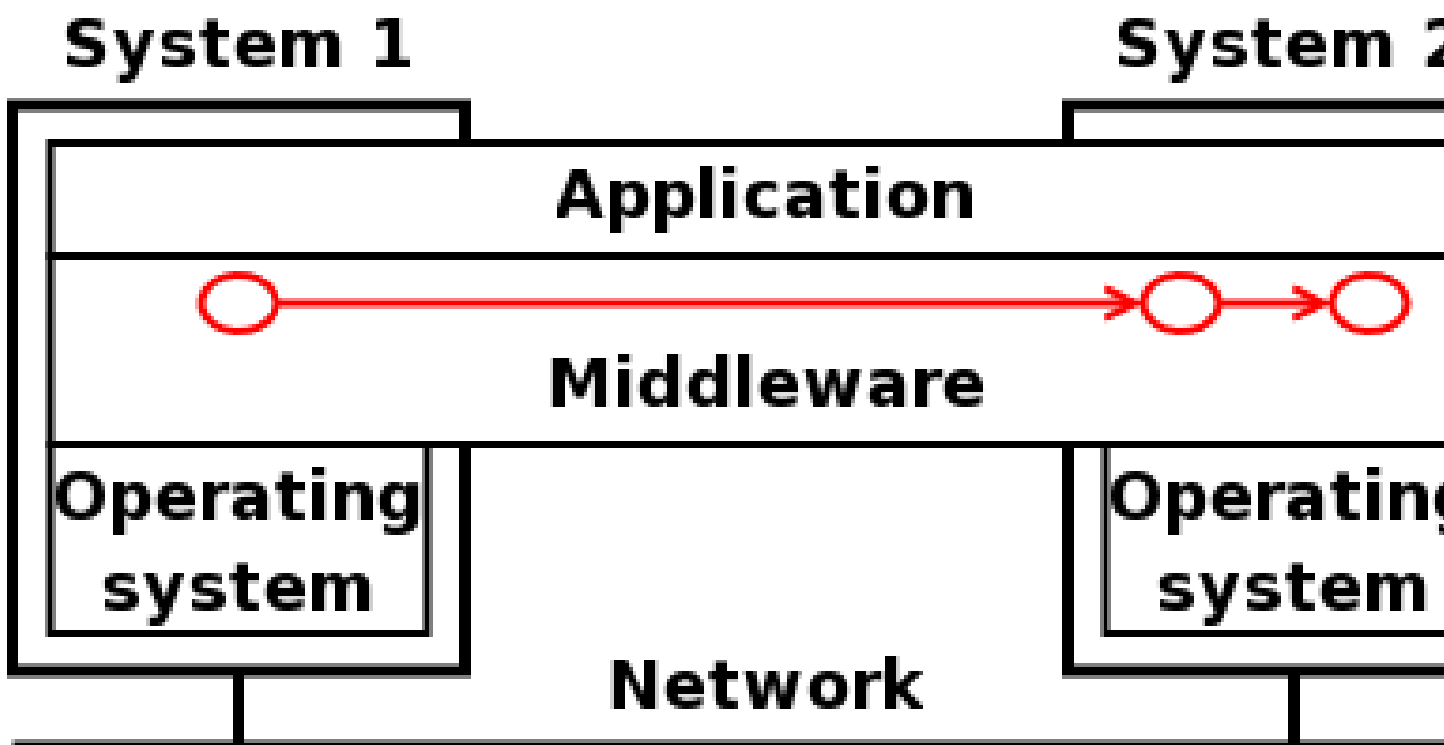
```
m_media->setUrl( KUrl( "fish://mymusicserver/home/user/music/song.ogg" ) );  
m_media->play();
```

4 Network-Integrated Multimedia Middleware (NMM)

The GNU/Linux operating system is rapidly becoming a favored platform not only for PCs but also for embedded systems, in particular for multimedia appliances such as set-top boxes, PDAs, and mobile phones. Besides the PC, an increasing number of these multimedia device already provides networking capabilities. However, today's multimedia architectures adopt a centralized approach, where all multimedia processing takes place within a single system. The network is, at best, used for streaming predefined content from a server to clients. Conceptually, such approaches consist of two isolated applications, a server and a client. The realization of complex scenarios is therefore complicated and error-prone – especially since the client has typically no or only limited control of the server.



The Network-Integrated Multimedia Middleware (NMM) <<http://www.networkmultimedia.org/>> overcomes these limitations by enabling access to all resources within the network: distributed multimedia devices and software components can be transparently controlled and integrated into an application. In contrast to all other multimedia architectures available, NMM is a *middleware*, i.e. a *distributed* software layer running in between distributed systems and application.



As an example, this allows for the quick and easy development of an application that receives TV from a remote device – including the transparent control of the distributed TV receiver. Even a PDA with only limited computational power can run such an application: the media conversions needed to adapt the audio and video content to the resources provided by the PDA can be distributed within the network. A running application can also be joined by several users, e.g. for simultaneously and synchronously accessing the same content on different devices, e.g. different audio/video systems within a household. While the distribution is transparent for developers, no overhead is added to all locally operating parts of the application. To this end, NMM also aims at providing a standard multimedia framework for all kinds of desktop applications.

NMM is both an active research project at Saarland University in Germany and an emerging Open Source project. NMM runs under GNU/Linux (and other UNIX operating systems), is implemented in C++, and distributed under the LGPL and GPL.

5 General Design Approach

5.1 Nodes, Jacks, and Flow Graphs

The general design approach of the NMM architecture is similar to other multimedia architectures. Within NMM, all hardware devices (e.g. a TV board) and software components (e.g. decoders) are represented by so called *nodes*. A node has properties that include its input and output ports, called *jacks*, together with their supported multimedia *formats*. A format

<http://graphics.cs.uni-sb.de/NMM/current/Docs/format/index.html> precisely defines the multimedia stream provided, e.g. by specifying a human readable type, such as 'audio/raw' for uncompressed audio streams, plus additional parameters, such as the sampling rate of an audio stream. Since a node can provide several inputs or outputs, its jacks are labeled with tags. Depending on the specific kind of a node, its innermost loop produces data, performs a certain operation on the data, or consumes data.

Our system distinguishes between different types of nodes: A *source* produces data and has one output jack. A *sink* consumes data, which it receives from its input jack. A *filter* has one input and one output jack. It only modifies the data of the stream and does not change its format or format specific parameters. A *converter* also has one input and one output jack but can change the format of the data (e.g. from raw video to compressed video) or may change format specific parameters (e.g. the video resolution). A *multiplexer* has several input jacks and one output jack; a *demultiplexer* has one input jack and several output jacks. Furthermore, there is also a generic mux-demux node available. A manual describing the development of nodes is available online <http://graphics.cs.uni-sb.de/NMM/current/Docs/plugins/index.html>.

These nodes can be connected to create a *flow graph*, where every two connected jacks need to support a 'matching' format, i.e. the formats of the connected input jack respectively output jack need to provide the same type and all parameters and the respective values present in one format need to be available for the other and vice versa. The structure of this graph then specifies the operation to be performed, e.g. the decoding and playback of an MP3 file.

Together, more than 60 nodes are already available for NMM, which allows for integrating various input and output devices, codecs, or specific filters into an application. A complete list is available online <http://graphics.cs.uni-sb.de/NMM/current/Docs/pluginlist/index.html>.

5.2 Messaging System

The NMM architecture uses a uniform message system for all communication. There are two types of messages. Multimedia data is placed into messages of type *buffer*. Messages of type *event* forward control information such as a change of speaker volume. Events are identified by a name and can include arbitrary typed parameters.

There are two different types of interaction paradigms used within NMM. First, messages are streamed along connected jacks. This type of interaction is called *instream* and is most often performed in *downstream* direction, i.e. from sources to sinks; but NMM also allows for sending messages in *upstream* direction.

Notice that both buffers and events can be sent instream. For instream communication, so called *composite events* are used that internally contain a number of events to be handled within a single step of execution. Instream events are very important for multimedia flow graphs. For example, the end of a stream (e.g. the end of a file) can be signalled by inserting a specific event at the end of a stream of buffers. External listener objects can be registered

to be notified when certain events occur at a node (e.g. for updating the GUI upon the end of a file or for selecting a new file).

Events are also employed for the second type of interaction called *out-of-band*, i.e. interaction between the application and NMM objects, such as nodes or jacks. Events are used to control objects or for sending notifications from objects to registered listeners.

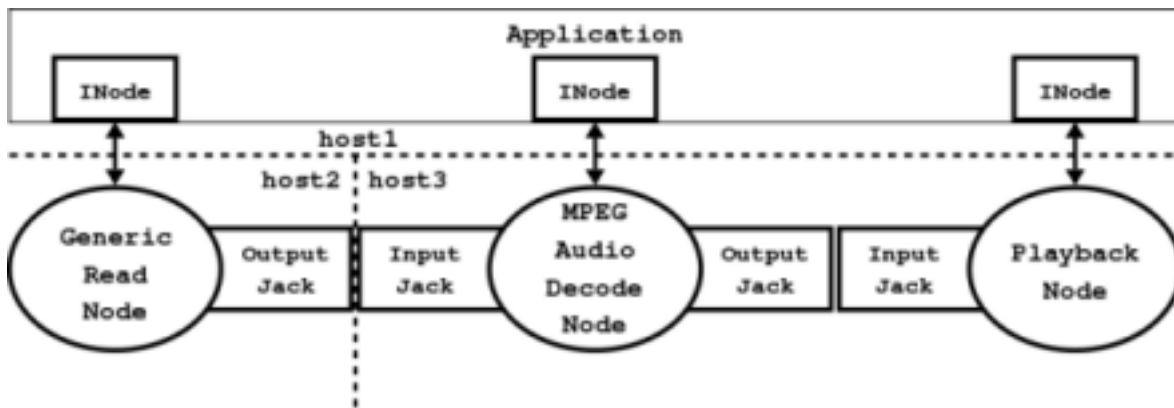
5.3 Interfaces

In addition to manually sending events, object-oriented *interfaces* allow to control objects by simply invoking methods, which is more type-safe and convenient than sending events. These interfaces are described in NMM Interface Definition Language (IDL) <<http://graphics.cs.uni-sb.de/NMM/current/Docs/idl/index.html>> that is similar to CORBA IDL. According to the coding style <<http://graphics.cs.uni-sb.de/NMM/current/Docs/style/index.html>> of NMM, interfaces start with a capital 'I'. For each description, an IDL compiler creates an interface and implementation class. While an implementation class is used for implementing specific functionality within a node, an interface class is exported for interacting with objects. During runtime, supported events and interfaces can be queried by the application. Notice that interfaces described in NMM IDL describe out-of-band and instream interaction.

5.4 Distributed Flow Graphs

What is special about NMM is the fact that NMM flow graphs can be distributed across the network: local and remote multimedia devices or software components encapsulated within nodes can be controlled and integrated into a common multimedia processing flow graph, a *distributed flow graph*. While this distribution is transparent for application developers, no overhead is added to all locally operating parts of the graph: In such cases, references to already allocated messages are simply forwarded – no networking is performed at all.

The following shows an example for a distributed flow graph for playing back MP3 files. A source node for reading data from the local file system (GenericReadNode) is connected to a node for decoding MPEG audio streams (MPEGAudioDecodeNode). This decoder is connected to a sink node for rendering uncompressed audio using a sound board (PlaybackNode). Once the graph is started, the source node reads a certain amount of data from a given file, encapsulates it into buffers, e.g. of size 1024 bytes, and forwards these buffers to its successor. After being decoded to uncompressed 'raw' audio, the converter node forwards data buffers to the sink node.



The application controls all parts of this flow graph using interfaces, e.g. INode for controlling the generic aspects of all instantiated nodes. Notice that three different hosts are present in our example. The application itself runs on host1, the source node on host2, and the decoder and sink node on host3. Therefore, NMM automatically creates networking connections between the application and the three distributed nodes (out-of-band interaction), but also between the source and the decoder node (instream interaction). Therefore, compressed MPEG audio data is transmitted over the network.

Notice that such a simple but distributed flow graph already provides many benefits. First, it allows an application to access file stored on distributed systems without the need for a distributed file system, such as NFS. Second, the data streaming between connected distributed nodes is handled automatically by NMM. Third, the application acts as 'remote control' for all distributed nodes. As an example, this allows for transparently changing the output volume of the remote sound board by a simple method invocation on a specific interface, e.g. IAudioDevice.

5.5 Distributed Synchronization

Since NMM flow graphs can be distributed, they allow for rendering audio and video on different systems. For example, the video stream of an MPEG2 file can be presented on a large screen connected to a PC while the corresponding audio is played on a mobile device. To realize synchronous playback of nodes distributed across the network, NMM provides a generic distributed synchronization architecture <<http://graphics.cs.uni-sb.de/NMM/current/Docs/time/index.html>>. This allows for achieving lip-synchronous playback as required for the above described setup. In addition, media presentations can also be performed on several systems simultaneously. A common application is the playback of the same audio stream using different systems located in different rooms of a household – a home-wide music system.

The basis for performing distributed synchronization is a common source for timing information. We are using a static clock within each address space. This clock represents the system clock that is globally synchronized by the Network Time Protocol (NTP) <<http://www.ntp.org/>> and can therefore be assumed to represent the same time basis throughout the network. A primer on how to set up NTP for NMM can be found online <<http://www.nmm.org/>>

[//graphics.cs.uni-sb.de/NMM/current/Docs/ntp/index.html](http://graphics.cs.uni-sb.de/NMM/current/Docs/ntp/index.html)>.

6 Registry Service

The registry service in NMM allows discovery, reservation, and instantiation of nodes available on local and remote hosts. On each host a unique *registry server* administrates all NMM nodes available on this particular system. For each node, the server registry stores a complete *node description* that includes the specific type of a node (e.g. 'sink'), its name (e.g. 'PlaybackNode'), the provided interfaces (e.g. 'IAudioDevice' for increasing or decreasing the output volume), and the supported input and output formats (e.g. an input format 'audio/raw' including additional parameters, such as the sampling rate).

The application uses a *registry client* to send requests to registry servers running on either the local or remote hosts. Registry servers are contacted by connecting to a well-known port. After successfully processing the request the server registry reserves the requested nodes. Nodes are then created by a factory either on the local or remote host. For nodes to be instantiated on the same host, the client registry will allocate objects within the address space of the application to avoid the overhead of an interprocess communication.

To setup and create complex distributed flow graphs, an application can either request each node separately or use a *graph description* as query. Such a description includes a set of node descriptions connected by edges.

For an application to be able to create a distributed flow graph, the NMM application called *serverregistry* needs to be running on each participating host. For purely locally operating applications this is not required. Then, a server registry is running within the application itself but not accessible from remote hosts.

Before a server registry can be used, it needs to determine which devices and software components are available on a particular host. Therefore, the registry needs to be initialized once using following command. More information on the serverregistry is available online <http://graphics.cs.uni-sb.de/NMM/current/Docs/server-registry/index.html>>.

7 Graph Builder

Manually setting up a distributed multimedia applications can become a complex task. For example using a graph description to playback an avi file requires additional knowledge about the audio and video format of the content to specify the correct decoder plugins. Therefore, NMM provides a service called graph builder which automatically creates a (distributed) flow graph for playing back the content from a given URL. The integration of required processing elements and their optimal distribution within the network is performed automatically. All file formats and codecs supported by NMM are automatical-

ly available when using the graph builder. A list of supported URLs can be found online <http://graphics.cs.uni-sb.de/NMM/current/Docs/clic/x142.html>.

8 Developing Multimedia Applications with NMM

A main goal of NMM is the ease the development of (distributed) multimedia applications. In the following, two simple examples are demonstrated. These can be found in the sub-directory `examples/helloworld` of NMM; an extended tutorial is also available online <http://graphics.cs.uni-sb.de/NMM/current/Docs/helloworld/index.html>.

8.1 Hello World!

The above described MP3 player will be used as our first example. The corresponding flow graph consists of three NMM nodes. For creating a request that can be used to query the registry service, following lines of code need to be provided.

```
GraphDescription graph;

NodeDescription readfile_descr("GenericReadNode");
NodeDescription decoder_descr("MPEGAudioDecodeNode");
NodeDescription audioplay_descr("PlaybackNode");

graph.addEdges(&readfile_descr,
              &decoder_descr,
              &audioplay_descr);
```

Such a graph description is forwarded to the registry service by forwarding it to a client registry, which can be accessed by creating a central NMM application:

```
NMMApplication* app =
    ProxyApplication::getApplication(argc, argv);
ClientRegistry& registry = app->getRegistry();
registry.requestGraph(graph);
```

If the last method invocation succeeded, all nodes were created. Before the flow graph can be started, we need to set the MP3 file to be played, e.g. the file given as command line parameter: First, we need to request the generic interface `INode` of the node description of the data source. Second, the specific interface `IFileHandler` is tried to be acquired. If it is not available for the instantiated node, an exception is thrown. Third, the set the filename by a simple method invocation. Together, this results in following lines of source code.

```
INode* readfile = graph.getINode(readfile_descr);
IFileHandler_var filehandler(readfile->getParentObject()
                             ->getCheckedInterface<IFileHandler>());
filehandler->setFilename(argv[1]);
```

The flow graph is configured and started by two additional commands:

```
graph.realizeGraph();
graph.startGraph();
```

For terminating and releasing the flow graph, following two lines are required:

```
graph.stopGraph();
registry.releaseGraph(graph);
```

The complete source code of the application 'helloworld1' is available online <<http://graphics.cs.uni-sb.de/NMM/current/Docs/helloworld/index.html>>.

8.2 Hello NMM!

Since NMM flow graphs can be distributed transparently, only a few additional lines of source code are required to redirect the audio output to a remote host. In particular, the node decoding the MPEG audio stream and the sink node for rendering audio using the audio device host need to be requested from a remote host. Therefore, only compressed audio is transmitted over the network.

```
GraphDescription graph;

NodeDescription readfile_descr("GenericReadNode");
NodeDescription decoder_descr("MPEGAudioDecodeNode");
NodeDescription audioplay_descr("PlaybackNode");

decoder_descr.setLocation(argv[2]);
audioplay_descr.setLocation(argv[2]);

graph.addEdges(&readfile_descr,
               &decoder_descr,
               &audioplay_descr);
```

In our example, only two additional lines are required: the two setLocation() calls. The host name of the remote system is given as second argument when starting the application. On this remote host, the two required nodes need to be available, i.e. the corresponding names need to be printed out when calling

```
user@linux:~/nmm/bin> ./serverregistry -s
```

During runtime, a serverregistry needs to be started:

```
user@linux:~/nmm/bin> ./serverregistry
ServerRegistry successfully started!
```

The complete source code of the application 'hellonmm1' is available online <<http://graphics.cs.uni-sb.de/NMM/current/Docs/helloworld/index.html>>.

8.3 Hello GraphBuilder!

The following example shows how to use the graph builder of NMM that automatically creates a flow graph from a given URL. All supported codecs and file formats supported by NMM can be played with this simple example.

First you have to create a central NMM application and a NodeDescription for the used audio and video sink.

```
app = ProxyApplication::getApplication(argc, argv);
NodeDescription playback_nd("PlaybackNode");
NodeDescription display_nd("XDisplayNode");
```

Then you have to create a synchronizer object for synchronized audio and video playback.

```
MultiAudioVideoSynchronizer av_sync;
IMultiAudioVideoSynchronizer_var sync(av_sync.getCheckedInterface<IMultiAudioVideoSynchr
```

After this, you have to create a GraphBuilder object and set the URL that should be played back.

```
GraphBuilder2 gb;
if(!gb.setURL(argv[1])) {
    throw Exception("Invalid URL given");
}
```

Now, you can specify the audio and video sink as well as the synchronizer that should be used.

```
gb.setMultiAudioVideoSynchronizer(sync.get());
gb.setAudioSink(playback_nd);
gb.setVideoSink(display_nd);
```

Finally, you can start the media playback using the following two lines of source code.

```
composite = gb.createGraph(*app);
composite->reachStarted();
```

This source code is taken from the application 'hellographbuilder1'. The complete source code of this application is available online <<http://graphics.cs.uni-sb.de/NMM/current/Docs/helloworld/index.html>>.

To redirect audio and video output to a remote host, again only the following two setLocation() calls must be added to the above example.

```
NodeDescription playback_nd("PlaybackNode");
NodeDescription display_nd("XDisplayNode");
playback_nd.setLocation(argv[2]);
display_nd.setLocation(argv[3]);
```

The complete source code of the application 'hellographbuilder2' is available online <<http://graphics.cs.uni-sb.de/NMM/current/Docs/helloworld/index.html>>. Please note that the graph builder also automatically requests the required nodes to decode audio and video from the corresponding remote host. Therefore, only compressed audio and video data is transmitted over the network. Before starting the application 'hellographbuilder2' the application serverregistry must be started on all remote hosts.

9 clic

The tool 'clic' <<http://graphics.cs.uni-sb.de/NMM/current/Docs/clic/index.html>>' (command line interaction and configuration) allows for setting up distributed flow graphs from a simple textual description. Using this tool, even complex application, such as distributed transcoding of audio/video files, can be realized quickly.

9.1 MP3 Player

A simple MP3 player can be set up using clic by providing following description stored in a .gd file, for example mp3play.gd:

```
GenericReadNode ! MPEGAudioDecodeNode ! PlaybackNode
```

The tool clic can now be started using following command:

```
user@linux:~/nmm/bin> ./clic mp3play.gd -i ~/music/song.mp3
```

9.2 Distributed MP3 Player

If, again, the decoding and rendering of the audio stream is to be performed on a remote host, following description needs to be provided.

```
GenericReadNode  
! MPEGAudioDecodeNode #setLocation("host2")  
! PlaybackNode         #setLocation("host2")
```

Clic is started as usual. In addition, the application 'serverregistry' needs to be running on host 'host2' as described above.

9.3 Using the Graph Builder

The tool clic also allows to use the graph builder of NMM to automatically create a (distributed) flow graph for playing back the content from a given URL. All file formats and codecs supported by NMM are also available when using the graph builder. Several examples how to create a locally running or distributed flow graph can found online <<http://graphics.cs.uni-sb.de/NMM/current/Docs/clic/index.html>>.

10 NMM Backend for Phonon

The goals of the NMM backend for Phonon are to meet the requirements as defined by Phonon, and to make NMM specific features available to the end-user. Phonon aims to provide an API for programmers of desktop applications. Therefore, such versatile network integration as implemented within NMM can't be the part of such an API. Otherwise, Phonon users

might be overwhelmed by the complexity of the API. Furthermore, the requirements for a Phonon backend would be quite difficult to comply for some multimedia frameworks.

The difficulty when bringing two technologies together, is to make their concepts work together. Due to the fact, that the ideas and the field of application of Phonon and NMM differ, their concepts differ as well. Phonon is fairly easy to use for achieving simple tasks, but it can still be used for more complex tasks. All you need is a MediaObject. Then add some paths and outputs and the playback can begin. In a more complex scenario, there might be more audio streams present. Maybe one should be played back using the sound board, and the other should be dumped on the hard disk.

10.1 Implementation of the NMM Backend

In order to render multimedia content, NMM uses a flow graph. In general, there are two, more or less automated, ways to build such a flow graph: The first one is to use a Graph Description file as described before. The second one is to use the graph builder to build up the flow graph automatically. Even though, with the help of a graph description a lot of complex scenarios can be realized very easily, the user still has to explicitly specify all used nodes and how they are interconnected. Depending on the content to be rendered, the needed flow graphs with its containing nodes are likely to differ. So, for creating a graph description, a user needs to know the details of the multimedia content, like encoding or available streams. This way provides the most flexibility for creating a (distributed) flow graph, but unfortunately the graph description files need to be edited by a user, and can hardly be generated automatically. Thus, this way won't work for the NMM backend.

On the other hand the graph builder, as introduced before, already realizes a automatic flow graph creation, but it is restricted to one audio sink and one video sink. Furthermore, the desired audio stream has to be specified before the flow graph is created. So, the user needs to know if there is more than one audio stream present and moreover, which one should be used. With Phonon a user can query for present streams and then decide which streams to use. So, all sum up, the current implementation of the graph builder needs some extensions to handle all Phonon requirements.

In the NMM backend for Phonon the class GraphHandler is used. This class is an adaption and extension from the graph builder. As well as the graph builder, the graph handler is able to automatically build up a flow graph. But while the graph builder creates the flow graph all at once, the graph handler stops the creation process when it has all information about the media that is needed by Phonon. So when a new MediaObject is created and the URL was set, the graph handler will reach this state. This allows the Phonon user to query for available streams. When new Paths are added to the MediaObject and got assigned to a stream, the graph handler will then add all necessary nodes to achieve the task represented by Phonon's path and output. These changes in the graph builder allow adding and removing numerous (remote) sinks and their needed preceding nodes.

The following example shows the basic usage of the graph handler:

Similar to the example given for the graph builder, first the NMM application, the NodeDescriptions for the used sinks and a synchronizer have to be created.

```
app = ProxyApplication::getApplication(argc, argv);

NodeDescription local_playback_nd("PlaybackNode");
NodeDescription remote_playback_nd("ALSAPlaybackNode");
NodeDescription local_display_nd("XDisplayNode");
NodeDescription remote_display_nd("XDisplayNode");

MultiAudioVideoSynchronizer av_sync;
IMultiAudioVideoSynchronizer_var sync(av_sync.getCheckedInterface<IMultiAudioVideoSynchr
```

Then the graph handler object needs to be created and the synchronizer has to be set.

```
GraphHandler gh;
if(!gh.setURL(argv[1])) {
    throw Exception("Invalid URL given");
}

gh.setMultiAudioVideoSynchronizer(sync.get());
```

The next step is to bring the graph handler to a state, where it can provide information about the media content, like available streams. To reach this state, the flow graph will be built up, like the graph builder does, but the graph handler stops the build up process, when sufficient information about the media is available. The number and kind of the nodes necessary, to reach this state depend on media content. While, in case of audio only files, like mp3 or wav, only a read node is needed to know that there is only one audio stream present, in case of an MPEG-2 file, some more processing is needed; hence, more nodes are automatically requested and connected.

```
gh.state(*app);
cout << "hasAudio(): " << gh.hasAudio() << endl;
cout << "hasVideo(): " << gh.hasVideo() << endl;
cout << "getNumberOfAudioStreams(): " << gh.getNumberOfAudioStreams() << endl;
cout << "getAudioStreamName(): " << gh.getAudioStreamName(0) << endl;
```

At this point various (remote) sinks can be added. All the nodes needed, to make a connection from the existing flow graph to the sink possible, including the sink node itself, is called a "branch". In most cases only an additional decoding node is needed.

```
int local_aid = gh.addBranch(gh.getAudioStreamName(0), local_playback_nd);
remote_playback_nd.setLocation("remotehost");
int remote_aid = gh.addBranch(gh.getAudioStreamName(1), remote_playback_nd);

int local_vid = gh.addBranch(gh.getAudioStreamName(0), local_display_nd);
remote_playback_nd.setLocation("remotehost");
int remote_vid = gh.addBranch(gh.getAudioStreamName(1), remote_display_nd);
```

Another nice feature of the graph handler is the possibility of adjusting the volume of audio branches independently.

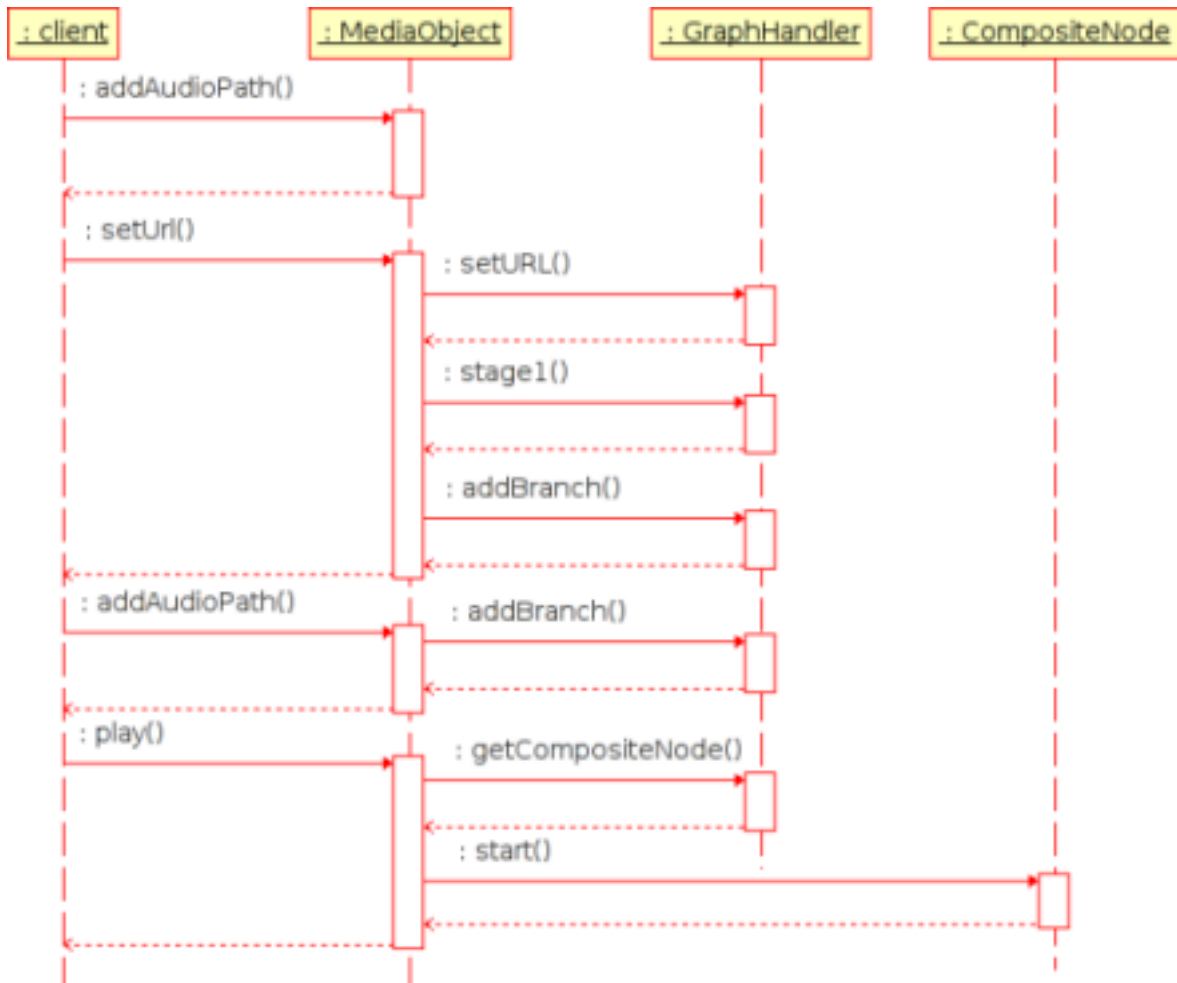
```
cout << "current volume is: " << gh.getBranchVolume(local_aid) << endl;
gh.setBranchVolume(local_aid, 80);
```

Finally, the composite node can be obtained, to e.g. register event listeners, or to simply start the flow graph. The usage of the composite node, obtained by the graph handler, is similar to the one returned by `GraphBuilder2::createGraph()`.

```
CompositeNode* composite = gh.getCompositeNode();

composite->start();
```

The following sequence diagram provides an overview about how the graph handler is used in the NMM backend for Phonon.



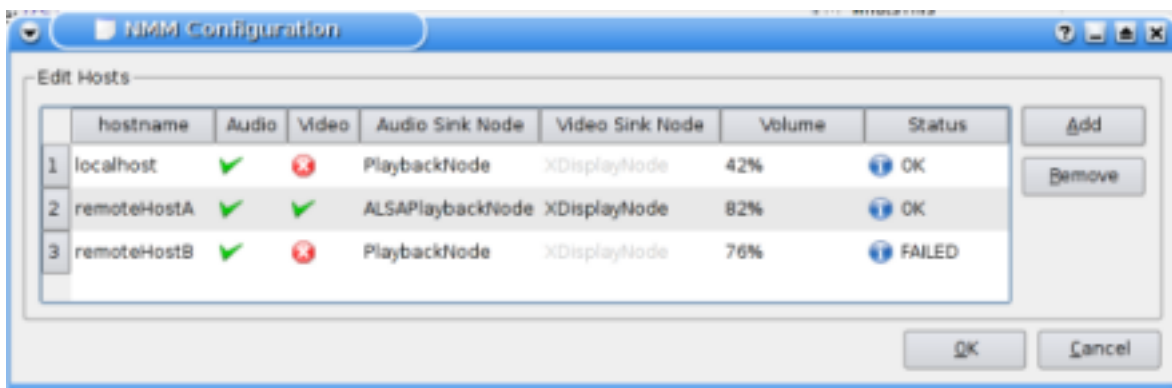
10.2 Integration of NMM Specific Features

Phonon itself already allows usage of a very useful NMM network feature. Since Phonon's `MediaObject` uses an URL to define the data source, the source may be located on a remote host. The benefit of this may be as simple as using a remote media file as source (without the need to set up some network file sharing); moreover, the remote source can be a TV card or a live camera feed as well.

An NMM configuration application will be used to make the NMM specific features accessible for end-users. This configuration application allows users to define additional hosts on

which audio and/or video should be played. The only requirement those remote hosts need to comply is a running serverregistry and, of course, providing the means to render audio and/or video. It is very likely, that the desired remote hosts differ from Phonon using KDE application to Phonon using KDE application. Therefore, the set of remote hosts can be specified for each application independently. This set can be dynamically adopted to the user's requirements. Additionally to simply specifying remote hosts, the audio stream can be selected, if there are more than one, and the volume can be adjusted. Furthermore, the user can select, whether audio and/or video should be rendered and which sink nodes should be used.

The following picture provides a preliminary glimpse at the configuration application. The basic idea for the looks of the configuration application was taken from Robert Gogolok's NMM-engine configuration widget for amarok.



The benefit of such a configuration tool is pretty clear: A lot of complex scenarios can be realized, while there is no need to actually implement/use a NMM based application, or create a graph description file yourself. Simply use the NMM backend as Phonon backend, and with the additional help of the NMM configuration application a lot of amazing NMM features are accessible with the Phonon using KDE multimedia application of your choice!

10.3 Status of the NMM Backend

Finally, the most important parts needed for audio rendering of the Phonon API are already implemented and are available as part of kdemultimedia (trunk/KDE/kdemultimedia) via svn. The current ongoing work mainly focuses on implementing a video widget and finish the NMM configuration application.

Since Phonon is still under development, its API is subject to changes. Thus, the backend needs to adopt to these changes. Furthermore, not all parts of the Phonon API are implemented yet. For example, the Phonon filters are currently not supported, but it is aimed at starting this work as soon as the current ongoing work will be finished.

11 Conclusions

In this paper we first presented the concepts of Phonon, the new multimedia architecture for the upcoming KDE 4 and discussed how this architecture meets the requirements of a modern desktop environment. Internally, the Phonon architecture uses backends to allow the integration of different multimedia frameworks. After this, we presented the Network-Integrated Multimedia Middleware (NMM) which provides a new approach for developing distributed multimedia applications. This architecture considers the network as an integral part and enables the intelligent use of devices and software components distributed across a network including synchronous rendering – a feature not available for other architectures. Finally, we discussed the integration of the first Phonon backend that is based on NMM and explained the extensions for NMM to meet the requirements of Phonon. This also includes the current state of the implementation of this backend as well as the ongoing work and an outlook of new features for the KDE user.