

Integration of the Network-Integrated Multimedia Middleware in KDE Phonon

Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Ingenieur (Fachhochschule)

eingereicht von

Bernhard Fuchshumer

Betreuer: Dr.-Ing. Marco Lohse, Lehrstuhl für Computer Graphik
der Universität des Saarlandes, Saarbrücken, Deutschland
Begutachter: Dipl.-Ing. (FH) Peter Kulczycki

26. Juni 2006

Preface

During my studies in the department of Software Engineering at the University of Applied Sciences of Upper Austria, Hagenberg, Austria, my particular interests in the field of Linux, computer networks and multimedia emerged. Thus, I was deeply impressed by the Network-Integrated Multimedia Middleware (NMM), when I first read about this project and its outstanding capabilities. Unsurprisingly, I was very excited when I got the opportunity to work as intern at the Computer Graphics Lab at the Saarland University, Saarbrücken, Germany, where NMM is developed. This job made it possible to take a closer look at NMM and gain a better understanding in a very pleasing, supporting work environment.

At the end of the internship the Computer Graphics Lab and Motama GmbH, Saarbrücken, Germany, offered me to develop an NMM-based backend for Phonon. This thesis results from my work on this backend. Since I have not been that deeply involved in an open source project before, this has been a great experience for me. Most important, it provided the opportunity to work with software that was very recent or even still under development, which was challenging and very interesting.

Furthermore, this project gave me the chance to contribute to a paper on NMM, Phonon and how NMM can be integrated into Phonon for the LinuxTag 2006 in Wiesbaden, Germany [FKRL06]. This paper is a joint effort of Phonon and NMM developers.

I wish to thank Dr.-Ing. Marco Lohse, post-doctoral researcher at the Computer Graphics Lab at the Saarland University and chair man and co-founder of Motama GmbH, Dipl.-Ing. Michael Repplinger, PhD student at the Computer Graphics Lab at the Saarland University and co-founder of Motama GmbH, and Prof. Dr.-Ing. Philipp Slusallek, professor at the Computer Graphics Lab at the Saarland University and co-founder of Motama GmbH, for offering this topic to me, for the opportunity to contribute to the paper for the LinuxTag, and for their technical and financial support. For his assistance during the development of the backend and the writing of the thesis even when his time was short, I would like to especially thank Dr.-Ing. Marco Lohse.

For his support during the implementation for this work and for providing valuable insight into Phonon and into the KDE project, I wish to thank Matthias Kretz, the project leader of Phonon.

Finally, I would like to express my dearest gratitude to my whole family, who always supported and encouraged me. Without them this would not have been possible. Especially to my brother Stefan I would like to address my deepest appreciation for sharing his experience with me and for providing vital feedback during writing this thesis.

Bernhard Fuchshumer

Abstract

Phonon aims to provide a common multimedia API for the upcoming KDE 4. Since Phonon only defines the API, it allows multimedia frameworks to implement a so-called backend. These backends are used to make available the functionality as defined by Phonon. The objective of this work is to provide a backend based on the Network-Integrated Multimedia Middleware (NMM). Additionally, the NMM backend for Phonon should make it possible to use NMM specific features, like distributing playback across the network, that are not available by means of the Phonon API.

NMM uses a flow graph concept for playback of multimedia data, for example. Thus, the job of the backend is to build up an appropriate flow graph due to the requirements of Phonon. As NMM does not provide sufficient facilities itself, a new class, the `GraphHandler`, was implemented as part of this work. This class offers a very flexible way to automatically build up a flow graph and even manipulate this graph while it is running.

In order to integrate the advanced features of NMM into the backend, a DCOP interface was implemented as part of the backend. This interface offers functionality to alter the flow graph, e.g. by adding additional hosts for playback. Any application that can use DCOP can use the interface of the backend; thus, a developer using Phonon does not need to add extra, NMM-specific code to make these advanced features of NMM available to users. As part of this work a configuration application using the DCOP interface was implemented.

The NMM backend for Phonon does not only provide the functionality as defined by Phonon, but also it allows any KDE users, who use Phonon-based applications, to benefit from the advanced features of NMM.

Kurzfassung

Das Ziel von Phonon ist es, eine einheitliche Multimedia-API für das im Frühjahr 2007 zur Veröffentlichung vorgesehene KDE 4 zur Verfügung zu stellen. Konzeptbedingt, um ein gewünschtes Multimedia-Framework einzubringen, bedingt nun die Realisierung des Phonon APIs die Implementierung eines sogenannten Backends, welches die Abbildung des Phonon APIs auf das Konzept des jeweiligen Frameworks vornimmt.

Die Aufgabe dieser Arbeit ist nun die Entwicklung eines Backends zur Integration der Network-Integrated Multimedia Middleware (NMM) in Phonon. Desweiteren soll dieses Backend insbesondere auch den Zugriff auf NMM-spezifische Funktionalitäten, zum Beispiel das Verteilen der Wiedergabe einer Datei im Netzwerk, die mittels Phonon nicht zugänglich sind, ermöglichen.

NMM bedient sich eines Flussgraphen-Konzepts, um Aufgaben, wie zum Beispiel die Wiedergabe von Multimedia-Daten, zu bewerkstelligen. Deshalb ist auch die Aufgabe des Backends, einen den Anforderungen von Phonon entsprechenden Flussgraphen zu erstellen. Da NMM für diese Aufgabe selbst keine ausreichenden Mittel zur Verfügung stellt, wurde im Rahmen dieser Arbeit die Klasse `GraphHandler` entwickelt. Diese Klasse ermöglicht den automatischen Aufbau eines Flussgraphen und sogar dessen Anpassung und Erweiterung zu jedem beliebigen Zeitpunkt.

Um auch die erweiterten Möglichkeiten von NMM in das Backend zu integrieren, wurde ein DCOP-Interface implementiert. Dieses Interface bietet ausreichend Funktionalität, um den Flussgraphen anzupassen, zum Beispiel durch das Hinzufügen von zusätzlichen Hosts für die Wiedergabe. Jegliche Anwendung, welche die Möglichkeit hat, DCOP zu verwenden, kann auch dieses Interface benutzen. Darum müssen Entwickler, die Phonon für die Realisierung von Multimedia-Funktionalität ihrer Anwendung verwenden, keinen zusätzlichen, NMM spezifischen Code hinzufügen, um den Benutzern dieser Anwendung die fortgeschrittenen Möglichkeiten von NMM zur Verfügung zu stellen. Im Rahmen dieser Arbeit wurde eine Konfigurationsanwendung entwickelt, die dieses DCOP-Interface nutzt.

Das NMM Backend für Phonon bietet also nicht nur die Funktionalität, die Phonon definiert, sondern es erlaubt auch allen KDE-Benutzern, welche Phonon basierte Anwendungen verwenden, von den erweiterten Möglichkeiten von NMM zu profitieren.

Contents

1	Introduction	1
2	State of the Art	3
2.1	NMM	3
2.1.1	Concept Overview	5
2.1.2	Formats, Jacks and Nodes	5
2.1.3	Flow Graph	9
2.1.4	Buffers	10
2.1.5	Registry Service	10
2.1.6	Node and Graph Descriptions	11
2.1.7	Messaging System	11
2.1.8	Communication Architecture	13
2.1.9	Binding Framework	14
2.1.10	Interfaces	16
2.1.11	Synchronization	18
2.1.12	Application Development	19
2.1.13	Graph Builder	21
2.2	Phonon	23
2.2.1	Motivation	23
2.2.2	Concept	24
2.2.3	Sample Application	26
2.2.4	Current Status	27
2.3	Summary	27
3	Concept	28
3.1	Backend Concept	29
3.1.1	MediaObject	29
3.1.2	AudioPath and VideoPath	34
3.1.3	AudioOutput and VideoOutput	35
3.2	Integration of Advanced Capabilities of NMM	35
3.2.1	Goals	36

3.2.2	Configuration Concept	36
3.3	Summary	37
4	Implementation	38
4.1	GraphHandler	38
4.1.1	Design and Implementation	39
4.1.2	The Three-Stages-Concept	41
4.1.3	Specifying Additional Sink Nodes	42
4.1.4	The Filter/Effect API	44
4.1.5	Automatic Flow Graph Build-up	46
4.1.6	Field of Application	47
4.2	Advanced Configuration	47
4.2.1	GUI Design	47
4.2.2	Implementation	49
4.2.3	Integration in Backend	51
4.3	Summary	52
5	Examples	53
5.1	Distributed Audio Playback	53
5.2	Distributed Audio and Video Playback	55
5.3	Using a Remote Data Source	57
5.4	Using a Remote Live Data Source	57
5.5	Summary	58
6	Conclusion	60
6.1	Achievements	60
6.2	Future Work	61
	Bibliography	63

Chapter 1

Introduction

A few years ago, integrating multimedia features into a Linux/Unix based application required a lot of work. Since back then this functionality had to be implemented from scratch. Fortunately, nowadays multimedia frameworks that aid development of multimedia applications on Linux/Unix platforms are available, like GStreamer [gst], NMM [wgc], Helix [hel], or libxine [xin]. All those frameworks have more or less certain unique benefits. Thus, the developer has to choose one. Furthermore, since they all offer a wide variety of features, their concepts and; moreover, their APIs differ. A consequence of a rich feature-set often is a complex API that requires time to fully comprehend, before a developer can use it for implementing the desired features.

Phonon [Kre], which will be available as part of the upcoming KDE 4, aims to overcome the limitations stated above. The KDE project provides a graphical desktop environment for Linux/Unix platforms [KDEb]. Phonon offers a fast to learn API that can be used to implement trivial as well as sophisticated scenarios. Futhermore, it only defines the API, while the actual work is done by a multimedia framework. Any framework that meets the requirements of Phonon can be used to be integrated into Phonon. Phonon uses so-called backends to integrate such multimedia frameworks. Notice, that the user of the Phonon-based applications can decide which backend and therefore which multimedia framework is supposed to be used.

The objective of this work is to integrate the *Network-integrated Multimedia Middleware (NMM)* into Phonon. The emphasis of NMM lies in integrating and configuring distributed components in a network. Since NMM offers features that are not available when using the Phonon API, additional means of making those features available to end-users are required. To sum up, the two major tasks for integrating NMM into Phonon are developing a proper coupling to the Phonon API, by implementing a backend, and providing additional facilities to make the advanced features of NMM

accessible for users of Phonon.

In Chapter 2 **State of the Art** the concepts of NMM and Phonon will be introduced. The main goal of this chapter is to provide a fundamental knowledge of those two technologies; therefore, some aspects with less importance for this work are omitted. Chapter 3 **Concept** will address *what* the NMM backend for Phonon has to accomplish. Furthermore, this chapter introduces the concept of how to make a selection of the advanced features of NMM available for Phonon users. *How* this has been achieved is stated in Chapter 4 **Implementation**. Some of the scenarios a user can create with the help of Phonon and the NMM backend are shown in Chapter 5 **Examples**. Finally, Chapter 6 **Conclusion** will provide the achievements of this work as well as the future work.

Chapter 2

State of the Art

This chapter will introduce the two main technologies used in the context of this work. However, not all details can be addressed since both technologies offer a high complexity. Therefore, aspects of less importance for this work are omitted, but can be found in the referred literature.

2.1 NMM

Network-Integrated Multimedia Middleware (NMM) started off as a research project of the Computer Graphics Lab at the Saarland University, Saarbrücken, Germany. The source code is available under LGPL and GPL version two. Since February 2006, commercial solutions based on NMM have been provided by the company Motama GmbH, Saarbrücken, Germany. NMM can be found online under [wgc].

The Network-Integrated Multimedia Middleware is explained in detail in [Loh05b]. However, a brief introduction to NMM can be found in [Loh05a]. The goal of NMM is not only to provide an easy-to-use multimedia framework for rendering and manipulation of multimedia content on GNU/Linux systems but also to integrate the network with all its benefits. Essentially this means network transparent connection and control of distributed components. This allows the application developer using NMM to share devices reachable within the network, which makes it possible for a PDA, for example, to receive data from a TV card located in a standard PC.

Despite the rather complex scenarios that can be created with NMM, the application development is not complicated at all. NMM abstracts all its network functionality so that the developer does not need to care about networking implementation. This is all taken care of in NMM.

Most multimedia frameworks are designed for local usage only, which

means that all necessary data processing has to be done locally as well. If the network is supported at all, it usually only serves for the streaming of data from a server to clients (see Figure 2.1). Client and server are isolated applications, which makes the realization of complex scenarios very complicated and error-prone.

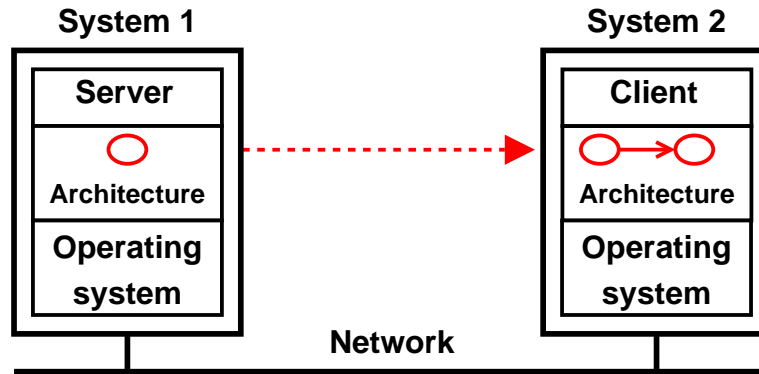


Figure 2.1: Client/Server approach [Loh05a].

Since NMM is *middleware*, it follows a different approach to integrate the network. A definition of the term middleware can be found in [BCP03]:

The role of middleware is to ease the task of designing, programming and managing distributed applications by providing a simple, consistent and integrated distributed programming environment. Essentially, middleware is a distributed software layer, which abstracts over the complexity and heterogeneity of the underlying distributed environment with its multitude of network technologies, machine architectures, operating systems and programming languages.

Middleware has to find and bind distributed components in order to provide the service as expected [RP02]. Thus, NMM is middleware that makes it possible to implement applications using distributed components as if they were all available locally. There is no need for the developer to implement any server or client applications that would achieve the integration of these distributed components since the integration is already implemented within the middleware that is part of NMM, as shown in Figure 2.2 on the next page.

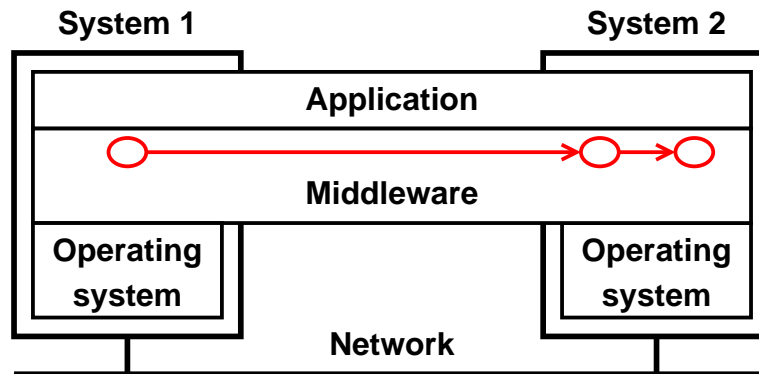


Figure 2.2: Middleware approach [Loh05a].

2.1.1 Concept Overview

The general concept of NMM is not that difficult to understand. A multimedia framework usually provides means to at least make rendering of media content possible. In the simplest case, an application using such frameworks is used to render multimedia content. NMM uses a logical model called *flow graph* to achieve such tasks.

A flow graph consists of *nodes*. Within NMM a node represents a task, like reading data from the hard disk, decoding, encoding, or rendering, to name only a few. In general, nodes are the processing elements of a flow graph. Nodes have defined data inputs and outputs, so-called *jacks*. A *format* is used to define the nature of the data. Usually, an input jack accepts several different formats, but this completely depends on the capabilities of the software and hardware component used to achieve the task as represented by the node [LS01]. See also [Loh05a], [Loh05b], [Rep03], and [LRS02].

2.1.2 Formats, Jacks and Nodes

Format

The format defines the type and certain additional properties of multimedia data [Loh05b, page 158], and [LS01]. The type is a tuple made up with a value representing the kind of media, like *audio*, *video*, or *av*, and a value representing the encoding, e.g., *mpeg*, *mp3*, *ac3*, or *raw*. The additional properties usually describe, for example, the resolution, the aspect ratio, or the color space of the data.

An MPEG-2 Program Stream (PS) would have the type *av/mpeg*, but after the comprised data streams have been separated, and the individual Packetized Elementary Streams (PES) are accessible, the type might be, for

example, *audio/mp3* or *video/mpeg2*. The concrete type of PES depends on the encoding used [mpe]. Thus, as the data is processed by the nodes, the format changes, until the data has an appropriate format to be rendered (or whatever task the sink node accomplishes).

Jacks

Jacks, as introduced by [Loh05b, page 155], are the defined data inputs and outputs of nodes. Therefore, jacks are used to enable connections between two nodes. In order to achieve this, the output jack of the one node is connected with the input jack of the other node. Furthermore, the formats of the jacks have to match to successfully establish a connection. To identify a jack, a node-wide unique tag is used.

Nodes

Nodes are the processing elements of NMM. They achieve certain tasks, for instance reading data from the hard disk, demultiplexing data, decoding data, encoding data, rendering, etc. (see [Loh05b, page 148]). Despite the fact that the purpose of nodes differ, NMM nodes can be grouped into the following types of nodes [Loh05a], [Loh05b, page 150], and [Rep03, page 38].

Source: These nodes do not have an input jack, since they usually obtain the data directly, e. g. from the hard disk, or from some recording device; they only provide this data for further usage. They *produce* the data used in the flow graph.

Sink: Sink nodes do not have an output jack, because sinks are used to render data or to dump data on the hard disk. In general, they *consume* the data.

Filter: Filter nodes modify the data while keeping the format unchanged.

Converter: A Converter, as the name already says, converts the data; as a consequence, the format changes. A decoding node is a converter node, since the format changes from some encoded type to raw type.

Demultiplexer: These nodes are used when file specifications make it possible for numerous media streams to be present in one file, e.g. MPEG-2 [mpe]. Before these streams can be processed any further, they need to be separated. This task is done by a demultiplexer. These kinds of nodes have one input jack and several output jacks.

Multiplexer: A multiplexer is used for the opposite task as the demultiplexer. If multimedia data is supposed to be dumped onto the hard disk, for example, into an MPEG-2 compliant file, all present streams (PES) have to be multiplexed to one single stream (PS) before the data can be saved [mpe].

Multiplexer-demultiplexer: This node type is a generalization of the node types introduced above, therefore it supports any number of inputs and outputs.

Currently, there are more than 60 nodes available. A complete list of all nodes can be found online at [wgb].

All the nodes functionality is defined by numerous interfaces, but usually a node does not implement all required interfaces directly. A generic base class for all different types of nodes, previously introduced, is available to aggregate all interfaces needed to define the generic functionality expected by those nodes. These generic base classes also provide generic implementations for some methods. In addition to implementing the generic base class, a node can implement several specific interfaces in order to make the specific functionality, as defined in the interface, accessible to users of the node. The `PlaybackNode`, for example, is used for audio playback. It implements, aside from its generic base class, an interface called `IAudioDevice`, which defines the means for adjusting the volume. For more details see the upcoming Subsection on interfaces 2.1.10, or [Loh05b, page 120].

A state machine is used to define the life-cycle of a node. Figure 2.3 shows a quick overview of all states and their transitions [Loh05a], [Loh05b, page 151], and [Rep03, page 47].

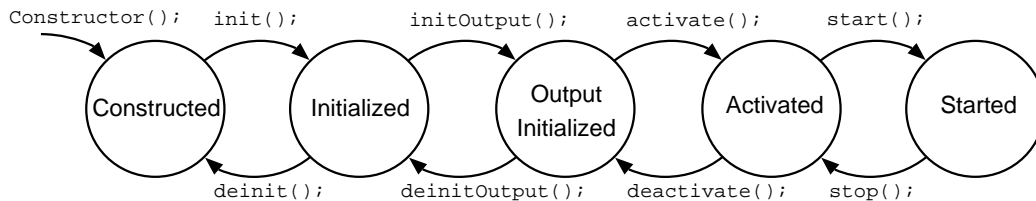


Figure 2.3: States of a node [Loh05b, page 151].

Constructed: After a node was instantiated, it is in this state.

Initialized: The call of the method `init()` puts the node into this state. Due to that, all processing-data-independent resources are requested and the static input and output properties are set. At this

state, the input jack of the node can be connected with the output jack of another node. A connection between two jacks always has a designated format. The static input properties are a list of supported formats that is used during the connection process.

Output initialized: When a nodes reaches this state, the so-called working properties are set. Concerning the working input property, this can be done quite easily, since this only depends on the format used to connect the input jack with the output jack of the preceding node. Of course, this connection is required to be successfully established prior to bringing the node in the state *output initialized*, by calling the method `initOutput()`. However, setting the working output property is a much more difficult task. First, the available jacks have to be created, then their working output properties have to be set. Even though the available jacks of a converter or filter node are quite clear, the number of jacks available for a demultiplexer node depends on the given data. Therefore, the node can request data for analysis from its predecessor in order to gather enough information about the data to create all output jacks and set their working output properties.

All output jacks can now be used to establish connections with the input jacks of the successor nodes.

Activated: After reaching the state *activated*, all resources required for the input and output formats are reserved. Now, all needed resources for processing the data are reserved. Therefore, this state is required before any direct successor nodes can reach the state *output initialized*.

Started: When the method `start()` is called, the node will start processing available data. Since all nodes of a flow graph are supposed to work concurrently, the *main loop* of every node runs in a separate thread.

CompositeNode

The convenience class `CompositeNode` is a wrapper for mulitple nodes. It makes registering event handlers easier, since without this class, registering the eventhandler at each node would be necessary (see Subsection 2.1.7 for information on events). Furthermore, the `CompositeNode` can be used to bring all contained nodes to a certain state, like *started*, with one sinlge method call. Subsection 2.1.13 gives an example of how the class `CompositeNode` can be used.

2.1.3 Flow Graph

In NMM a flow graph represents a task requested by the user. This might be, for example, playback, transcoding, or recording of media data. In order to render multimedia content, certain tasks have to be done. At first, the plain data has to be read from the hard disk, network, or whatever else the source of the data might be. Then, if there is more than one media stream present, the data needs to be demultiplexed. After each stream is accessible independently, those streams need to be decoded before they can be finally rendered. A flow graph needs to accomplish all these tasks. Therefore, a flow graph consists of nodes that can achieve these required tasks. Figure 2.4 shows a very simple flow graph for the purpose of MP3 file playback.

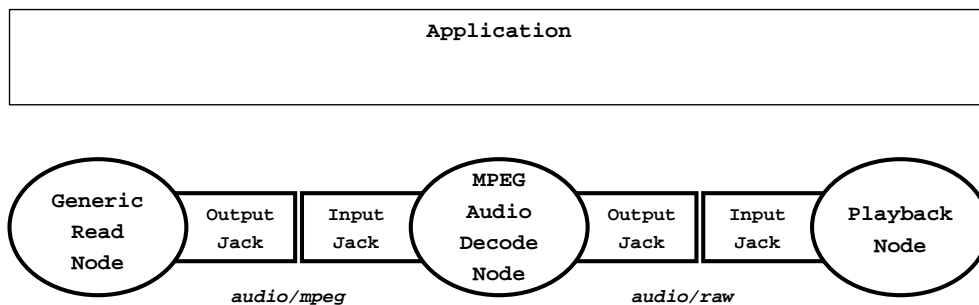


Figure 2.4: A flow graph for MP3 playback [Loh05a].

As already mentioned before, a goal of NMM is to integrate the network. The huge advantage that results from the fact that NMM uses the flow graph concept is that each node can run on a different host. Figure 2.5 on the following page shows how the flow graph from Figure 2.4 might be distributed. The big advantage of the flow graph in Figure 2.5 on the following page, however, is that the source node is located on a different host than the nodes that decode and play back the data. Furthermore, the NMM application controlling the flow graph is located on a third host. Even though the concept would make it possible to run the nodes responsible for decoding and rendering the data on a different host as well, it is not done in this example, because it would require a lot of bandwidth to send the decoded data over the network, which should be avoided.

Another advantage that arises from building up a flow graph with nodes is the possibility of creating new complex scenarios by simply connecting available nodes. Nodes do not require a certain predecessor or successor, only the format used for the connection has to match.

For further information concerning the flow graph concept of NMM see [Loh05a], [Loh05b, page 41], [Rep03, page 34], and [LRS02].

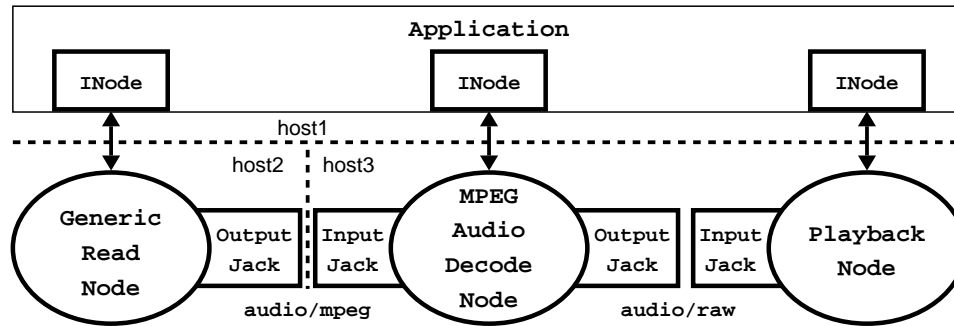


Figure 2.5: A distributed flow graph for MP3 playback [Loh05a].

2.1.4 Buffers

As stated in [Loh05b, page 113], and [Rep03, page 39], exchange of multimedia data within NMM is done with *buffers*. Allocating and releasing memory are quite time-consuming operations; thus, buffers help to minimize time and memory costs by enabling its reuse when it is not needed anymore. Hence, NMM provides a certain number of preallocated buffers that can be requested by nodes. A so-called *buffer manager* handles these requests for buffers. Additionally, the *buffer handler* administrates a memory pool to make variable-sized buffers possible, which is necessary since buffer size requirements are likely to change within a flow graph. The desired buffer size usually depends on the data, since a buffer should optimally contain enough data for the successor node to process independently, without the need to wait for other buffers. Of course, this does not work for every node, but at least the buffers sent to sink nodes provide the data needed to render one frame.

Buffers can have a time stamp that reflects the supposed presentation time of the contained data. Nodes that acquire information about this presentation time while processing the data usually set this time stamp.

2.1.5 Registry Service

The *registry service* provides some very important services [Loh05a]. [Loh05b, page 203], and [Rep03, page 109] provide a very detailed description of the registry service. Every host needs a running *registry server*, the so-called *serverregistry*. It is responsible for all node management, for example, discovery, reservation and instantiation of local nodes. An application can use a *registry client* to query either local or distributed running registry servers. Thus, when an NMM application requests a node from the registry server,

the registry server needs to check whether the node is available. If the node is available and instantiation is allowed, the node can finally be instantiated. Some nodes can only be instantiated a limited number of times. This number usually depends on the hardware present on the host, e.g. a node that reads the data from a TV board—the number of possible instantiations of this node equals the number of TV boards present. Thus, the registry server is responsible for managing hardware access as well.

To minimize system memory usage, every node can be loaded and unloaded separately. Therefore, the code for each node is linked into an independent library, a so called *plugin*. This allows the registry server to dynamically load and unload libraries as needed.

2.1.6 Node and Graph Descriptions

Node Descriptions

A *node description* contains the type (e.g. sink), the name (e.g. Playback-Node), the supported input and output formats, and some additional information about the capabilities of a node, like a list of supported interfaces. Node descriptions can be used to request nodes at the registry server. The class `NodeDescription` is used to represent such node descriptions in NMM. See Subsection 2.1.12 for examples of how to use node descriptions.

Graph Descriptions

A *graph description* is used to represent a flow graph. Therefore, it contains the desired node descriptions and how they are interconnected—the edges of the flow graph. Such graph descriptions can be used to request a whole flow graph at once. This makes the process of building a flow graph easier. Within NMM, the class `GraphDescription` implements graph descriptions. Examples in Subsection 2.1.12 show the usage of graph descriptions.

2.1.7 Messaging System

As stated in [Loh05b, page 111], the messaging system of NMM consists of three different parts.

1. There are message types for multimedia data and control information. It is necessary that not only media data is exchanged between nodes, but also additional control information. With the help of control information, so-called *events*, a node can request multimedia data from its predecessor during the `initOutput()` call of this node, for example. As

already introduced in the Subsection on Buffers 2.1.4, the class `Buffer` is used for sending multimedia data. However, the base class for both message types is the class `Message`. This class makes it possible to specify the desired direction. In general, there are two different ways such events can be passed to a node: *out-of-band* and *instream*. While out-of-band communication is used for communication between application and nodes, instream communication is used to communicate between nodes. Therefore, instream communication can be further differentiated in *downstream* and *upstream* communication. Downstream communication would mean that the event is passed to the successor node; upstream communication means passing the event to the predecessor node. All sum up, the direction of a message can be out-of-band, downstream, or upstream.

In order to obtain some level of quality of service a message can be marked as *mandatory*. Such messages are not allowed to be discarded, for example, during flushing the whole flow graph.

Furthermore, a time stamp can be specified for a message. This time stamp will be used for synchronization purposes.

2. Abstract interfaces that can be used for interaction. Since the messaging system of NMM supports different interaction paradigms, different interface classes are used to identify the intended kind of interaction. Figure 2.6 shows an overview of the classes that can be used for interaction.

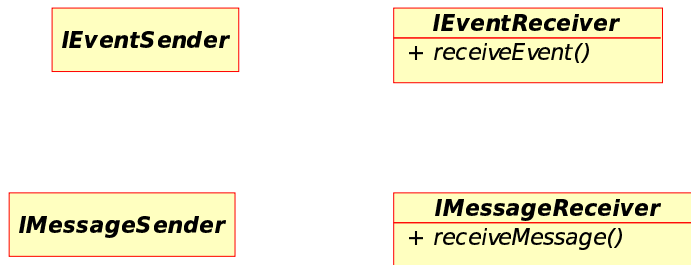


Figure 2.6: Interface classes that are used to identify the different types of interaction.

The abstract interface classes `IEventSender` and `IEventReceiver` are used for *request-reply interaction*. When using request-reply interaction the event sender will block until the processed event is returned. If *one-way interaction* is desired, the classes `IMessageSender` and `IMessageReceiver` need to be used. Sending a message is a non-blocking operation. Thus, the sender will not wait for any replies.

3. Dispatchers that will dispatch events to all registered listeners. Within NMM the class `EventDispatcher` implements such a dispatcher. It makes it possible to register or remove event handlers. For registering an event handler a *template event dispatcher object*, so-called *TED objects*, is used. These TED objects contain all data necessary for dispatching an event. Therefore, the TED object needs to know what object and what method of that object has to be called. For more details see [Loh05b, page 114].

Notice that multimedia data exchange (using buffers) uses instream communication, since it is communication between two nodes. Furthermore, it is one-way interaction, because the sender of the data does not require any reply from the receiver. Events are either sent between nodes (instream), or from application to node (out-of-band). The type of interaction used for this communication depends on the kind of the event. Some events require a reply from the receiver (here request-reply interaction is used), some do not (here one-way interaction is sufficient). In essence, all kinds of communication can be combined with all types of interaction.

2.1.8 Communication Architecture

Since NMM makes it possible to distribute nodes across the network, providing a communication architecture to accomplish communication between the distributed components is necessary. Therefore, NMM uses the proxy design pattern as defined in [GHJV94]. When using a proxy object, method invocation and execution is separated. As a result, the user of such a proxy object does not need to know on which host the underlying node is running, only the proxy object needs this information. For further details on proxy objects in NMM see [Loh05b, page 93], and [LRS02].

NMM uses so-called *communication channels* as abstraction for all communication between NMM components [Loh05b, page 95]. For example, the output jack of a node uses such a communication channel to communicate with the input jack of the successor node and vice versa (see Figure 2.7 on the following page). Figure 2.7 on the next page also shows that even proxy objects communicate with their underlying objects by means of a communication channel. All sum up, a communication channel is responsible for all data exchange between two partners. If the partners are distributed, the communication channel needs to use the network to achieve this communication. Therefore, all data has to be serialized, sent across the network and then be deserialized again.

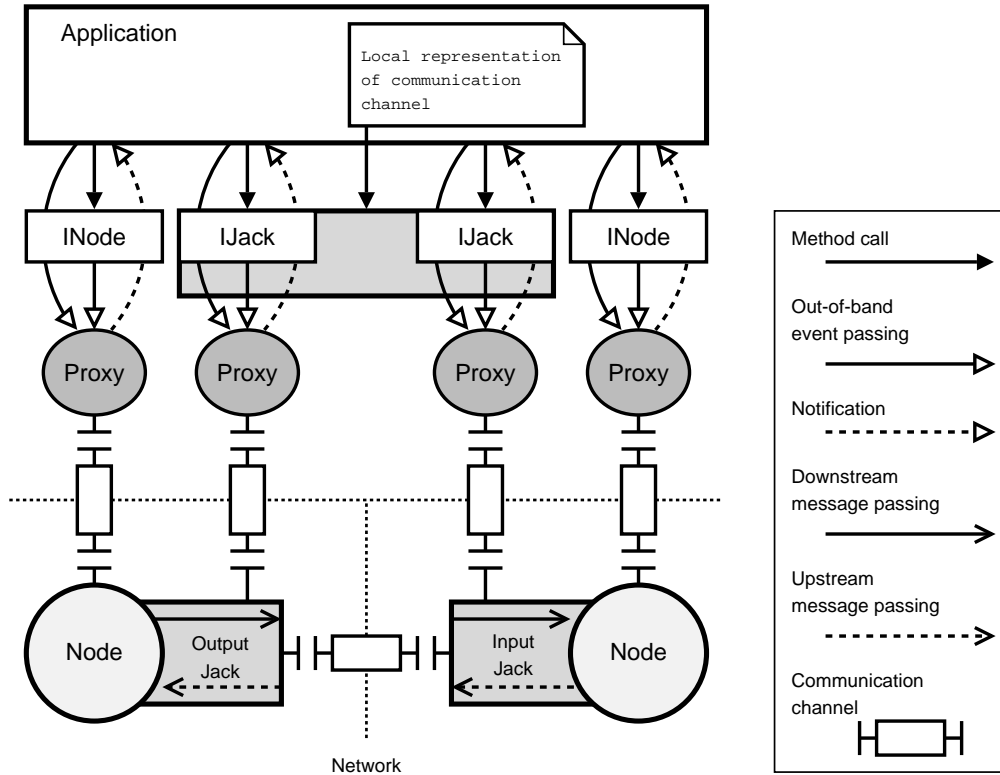


Figure 2.7: A distributed flow graph showing architectural elements [Loh05b, page 97].

2.1.9 Binding Framework

As stated before, a part of the requirements middleware has to comply with is the binding of distributed components. Communication channels have already been briefly introduced as abstraction of bidirectional binding between two components in NMM (see the Subsection on the communication architecture 2.1.8). An example of such a binding is shown in Figure 2.8 on the following page.

Transport strategies are used for transmitting and receiving objects. An integral part of a transport strategy is the serialization of the objects to be sent across the network. Therefore, a serialization stack, consisting of several *serialization strategies*, is used. In general, the serialization framework of NMM has to meet the following requirements [Loh05b, page 101]:

- Serialization of basic C++ data types.
- Serialization of new, complex data types.

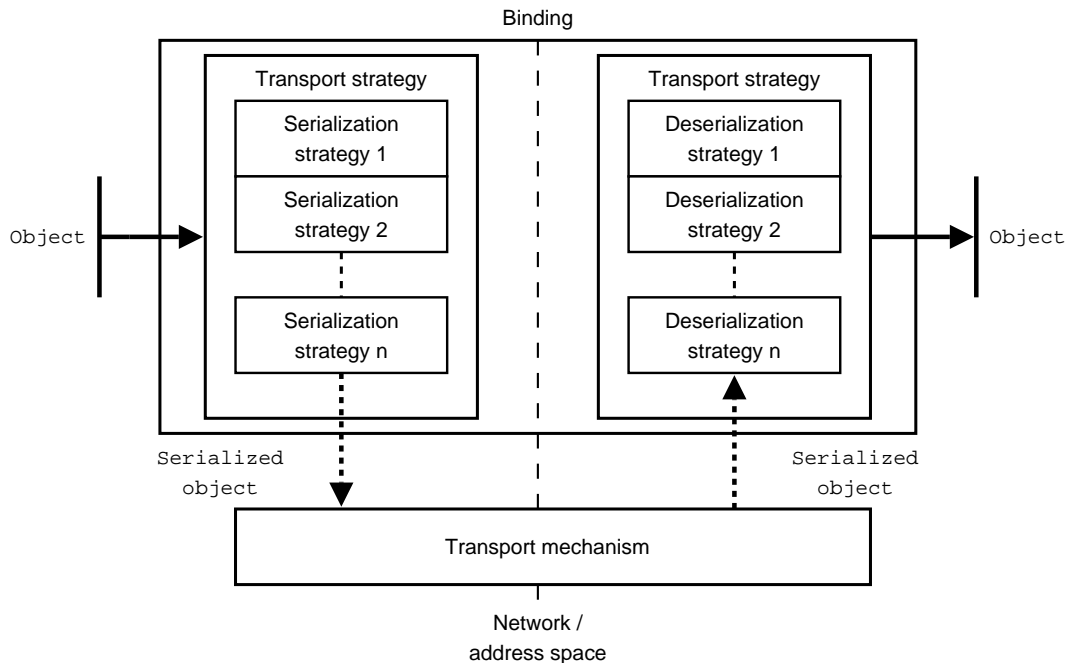


Figure 2.8: An example of binding two objects [Loh05b, page 127].

- Serialization of existing data structures that do not allow modifications.
- Providing the possibility of exchanging serialization strategies.
- Enable stacking of serialization strategies, in order to make decomposing of complex data structures possible.
- The serialization process needs to provide sufficient efficiency, concerning the time needed for serializing and deserializing, and the amount of data generated.

The *iostream library*, a part of the C++ standard library [Str97], with its stream concept is used in the serialization framework. In more detail, the output operator `operator<<` is used for serialization; the input operator `operator>>` is used for deserialization. NMM provides several different serialization strategies, like serializing an object into a XML representation. Further information about the serialization framework can be found in [Loh05b, page 101].

As shown in Figure 2.8, two transport strategies are needed to form a communication channel. Depending on the transport mechanism used, NMM

provides several transport strategies: TCP transport strategy, UDP transport strategy, RTP transport strategy, and local transport strategy. While TCP, UDP and RTP transport strategies obviously use the network as transport mechanism, the local transport strategy only forwards references. This keeps communication of two components, located on the same host, efficient.

Since NMM offers different types of communication—out-of-band and in-stream (see the Subsection on the messaging system 2.1.7)—different transport strategies, so-called *composite strategies* are used. Therefore, NMM provides the classes `OutOfBandCompositeStrategy` and `InstreamCompositeStrategy` [Loh05b, page 129]. Furthermore, a composite strategy needs to provide at least one kind of interaction role—e.g. request-reply interaction or one-way interaction.

When an application invokes a method of a client-side interface class, a corresponding event is created and sent to the proxy object. This event is then forwarded to the composite strategy. Inside the composite strategy, the transport strategy serializes the event and sends it to the transport strategy of the receiving composite strategy. There the data is deserialized and the restored event is sent to the node. If request-reply interaction is used, which is very likely for events, the node will generate a reply which will be sent back. This process works similar to the process of sending an event, as described above. Notice that out-of-band communication is used for this kind of communication, since it is application-to-node communication.

Within NMM, multimedia data and control information can be exchanged between nodes. Messages or events can arrive at a node concurrently; furthermore, in order to achieve a certain *Quality of Service (QoS)*, different transport protocols have to be used for different kinds of data. NMM uses a concept called *parallel bindings* to comply with these requirements [RWLS05], and [Loh05b, page 133].

2.1.10 Interfaces

As already mentioned in the paragraph in Nodes in Subsection 2.1.2, all the functionality of a node is specified by interfaces. These interfaces are defined with the help of the *interface definition language (IDL)*. A code generation tool can automatically produce a client-side *interface class* and a server-side *implementation class* from an interface defined with this IDL. The implementation class has the suffix *Impl* appended to the interface name. A node has to subclass all implementation classes of the corresponding interfaces the node wants to provide. The interface class is used by an NMM application, for example, to invoke a method defined by the interface. This method invocation of the interface class will result in an appropriate event being created

and sent to the proxy object, which will forward the event to the implementation class by using out-of-band communication (see Figure 2.9). The class `Interface` acts as base class for all client-side interfaces. A very detailed description of interfaces and the IDL of NMM can be found in [Loh05b, page 120], and [LRS02].

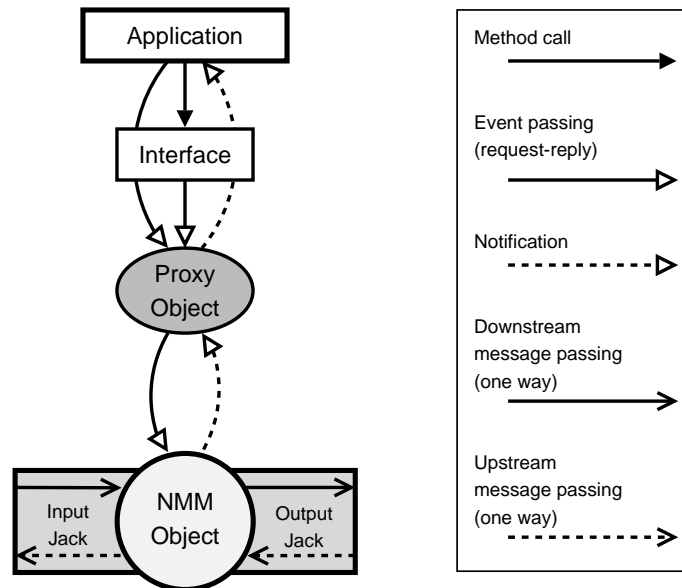


Figure 2.9: NMM IDL interfaces in the NMM architecture [Loh05b, page 125].

Notice that when an NMM object, e.g. a Node, is instantiated the programmer never gets direct access to this object. Since the communication between application and the node itself is out-of-band communication, all communication is done with a proxy object (see Figure 2.9). After a Node has been properly created, the programmer gains access to this node with a client-side interface with the name `INode`. Basically, this interface allows the programmer to control the nodes states. In order to use the other interfaces of a node, all interfaces derived from `Interface` can be queried for other interfaces by using the method `getInterface<IMyInterface>()`. This method returns a pointer to an object of the client-side interface class. Any methods from that particular interface can then be invoked.

Assume a Node has been properly created. How this is done is addressed in Section 2.1.12.

```
1  INode* node = //create node
```

Now `node` can be queried for other interfaces.

```
2  ISomeInterface* interface = node->getParentObject()
3                               ->getInterface<ISomeInterface>();
```

Then any methods of the interface `ISomeInterface` can be called.

2.1.11 Synchronization

When rendering audio and video data streams, they need to be synchronized. As stated in [Loh05b, page 30], synchronization consists of two parts.

Intra-stream synchronization: Media streams usually have a defined data rate, e.g. 25 frames per second in a PAL video stream [pal98]. For the example given, the intra-stream synchronization has to make sure that one video frame is displayed for exactly 40 msec. Therefore, the goal of the intra-stream synchronization is to ensure the defined data rate of the rendered stream.

Inter-stream synchronization: The inter-stream synchronization handles the chronological offset between different media streams. This comprises not only inter-stream synchronization of an audio and a video stream but also inter-stream synchronization of multiple (distributed) streams of any type.

While the intra-stream synchronization is addressed in the decoding standard used (the codec), the inter-stream synchronization is file-type-dependent. Even though everything necessary to achieve synchronous audio and video playback is present in the data used, this additional information still has to be properly used to actually gain synchronous playback. However, since NMM makes it possible to distribute nodes across the network, it has to provide the means to even synchronize distributed sink nodes as well. This makes the synchronization process much more complex since in a network data can be delayed or lost.

A basic requirement of NMM synchronization is that the system clocks of all participating hosts are running synchronously. This can be achieved by the *Network Time Protocol* (NTP) [ntp], and [Mil92]. NTP provides sufficient precision to accomplish audio and video synchronization. Of course, the system clocks of all hosts involved can not be completely synchronous, but it is sufficient for human eyes and ears not to recognize any asynchrony. Since the system clocks of all participating hosts act as the basis for sink synchronization, the quality of this synchronization depends on the accuracy of the system clock synchronization. Unfortunately, not every system clock

synchronization means can provide this accuracy. However, only offsets of a few ms between system clocks can be tolerated.

As stated before, all necessary information needed for intra-stream and inter-stream synchronization is usually present in the data stream, but it has to be extracted. As the data flows through the flow graph, more and more information about the data becomes available. Thus, at some point, all information required for synchronization is known. At least those buffers reaching the sink nodes provide a timestamp to make synchronisation possible.

Within NMM, every synchronized node has its own *controller* to control if and when data will be rendered [Loh05b, page 179]. In order to achieve intra-stream synchronization, the controller simply has to ensure that the right amount of data is sent to the rendering device at the right time. The controller of a video sink that renders a PAL video stream has to make sure that 25 pictures per second are delivered to the graphics card [pal98].

Gaining inter-stream synchronization is a little more difficult. Thus, all sink controllers need to be synchronized. This is done by the *sink synchronizer*, see Figure 2.10 on the following page. When rendering a movie featuring talking humans, it is of great importance that the audio and the video stream are *lip-synchronous*. According to [SN95], only offsets of up to ± 80 ms between audio and video playback are tolerable for human observers.

Another scenario that the inter-stream synchronization is responsible for is playback of audio on multiple hosts, for example. Whenever more than one sink needs to be synchronized, a master/slave approach is used to achieve proper synchronization. [Loh05b, page 185] “... the controller of the first audio stream is chosen as master; all other controllers act as slaves that have to adjust their playback speed to the master.” The reason why the first audio controller is chosen, lies in the fact that gaps in the audio playback are more disturbing than gaps in the video playback. Therefore, letting the other streams adjust to the audio stream results in a better synchronization as experienced by human observers.

2.1.12 Application Development

A local flow graph

The following example shows how the flow graph as shown in Figure 2.4 on page 9 can be created. At first, `NodeDescriptions` for the used nodes need to be created. Then their interconnections have to be defined.

```
1 GraphDescription graph;
```

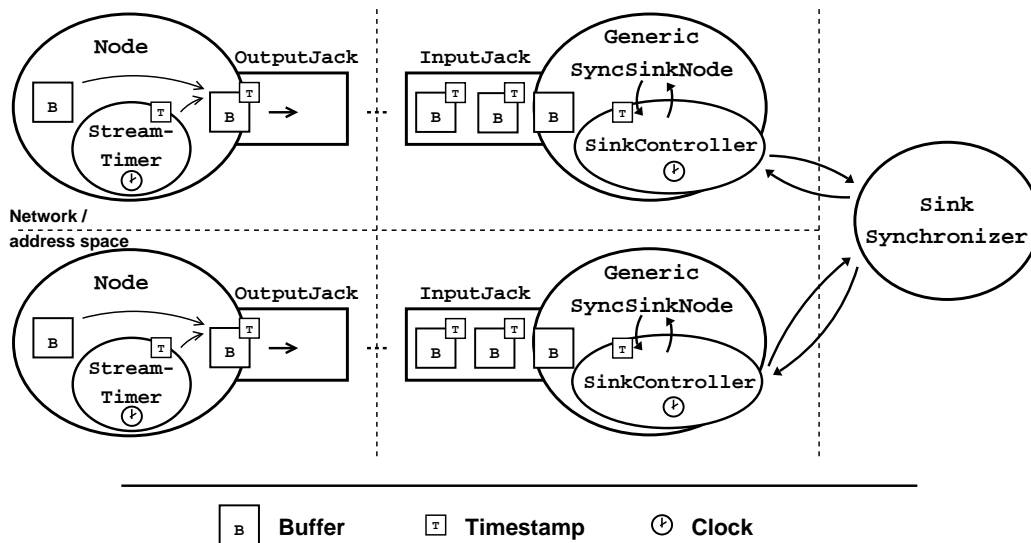


Figure 2.10: Distributed synchronization [Loh05b, page 182].

```

2
3  NodeDescription readfile_descr("GenericReadNode");
4  NodeDescription decoder_descr("MPEGAudioDecodeNode");
5  NodeDescription audioplay_descr("PlaybackNode");
6
7  graph.addEdges(&readfile_descr,
8                &decoder_descr,
9                &audioplay_descr);

```

Before any nodes can be requested, a reference to the registry client (see Section 2.1.5) is needed. Once this reference is obtained, the graph and its contained nodes can be requested.

```

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

```

When all nodes are requested, the file name of the source file can be set, as shown in the following.

```

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

```

The following commands will configure the flow graph and start it.

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

Stopping and releasing the graph requires these two lines.

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

The complete source code of this application, called *helloworld* is available as part of the distribution, or online at [wga].

A distributed flow graph

In order to create the distributed flow graph, as shown in Figure 2.5 on page 10, the last example has to be only slightly adapted. When creating the `NodeDescriptions`, additionally set the desired host the node should run on.

```
1  GraphDescription graph;
2
3  NodeDescription readfile_descr("GenericReadNode");
4  readfile_descr.setLocation("host2");
5  NodeDescription decoder_descr("MPEGAudioDecodeNode");
6  decoder_descr.setLocation("host3");
7  NodeDescription audioplay_descr("PlaybackNode");
8  audioplay_descr.setLocation("host3");
9
10 graph.addEdges(&readfile_descr,
11               &decoder_descr,
12               &audioplay_descr);
```

2.1.13 Graph Builder

NMM provides a class called `GraphBuilder` [LS05]. Its purpose is to build up the whole flow graph dynamically, based on a given URL. Additionally, the desired sinks and the synchronizer have to be specified. A list of currently supported URLs can be found online at [wgd].

How to Use the GraphBuilder

First, the `NodeDescriptions` for the sink nodes have to be created.

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

Since audio playback and video display need to be synchronized, a synchronizer object is required.

```
4  MultiAudioVideoSynchronizer av_sync;
5  IMultiAudioVideoSynchronizer_var sync(
6  av_sync.getCheckedInterface<IMultiAudioVideoSynchronizer>());
```

Create a `GraphBuilder` object and set the URL.

```
7  GraphBuilder2 gb;
8  if(!gb.setURL(argv[1])) {
9      throw Exception("Invalid URL given");
10 }
```

The previously created synchronizer object, as well as the desired sink nodes, can now be set.

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

These last two lines will finally create the graph and start it.

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

Audio and video can be easily redirected to a remote host by simply explicitly setting a location at their corresponding `NodeDescription` using the `setLocation()` method. Moreover, even the source can be located on a remote host by using a proper URL. The `GraphBuilder` is smart enough to intelligently distribute all needed nodes on suitable hosts so that unnecessary network abuse can be prevented.

2.2 Phonon

Phonon is the name of the multimedia API that will be available as part of KDE 4 [KDEb]. The leader of the Phonon project is Matthias Kretz. He has been a KDE developer since 2000. During his involvement in the KDE project he has worked on several components of KDE. Further information on Phonon can be found in [Kre], or [FKRL06]. The origin of the name Phonon is addressed in [Kre] as well:

Phonon is a name for quasiparticles in quantum mechanics that describe a quantized mode of vibration in a solid. Oversimplified, phonons are sound-particles. As there also exist optical phonons the name is also distantly related to the video part of multimedia.

2.2.1 Motivation

Since KDE 2, the *aRts* project (see [Wes]) has been used as soundserver and media framework. Even though aRts provides a lot of useful features, it was designed for sound playback and sound related application only. However, video support for some codecs was added later on. Additionally, there are some more reasons why aRts needed to be replaced for the upcoming KDE 4.

aRts has officially not been maintained since Decemeber 2nd, 2004. Because aRts is a quite complex piece of software, it is hard to comprehend the code. Therefore, it is very hard for new developers to gain a certain understanding of the internals of aRts. Thus, there are no new main developers and the project is not maintained now. Furthermore, application development and debugging using aRts is not as simple as it could be.

The objective while designing Phonon was not only to provide a replacement for aRts, but also to provide a full-featured multimedia API. To avoid to depend on one single multimedia framework, so-called *backends* are used to do the actual work. Phonon itself only defines the API. Thus, every available multimedia framework can implement such a backend.

Phonon allows KDE developers to add multimedia features to their application in a fraction of the time that would be needed without it. Prior to Phonon a KDE application developer had to implement such features almost from scratch, or use a complex multimedia framework. In order to really offer an advantage over already available frameworks, the Phonon API has to be easy to use and fast to learn for simple as well as complex scenarios. Furthermore, the API has to offer sufficient functionality for as much different kinds of KDE applications as possible. Pro-audio applications can not be fully supported by Phonon. This would make the API too complex.

Additionally, the Phonon API is designed and implemented in KDE/Qt style to seamlessly fit into the rest of the KDE code. Thus, the developer does not have to work with non-KDE/Qt formatted code, or programming paradigms.

2.2.2 Concept

As already mentioned in Subsection 2.2.1, the concept needs to be easy to understand and to use, while providing sufficient flexibility. Otherwise, users will not accept and use the Phonon API.

Core Classes

Figure 2.11 gives an overview of the core classes of Phonon.

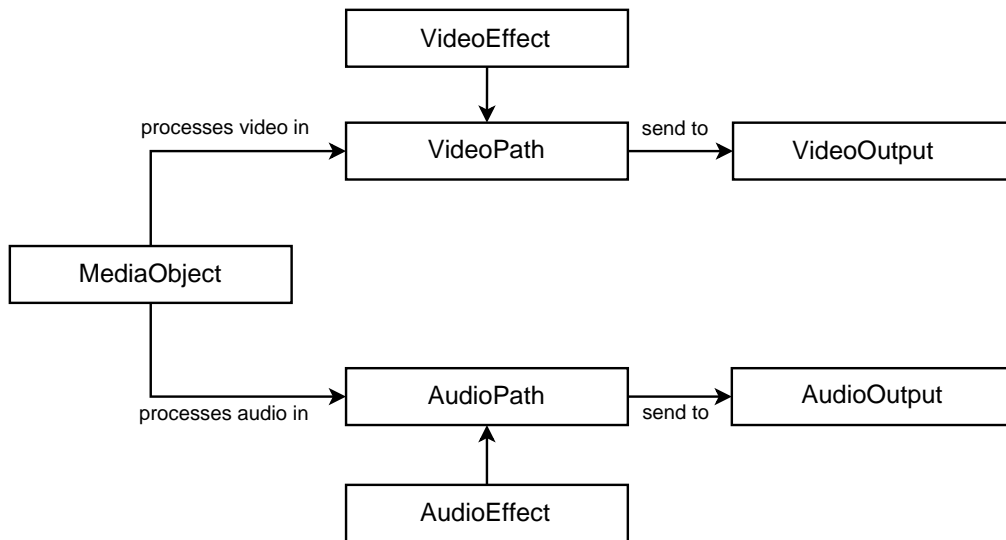


Figure 2.11: Core classes of Phonon.

MediaObject: The class **MediaObject** represents the multimedia data itself. Therefore, it allows the user to query for available audio stream or stream names, for instance. The method `setUrl` makes it possible to define the source of the media data. Furthermore, the **MediaObject** provides common methods used for playback, like `play()`, `pause()` or `stop()`.

AudioPath: With the help of the method `addAudioPath` a **AudioPath** can be added to a **MediaObject**. The **AudioPath** provides all means to

define a filter/effect chain. Additionally, a certain audio stream can be selected. This makes it possible to use various `AudioPaths` assigned to different audio streams with only one `MediaObject`.

AudioEffect: Audio filters/effects are represented by the class `AudioEffect`.

AudioOutput: As the name already suggests, the class `AudioOutput` represents devices that are capable of audio rendering.

VideoPath: `VideoPath` are used for similar purposes as `AudioPath`, except that they are used only for video data.

VideoEffect: This class represents video filters/effects.

VideoOutput: Video rendering devices are represented by the class `VideoWidget`. Since `VideoWidget` is derived from class `QWidget` provided by Qt, it allows a developer to easily integrate this widget into his Qt based GUI. For further details on Qt see [BS04], [Troa], or [Trob].

To sum up, everything a programmer needs to do, in order to implement media playback with the Phonon API, is creating a `MediaObject`, set a proper URL, add desired paths and outputs, and finally call `play` on the `MediaObject`. A more detailed example is given in Subsection 2.2.3.

Notice that the number of paths is not restricted. Phonon allows the programmer to add numerous `AudioPaths` or `VideoPaths`.

Phonon supports much more than simple playback only. It can be used for capturing as well. However, this part of Phonon is still under development.

Backends

Backends are used to provide the functionality defined by the API. Figure 2.12 on the following page gives an overview of the architecture of Phonon. Any multimedia framework that can comply with the requirements of Phonon can be used for implementing a backend. However, it is not that easy to provide the functionality and flexibility necessary. Since a major goal of Phonon is to make the use of its API as simple as possible, the usage is very less restrictive. In detail this means that a backend has to support at any time:

- Adding/removing of an arbitrary number of paths and outputs.
- Adding/removing of any number of filters/effects.
- Changing the assigned media stream of a path.

- Changing the URL of the `MediaObject`.

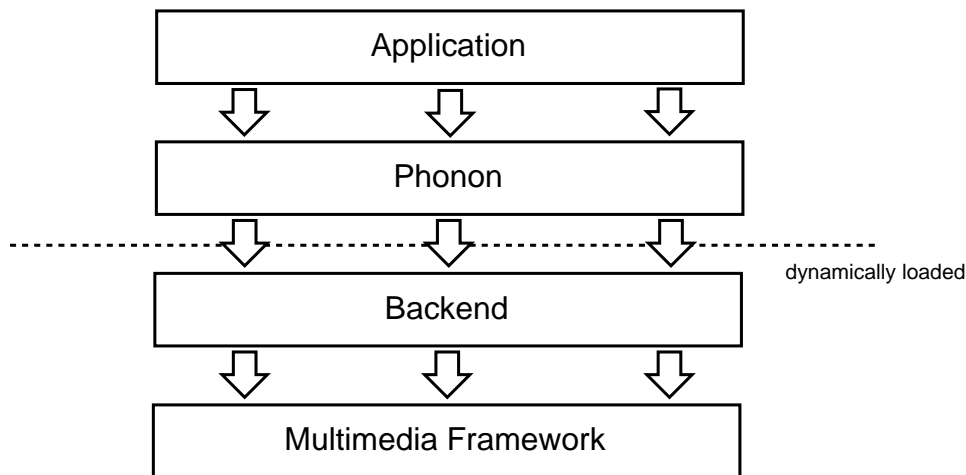


Figure 2.12: Architecture of Phonon.

The usage of backends brings, aside from being more media framework independent, some significant advantages. Because each backend is linked into a library, backends can be easily exchanged. Furthermore, the *application binary interface ABI* of the backend does not change, even if the version and ABI of the media framework used in a backend changed. This allows the Phonon user to always use the latest available version of the framework, without the need to recompile the application. Only the backend needs to adapt to the changes in the framework. Additionally, the KDE user can choose among all available backends and is not restricted to one single backend.

In order to make backend implementation easier, Phonon provides abstract base classes. All the backend developer has to do is to derive from those abstract base classes and implement all abstract methods. When this is done appropriately, the backend provides all the functionality as defined by Phonon. In essence, how the abstract methods need to be implemented depends on the media framework used to implement the backend.

2.2.3 Sample Application

The following code example shows how to use the Phonon API for simple MP3 playback.

```
1  output = new AudioOutput( this );
2  path = new AudioPath( this );
3  path->addOutput( output );
```

```
4  media = new MediaObject( this );
5  media->addAudioPath( path );
6  media->setUrl( "file:///home/user/myfile.mp3" );
7  media->play();
```

The code for playing a video file might look like the following.

```
1  output = new AudioOutput( this );
2  path = new AudioPath( this );
3  path->addOutput( output );
4  media = new MediaObject( this );
5  media->addAudioPath( path );
6  media->setUrl( "file:///home/user/myfile.mpeg" );
7  if( m_media->hasVideo() )
8  {
9      vpath = new VideoPath( this );
10     m_media->addVideoPath( vpath );
11     vout = new Ui::VideoWidget( this );
12     vpath->addOutput( vout );
13     vout->show();
14 }
15 media->play();
```

2.2.4 Current Status

Currently, Phonon as well as KDE 4 itself is still under development. However, all core classes needed for media playback are finished. For more up-to-date information about the status of Phonon see [Kre]. The release of KDE 4 (including Phonon) is scheduled for early 2007.

2.3 Summary

This chapter introduced selected aspects of NMM and Phonon. Phonon represents a high-level API which makes it possible for lower-level multimedia frameworks, like NMM, to implement a backend that provides the functionality as defined by the Phonon API. The emphasis of NMM lies in the integration and configuration of distributed components. Thus, it provides features that will not be available with plain Phonon. Since those features are likely to be useful for KDE users, additional facilities are required to make them available.

Chapter 3

Concept

The backend concept of Phonon (see the paragraph on backends in Subsection 2.2.2) makes it possible a multimedia framework to implement a backend. Of course, there are requirements such a multimedia framework has to comply with. However, developing a backend is the way to integrate NMM into Phonon.

As stated in Section 2.2, Phonon only defines the API for multimedia functionality. Therefore, it is up to the backend to implement this functionality compliant with Phonon. This means that an API call has supposed results, and the backend has to deliver exactly those results. The challenge when developing a backend is to provide the behavior als defined by Phonon. In general, this is not that easy to achieve, since the concepts are likely to differ. The flow graph of NMM with its nodes can not be directly assigned to certain parts of Phonon. Which means that, when a developer using Phonon creates a Phonon class or calls some methods of a Phonon class, this often can not be translated into a concrete node being instantiated, or into an NMM method call. Therefore, the backend can be seen as adapter (see adapter pattern [GHJV94]).

Section 3.1 will address *what* the NMM backend for Phonon needs to achieve. Additionally, Section 3.2 will introduce a concept of integrating a selection of the advanced features of NMM, that would not be accessible with the provided interfaces of Phonon, into the backend. Thus, the users of Phonon will have the possibility to take advantage of some of the unique benefits of NMM.

3.1 Backend Concept

Obviously, the major goal of the backend is to build up an appropriate NMM flow graph. As mentioned in Chapter 2.2, the API usage is quite less restrictive, which makes it easy to use for developers, but unfortunately this makes developing a backend even more difficult.

The following subsections will give an overview on what the backend implementations of the Phonon classes, as introduced in Section 2.2.2, have to accomplish.

3.1.1 MediaObject

This class makes it possible to define the data source and query for information about the data (number, type and name of available streams). When the data source is set, an NMM source node can be requested. However, the primary task of the source node is to provide data, e.g. by reading from the hard disk. Thus, the source node alone can not provide the necessary information about the data in some cases. In these cases, further nodes need to be connected in order to make this information become available.

If the media data only contains one stream, audio for instance, the source node can provide sufficient information about the data (see Figure 3.1). If there are more streams present (one video stream and one or more audio streams is quite common), a demultiplexer node is required (see Figure 3.2 on the following page). Since the primary objective of this node is to provide the data of the streams independently, the node needs to gather information about the available streams. This information can be used to finally provide the information as required by the `MediaObject`.

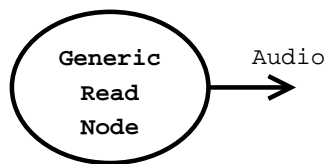


Figure 3.1: A partial flow graph for MP3 playback.

Since the paths and outputs assigned to the `MediaObject` might still change, or there even might not be any paths and outputs be assigned yet, the whole flow graph can not be completely built up at this point.

All sum up, when an object of the type `MediaObject` is created and the URL defining the data source was set, a part of the NMM flow graph needs to be built up. If there is only one single stream present, the source node

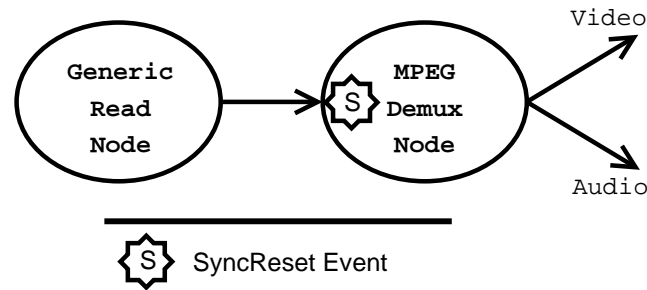


Figure 3.2: A partial flow graph for MPEG playback.

itself can provide sufficient information about the data. If there are more streams present, a demultiplexer node is necessary in order to provide this information.

Synchronization Issues

When paths and outputs are added, more NMM nodes need to be connected to the partial flow graph that has been built up at this time. The easiest thing to do would be to connect corresponding nodes, at the time a path and its output are added. Unfortunately, this can not be done.

The reason why the flow graph can not be built up, when a path and its output is added, lies in the synchronization concept of NMM (see Subsection 2.1.11). The source node sends out a so-called *SyncReset* event. When such an event reaches a demultiplexer node, this event is copied and sent to all connected successor nodes, at the time the demultiplexer node reaches the state activated. These events will be forwarded until they reach the sink node. The synchronizer of the sinks expects exactly the same number of *SyncReset* events as the number of connected sink nodes.

Usually, paths and their outputs are added sequentially. Which means that when a path and output is added, it is uncertain if more paths and outputs will be added, or if some paths and outputs will be removed again. Unfortunately, this is a problem for NMM.

In order to build up a flow graph, a node has to reach the state activated before it can provide data for the successor node. The successor node usually needs data for analysis purposes during its *initOutput* state. This means that the demultiplexer has to reach the state activated if further nodes need to be connected.

Therefore, when adding a path, this would mean that an appropriate node would be connected to the demultiplexer. Furthermore, the demultiplexer would need to reach the state activated, to allow the successor node to reach

the state `initOutput`, which is mandatory for connecting further nodes to the successor node of the demultiplexer (see Figure 3.3).

This would not cause any troubles, but when additional paths and outputs are connected, it would require additional nodes to be connected to the demultiplexer node. Since the demultiplexer node has already been brought to the state `activated`, the `SyncReset` event will **not** be copied for the successor nodes of the demultiplexer that were added after the node reached this state for the first time (see Figure 3.4 on the following page). When the flow graph is finally started the synchronizer of the sinks will then block indefinitely, since it expects more `SyncReset` events than can possibly reach the sinks and therefore the synchronizer.

To avoid such thing to happen, all the successor nodes of the demultiplexer have to be connected before the demultiplexer can be brought to the state `activated` (see Figure 3.5 on the next page). Thus, the flow graph **must** be built up breadth first.

Unfortunately, it is uncertain if additional paths and outputs will be added or removed until the playback is started. Therefore, when the method `start()` of the `MediaObject` is called the first time, the rest of the flow graph will be built up.

Since this `SyncReset` event is only sent once at the very beginning of the process of building up a flow graph, adding or removing of paths and its outputs after the flow graph has been started will not lead to blocks of the synchronizer.

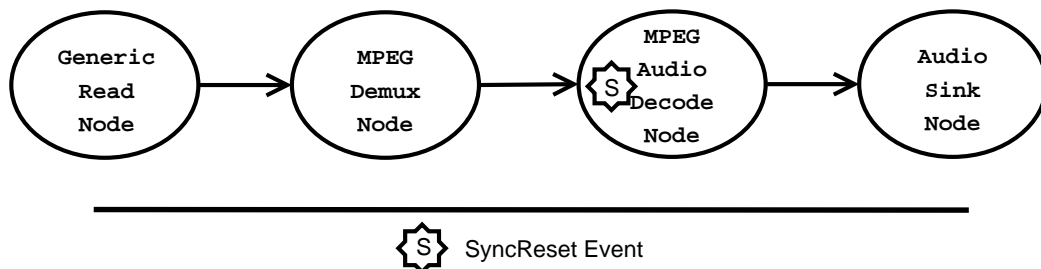


Figure 3.3: Step one. A `AudioPath` with its `AudioOutput` was added to the `MediaObject`. Hence, all nodes required for audio playback were added.

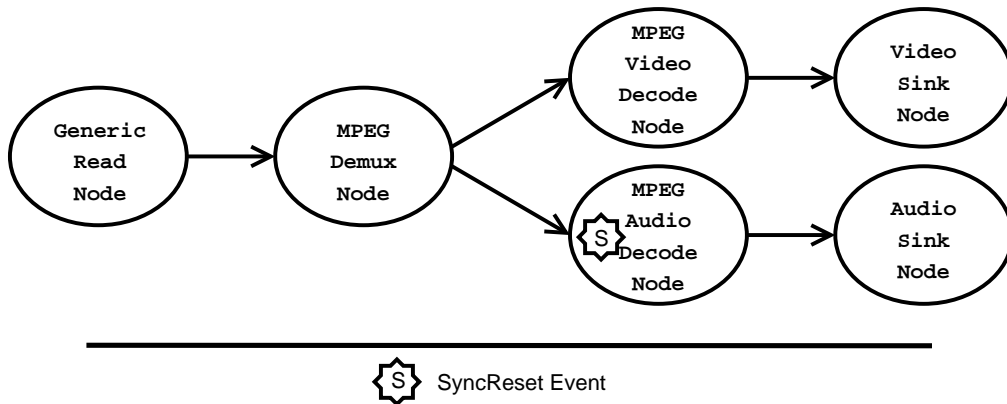


Figure 3.4: Step two. A VideoOutput with its VideoWidget was added to the MediaObject. Therefore, all nodes necessary to achieve video playback were added. Notice that the nodes required for audio playback have already been added. Thus, the MPEGVideoDecodeNode will not receive the SyncReset event, since the demultiplexer node has reached the state activated before the video decode node was connected.

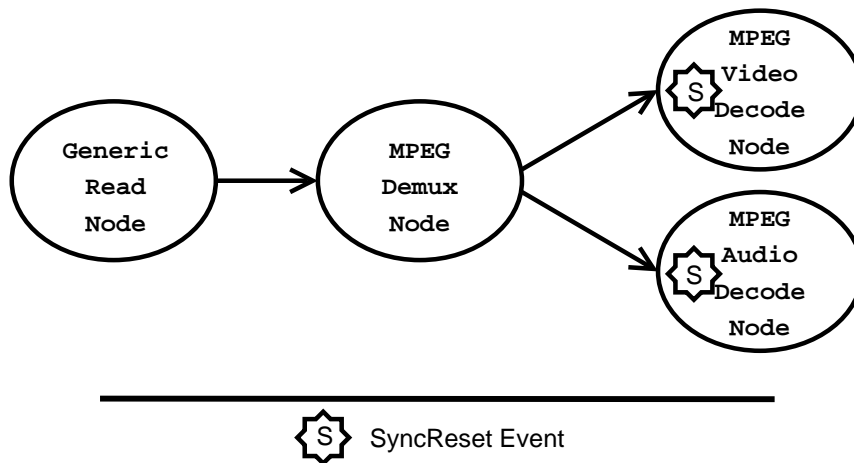


Figure 3.5: If all successor nodes of the demultiplexer were connected prior to bringing the demultiplexer node to the state activated, all successor nodes will receive a SyncReset event.

Examples

In Subsection 2.2.3, an example for MP3 playback and for MPEG video playback using the Phonon API was given. The following will show what the backend has to achieve when the code is executed.

MP3 playback:

```
1  output = new AudioOutput( this );
2  path = new AudioPath( this );
3  path->addOutput( output );
4  media = new MediaObject( this );
5  media->addAudioPath( path );
6  media->setUrl( "file:///home/user/myfile.mp3" );
7  media->play();
```

In line 5 the `AudioPath` is added to the `MediaObject`. Since so far the URL was not set, no nodes were created. At this point no information about the data is available; hence, it is not possible to use the `MediaObject` to query for such information. In line 6 the URL is set; therefore, a source node can be created. For the most types of multimedia data the `GenericReadNode` is used as source node. This node can provide sufficient information about the media data; thus, no further nodes will be connected (see Figure 3.1 on page 29). Even though a `AudioPath` was added in line 5, the corresponding NMM nodes will not be added in order to avoid a possible loss of `SyncReset` events. Finally, in line 7 the `MediaObject` is started. Now the rest of the flow graph can be built up, as shown in Figure 2.4 on page 9.

MPEG playback:

```
1  output = new AudioOutput( this );
2  path = new AudioPath( this );
3  path->addOutput( output );
4  media = new MediaObject( this );
5  media->addAudioPath( path );
6  media->setUrl( "file:///home/user/myfile.mpeg" );
7  if( m_media->hasVideo() )
8  {
9      vpath = new VideoPath( this );
10     m_media->addVideoPath( vpath );
11     vout = new Ui::VideoWidget( this );
```

```
12     vpath->addOutput( vout );
13     vout->show();
14 }
15 media->play();
```

There are no differences between both code examples until line 6. Thus, there are no differences in the backends behavior either. In line 7 the URL is set. Since the URL is pointing to an MPEG encoded file with multiple media streams present, creating a `GenericReadNode` only is not sufficient in order to provide the information about the media data. Therefore, a demultiplexer node has to be connected (see Figure 3.2 on page 30). This makes it possible to query the `MediaObject` whether it can provide video data (c.f. line 7). If so, a `VideoPath` will be created and added to the `MediaObject`. For the same reasons as stated above, the rest of the flow graph will not be built up until the playback is started, which is done in line 15.

3.1.2 AudioPath and VideoPath

The primary objective of the paths is to provide a filter/effect API. Additionally, the `AudioPath` allows the developer to select a certain audio stream (see the paragraph on the core classes of Phonon in section 2.2.2).

Filters or effects do not change the format of the data, they change the data. Furthermore, filters/effects only work with already decoded data. Therefore, the most suitable place for them in a flow graph would be as predecessor nodes of the sink nodes, since sinks usually only work with decoded data as well (see Figure 3.6).

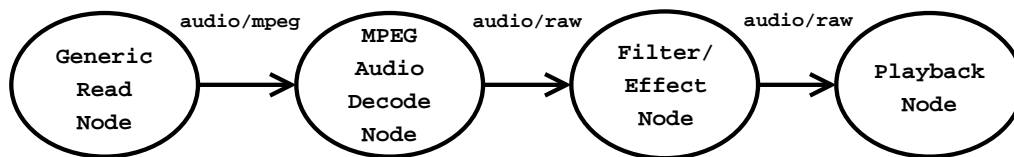


Figure 3.6: A flow graph containing one filter/effect node.

However, filters/effects can be added or removed at any time. Thus, if the flow graph has already been built up, nodes need to be disconnected and the filter/effect node has to be inserted (see Figure 3.7 on the following page).

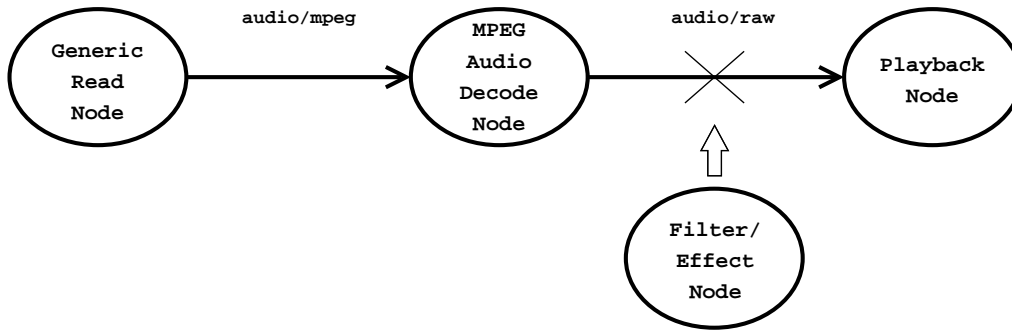


Figure 3.7: In order to add a filter/effect node to an already completely built up flow graph, the sink node needs to be disconnected from its predecessor node. Then the filter/effect node can be inserted.

3.1.3 AudioOutput and VideoOutput

The class `AudioOutput` makes it possible to set the desired output device and adjust the volume. Therefore, an appropriate sink node needs to be used. If the output device changes, the sink node used has to adapt to those changes, or be replaced by another sink node.

Currently, the `VideoOutput`, represented by the class `VideoWidget`, only allows the developer to activate/deactivate the fullscreen mode. These calls can be forwarded to the sink node used for rendering the video data.

3.2 Integration of Advanced Capabilities of NMM

NMM is a very powerful multimedia framework that offers unique features such as its versatile network integration and configuration. With the help of NMM, very complex scenarios can be created, with very little code.

However, Phonon aims to provide an API to aid desktop application development within the KDE environment. Since the Phonon API is designed to be used for desktop application development only, it can not reflect all of the special features of NMM. Otherwise the Phonon users might be overwhelmed by the complexity of such an API. Furthermore, the requirements for a Phonon backend would be quite difficult to comply with for other multimedia frameworks. The integration of the advanced capabilities of NMM into the Phonon-NMM backend is also addressed in [FKRL06].

3.2.1 Goals

NMM offers various features that might be useful for KDE users, even though they are not available by means of the Phonon API. Therefore, another way of making those features available to the user is required. However, Phonon itself already allows usage of a very useful NMM network feature. Since the `MediaObject` of Phonon 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.

The goal of the integration of the advanced capabilities of NMM into its backend is to provide facilities to distribute the flow graph across the network (especially the sink nodes) based on the user's input. In detail, this means that it should be possible for the user of any KDE application that uses Phonon and the NMM backend for Phonon to specify additional host for audio and/or video playback. As mentioned in Subsection 2.1.11, NMM already provides the means to synchronize the playback of audio and video on multiple, distributed hosts.

Notice that NMM offers a lot more features that are worth integrating into the backend (like *Session-Sharing* [LRS03b], or *Seamless Handover* [LRS03a]); however, at this point of development it was decided to start with this selection of capabilities of NMM.

3.2.2 Configuration Concept

In order to make the designated additional NMM features available to the users of Phonon-based applications, implementing a *DCOP* (Desktop COmmunication Protocol) interface was chosen. DCOP is the protocol for inter-process communication (IPC) used by KDE, see [BE99] for an introduction to DCOP and some example implementations. This interface makes it possible to overcome the limitations of Phonon and offer more of the functionality of NMM to the end-user. Figure 3.8 on the next page gives an overview of how the interface is integrated into the NMM backend.

Using a DCOP interface allows end-users to get an access to the advanced capabilities of NMM, but the KDE developers that use Phonon for implementing their applications do **not** need to add extra, NMM-backend-specific code to their application. Otherwise this would by-pass the media framework independence which is one of the benefits of the backend concept of Phonon (see Subsection 2.2.2). However, the fact that a DCOP interface was chosen is no restriction to the ways of application. Moreover, it allows any DCOP

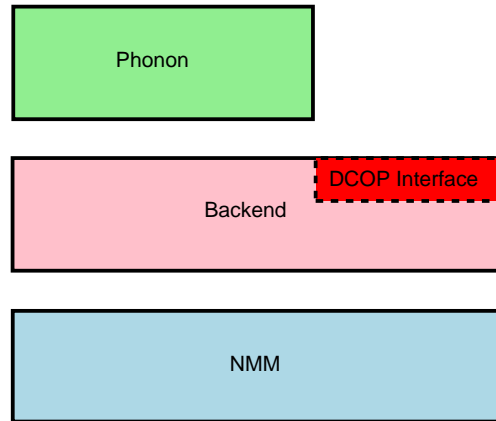


Figure 3.8: Architecture overview.

integrated application to use this interface. As part of this work a sample configuration application, that uses this DCOP interface, was implemented (for further details see Section 4.2).

3.3 Summary

The core-issue for implementing a backend based on NMM is to build up an NMM flow graph as required by Phonon. Section 3.1 introduced a concept of how this can be done.

Finally, in Section 3.2 the concept of how a selection of the advanced, NMM-specific features of NMM can be integrated into the NMM backend for Phonon. The challenge with this integration is that it should not eliminate the backend independence of Phonon. Therefore, it must not require any NMM-specific code to be added to Phonon-based applications.

Chapter 4

Implementation

Chapter 3 addressed *what* the NMM backend for Phonon has to accomplish and *what* needs to be done in order to integrate certain advanced features of NMM into the backend. Therefore, the main focus of this chapter lies in *how* these things can be achieved.

Section 4.1 will introduce the class `GraphHandler` which is the class responsible for creating and manipulating the NMM flow graph. Details on *how* the goals of the integration concept for the advanced capabilities of NMM are achieved can be found in Section 4.2.

4.1 GraphHandler

NMM provides various means to build up a flow graph.

- An application can be implemented that creates and connects all required nodes (see Subsection 2.1.12). Since the number and kind of the nodes needed to playback a certain file depends on the nature of the data (e. g. the encoding), the developer has to know certain properties of the data in order to have the necessary information to request and connect the proper nodes. If more than one audio stream is present, a particular stream needs to be selected as well. Hence, such programs are not flexible concerning different media data compositions. However, by implementing an NMM application, very complex scenarios can be created.
- Furthermore, NMM makes it possible to specify the nodes of a flow graph and how they are interconnected in a text-based file called *graph description file*. Even though an application can use such files to build up a flow graph, these files are not very flexible either, since the kind

of nodes as well as their interconnections are statically defined in the graph description file.

- To provide a way to dynamically (based on the nature of the media data) build up a flow graph the class **GraphBuilder** was implemented. How to use the **GraphBuilder** in an application is addressed in Subsection 2.1.13. This class offers a very easy, practical way to build up a flow graph. Most important is the fact that the user of the **GraphBuilder** does not need to know any details of the media data that is supposed to be played back.

What the NMM backend needs to achieve in order to implement an adequate backend for Phonon is addressed in Section 3.1. Even though the **GraphBuilder** is a very powerful mean of building up a flow graph, its functionality is insufficient to be used in the backend. The **GraphBuilder** allows the developer to specify one sink for audio and one sink for video playback, but the backend should make it possible to use multiple (remote) hosts for audio and video playback. Furthermore, the **GraphBuilder** is only used to build up the NMM flow graph, then the graph can not be manipulated anymore, but Phonon allows the developer to alter existing paths and outputs or add new ones at any time, which makes changing the NMM flow graph necessary. Therefore, in order to overcome the deficiencies of the **GraphBuilder**, the class **GraphHandler** was implemented.

4.1.1 Design and Implementation

The primary objective of the **GraphHandler** is to provide the functionality to create **and** manipulate a flow graph. As shown in Figure 4.1 on the following page, this functionality is used for implementing the **MediaObject** defined by Phonon. The tasks the **MediaObject** has to achieve are addressed in the paragraph on the core classes of Phonon in Subsection 2.2.2. However, building up a flow graph is a complex task. The difficulties that arise during the initial build-up of the flow graph have already been pointed out in Section 3.1.

The most important public methods of the **GraphHandler** are shown in Figure 4.2 on the next page. Notice that the **GraphHandler**, as well as the **GraphBuilder**, uses an URL to define the data source.

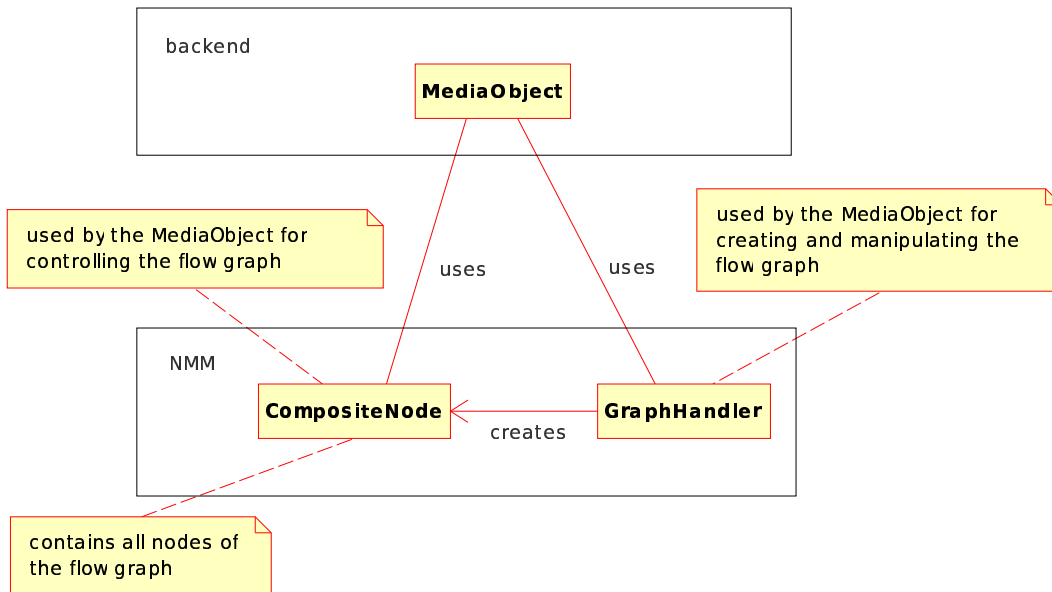


Figure 4.1: The MediaObject uses the GraphHandler for all flow graph management. The CompositeNode is used for controlling the flow graph (see the paragraph on the CompositeNode in Subsection 2.1.2)

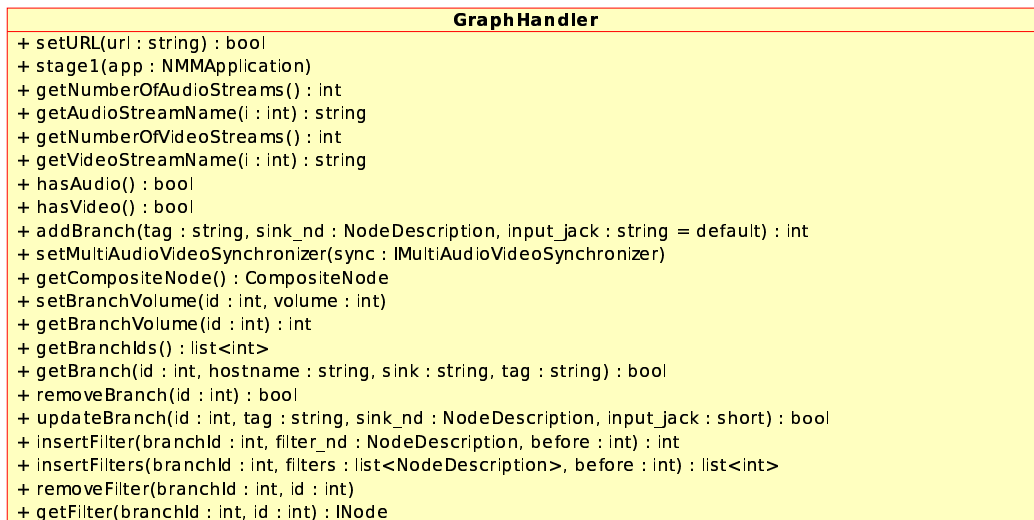


Figure 4.2: Some of the public methods of the class GraphHandler.

4.1.2 The Three-Stages-Concept

The lifecycle of a `GraphHandler` object is defined by three stages (see Figure 4.4 on the following page):

Stage 1: After the source of the data is set (by using the method `setURL()`), the `GraphHandler` can be brought to stage 1. This stage is reached when the format of all unconnected outputs have the type audio or video. If the media data only contains one stream, only a source node is required (see Figure 3.1 on page 29 for an example for MP3 data). If the data contains more streams, a demultiplexer is required (an example for MPEG data is shown in Figure 3.2 on page 30). At this point the `GraphHandler` can be queried for information about the media data. Additionally, all desired sink nodes can be specified now. How this is done is addressed in the following paragraph. However, the node that was connected last, is the so-called *stage-1-node* (see Figure 4.3). This node acts as base for all nodes that need to be connected later.

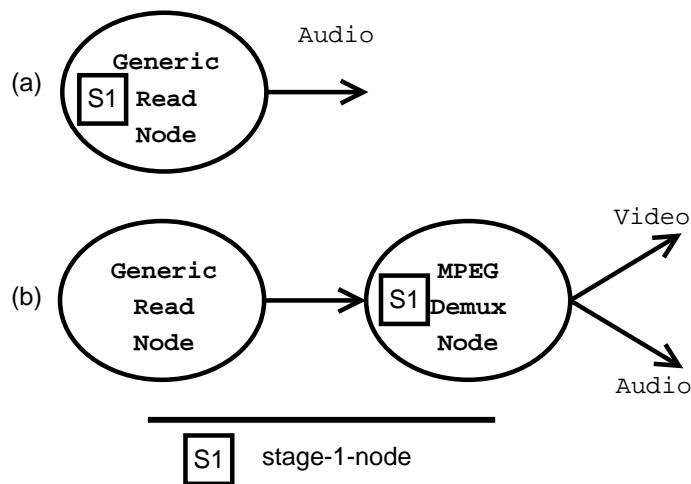


Figure 4.3: The flow graph to play back an MP3 file looks like (a) after the `GraphHandler` reached stage 1. (b) shows the partial flow graph for MPEG video playback at stage 1.

Stage 2: In order to reach stage 2, the `GraphHandler` has to connect further nodes. The defined output format of all unconnected output jacks when stage 2 is reached, is *raw*. Basically, this means that decoder nodes are connected. Since the media format of all outputs is raw, filter/effect nodes can be added now. Remember that all nodes connected to the stage-1-node need to be connected breadth first, in

order to avoid blocks of the synchronizer of the sink nodes (see Section 3.1).

Stage 3: Finally, when stage 3 is reached, all sink nodes are connected. Now the flow graph can be started.

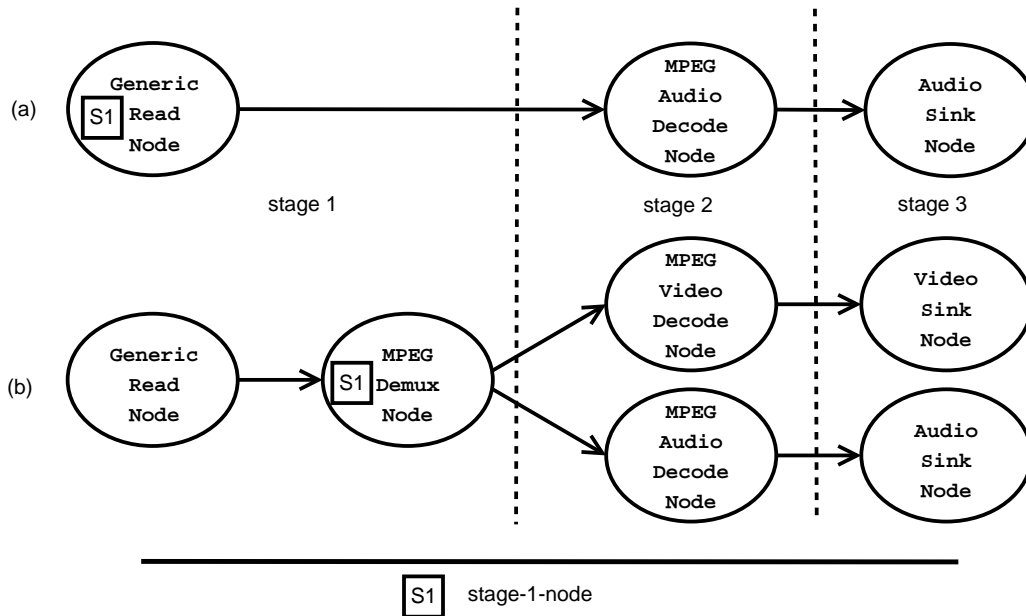


Figure 4.4: The flow graph used for MP3 (a) and MPEG video (b) playback.

Notice, that the `GraphHandler` can not be brought back one or more steps. Additionally, the URL can not be changed after stage 1 is reached. Thus, if resetting the URL is necessary, a new object of the class `GraphHandler` needs to be instantiated. The class `GraphHandler` does not offer methods to bring its object to stage 2 or stage 3. This is implicitly done when `getCompositeNode()` is called for the first time, since returning a partial flow graph does not make much sense. As stated in the paragraph on the `CompositeNode` in Subsection 2.1.2, the `CompositeNode` is used to register event handlers and to start the flow graph; thus, a completely built up flow graph is mandatory.

4.1.3 Specifying Additional Sink Nodes

In order to add a sink node to a flow graph, adding a so-called *branch* is required. A branch consists of all nodes that are needed to connect the

desired sink to the existing flow graph (see Figure 4.5). This can be done by using the method `addBranch()`.

```

1  int addBranch(const string& tag,
2                NodeDescription& sink_nd,
3                const string& input_jack = "default");

```

The parameter `tag` is used to select an output jack of the stage-1-node. With the help of the `NodeDescription` `sink_nd` not only the name of the desired sink node can be specified, but also the host on which this node is supposed to run on. Finally, the `input_jack` can be used to choose the input jack of the sink node, which is named *default* in most cases. Since the `GraphHandler` makes it possible to alter existing branches, the `addBranch()` method returns a unique ID, that can be used to identify a certain branch later on.

However, the `GraphHandler` is required to be at least in stage 1 when this method is called. Depending on the stage the `GraphHandler` is in, the tasks that have to be done when `addBranch()` is called differ. If the `GraphHandler` is in stage 1, only the sink node is requested. The rest of the information of the branch that was provided with the method call (like the `tag`) will be stored internally. When the `GraphHandler` is brought to the next stages, this information is used to request and connect proper nodes.

If the `GraphHandler` is already in stage 3, all nodes required to connect the sink node to the stage-1-node will be connected immediately. Notice, that the flow graph had to be started at least once for this to work; otherwise, the synchronizer of the sinks will block. Since the stage-1-node already reached the state activated, no `SyncReset` event can reach the sink of the new branch (see Section 3.1).

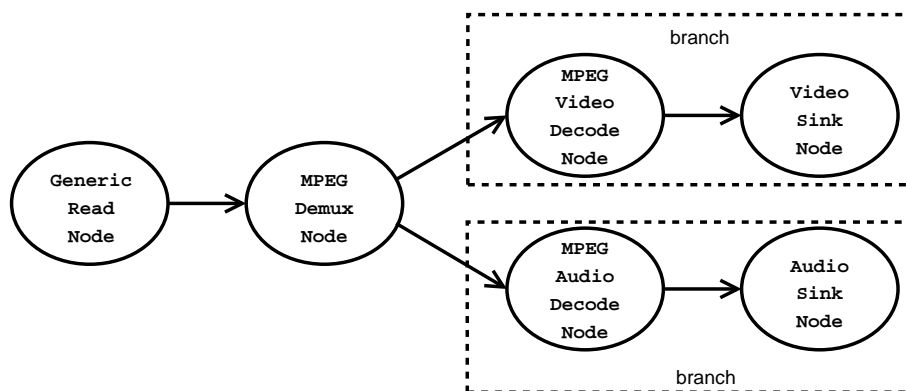


Figure 4.5: A flow graph used for MPEG video playback.

Since the sink node specified at the `addBranch()` method call might be located on a remote host, the `GraphHandler` has to properly distribute the nodes of a branch (see Subsection 2.1.3). Furthermore, the source node can be located on a remote host as well. Thus, the `GraphHandler` also has to deal with such scenarios. To avoid unnecessary network bandwidth usage the sink node, the filters/effects, and the converter node are located on the same host. This means that all nodes of a branch run on the host specified for the sink node. If a demultiplexer node is required, it is located on the same host as the source node.

4.1.4 The Filter/Effect API

The class `GraphHandler` provides three methods that allow specifying and manipulating a filter/effect chain for each branch independently.

`insertFilter()`: To add one filter/effect to a certain branch the method `insertFilter()` can be invoked. The `branchId` is used to specify the branch into the filter/effect node defined by `filter_nd` is supposed to be inserted. Similar to the method `addBranch()`, an ID that identifies the inserted filter is returned if the call succeeds. This ID can be used to remove the filter again, or to specify a position in the already present filter chain (see parameter `before`).

```
1 int insertFilter(int branchId,  
2                 NodeDescription& filter_nd,  
3                 int before = -1);
```

`insertFilters()`: With the help of this method a list of filters can be inserted into a branch with one single method call.

```
1 list<int> insertFilters(int branchId,  
2                        list<NodeDescription> filters,  
3                        int before = -1);
```

`removeFilter()`: This method can be used to remove a certain filter from a branch.

```
1 void removeFilter(int branchId, int id);
```

Figure 4.6 on the next page shows the runtime behavior of the `MediaObject` and its `GraphHandler` for the example for video playback with `Phonon` given in Subsection 2.2.3.

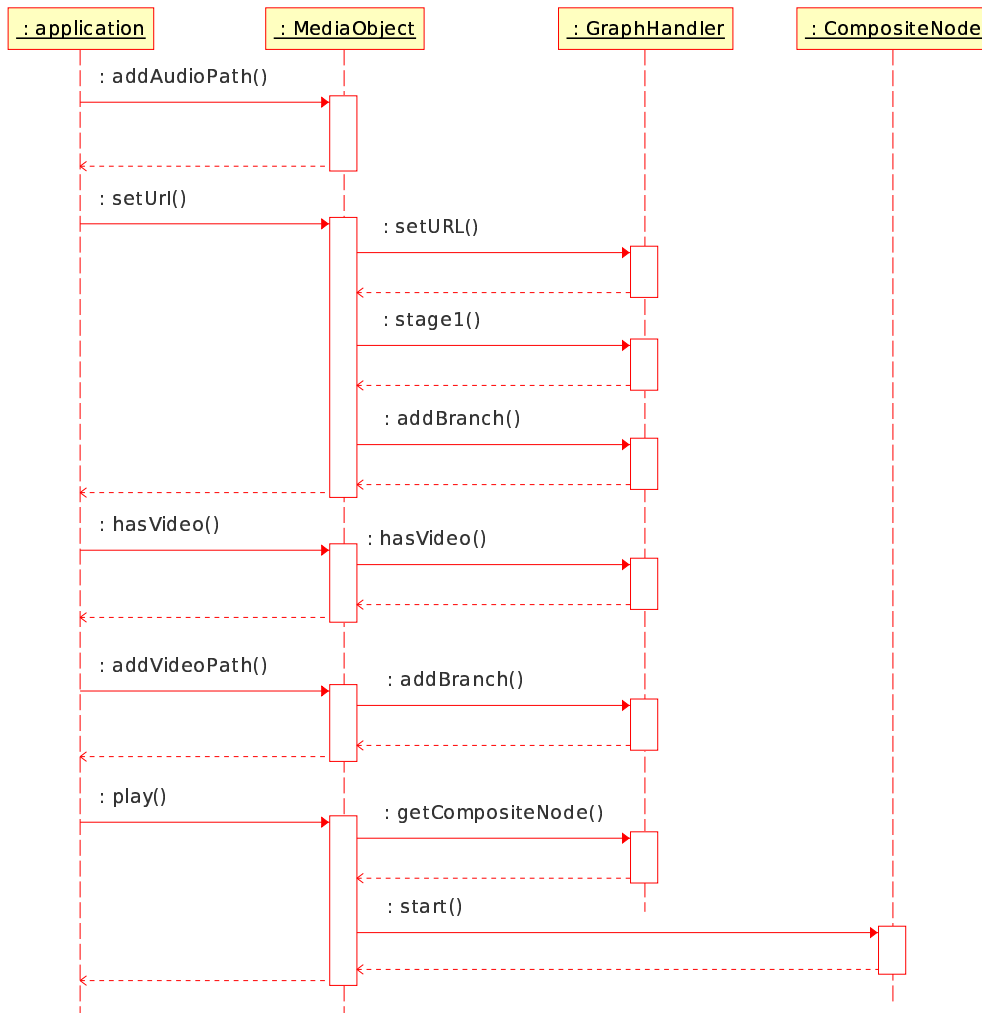


Figure 4.6: Usage of the `GraphHandler` at runtime (see the example for video playback in Subsection 2.2.3).

4.1.5 Automatic Flow Graph Build-up

Since the `GraphHandler` has to build up a flow graph automatically (based on a given URL), a proper algorithm to achieve this is necessary. The `GraphBuilder` already implements such an algorithm (see [LS05]). Unfortunately, this algorithm can not be used without further adaption, because the flow graph can not be built up at once (see the paragraph on the stages) like the `GraphBuilder` does. Thus, a more generic algorithm (based on the algorithm used in the `GraphBuilder`) that finds a proper successor node to an already requested and connected node was implemented.

In general, this algorithm distinguishes between looking for a demultiplexer node, or a converter node. When the `GraphHandler` is brought to stage 1, finding a demultiplexer node is required, while during stage 2 a converter node is necessary. In both cases, this algorithm searches for nodes, the right node, that can be possibly connected to an existing node, the left node (see Figure 4.7). Therefore, the output format of the left node has to match the input format of the right node. If there are more than one possible right nodes, the algorithm has to choose one. In case of searching for a converter node, the output format of those nodes will be compared with the input format of the sink node they will be eventually connected to. The *best* match will be used as right node. In case of searching for a demultiplexer node, the first of the possible nodes is chosen.

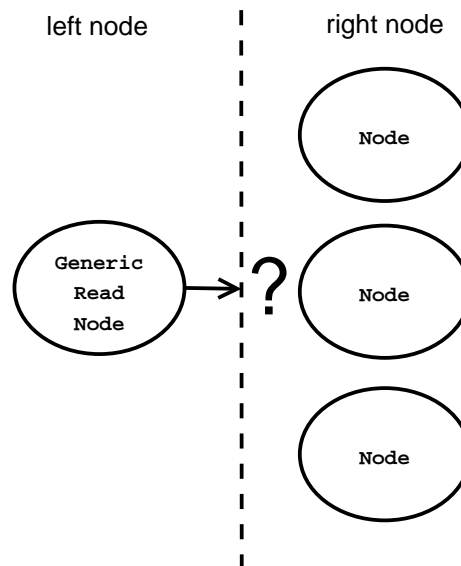


Figure 4.7: The algorithm has to find a matching right node to connect to the left node. In this example the left node is a `GenericReadNode`.

4.1.6 Field of Application

During the development of the `GraphHandler` it turned out that this class offers a lot of useful features that are not bound to the context of the NMM backend for Phonon. Thus, the API of the `GraphHandler` was adapted to meet the possible requirements of NMM applications as well. Furthermore, the API and the run-time behavior was adapted to the `GraphBuilder`. Since the functionality of the `GraphBuilder` is a subset of the functionality of the `GraphHandler`, the `GraphBuilder` can be easily replaced by the `GraphHandler`. Only minor changes in the code are necessary.

4.2 Advanced Configuration

In order to accomplish the goals as introduced in Subsection 3.2.1, developing a separate configuration application was selected. With the help of this application the user of Phonon-based applications will be able to define additional (remote) hosts for playback of audio and/or video. Furthermore, the user will be able to select a certain audio stream (if there are more than one) for each host independently. In addition, the volume for the audio playback can be adjusted; again, for each host independently. Besides this, selecting the desired sink nodes for playback is possible as well.

The only requirements those remote hosts have to comply with are a running serverregistry of NMM, and they have to provide the means to render audio and/or video, e.g. a sound card. The configuration application was first introduced in [FKRL06].

Since usually the designated hosts for playback differ from Phonon-based application to Phonon-based application, it has to be possible to specify and adjust different sets of host lists. One for every application.

This configuration application allows the user of Phonon-based applications to create complex scenarios with little effort. Hence, there is no knowledge about NMM required, which makes the application easy to use. Furthermore, there are no additional settings, or additional code needed to make the configuration application work. Only the NMM backend for Phonon needs to be used in order to benefit from this configuration application. Moreover, this configuration tool can be used in combination with any Phonon-based KDE application.

4.2.1 GUI Design

The objective of the GUI is to provide the KDE user access to the advanced features of NMM. The problem that arises when offering such versatile fea-

tures and therefore numerous possible settings by means of a GUI is that designing such a GUI to be user friendly is a hard task. In order to overcome the problem of the complexity the GUI was designed from a users perspective, since GUIs designed in this manner are more likely to be characterized as providing a proper usability. Thus, not all features of NMM can be used to their full extent with the help of the configuration application. However, the benefit of decreasing complexity is increasing usability of the GUI.

As already mentioned, the GUI was designed from a users perspective. Therefore, the configuration application allows the user to adjust settings arranged by (remote) hosts. Obviously, such hosts can be added and removed. Furthermore, the user can make the following adjustments for each host independently:

- Enable or disable audio rendering.
- Enable or disable video rendering.
- Select a Node that will be used for audio playback.
- Select a Node that will be used for video display.
- Select a certain audio stream, if there are more than one available in the current media data.
- Adjust the volume of the audio playback.
- A short status information concerning the connection to the host and therefore to the nodes running on this host.

Additionally, the status of the connection to the remote host is displayed. Figure 4.8 shows a screenshot of the current implementation of the GUI.

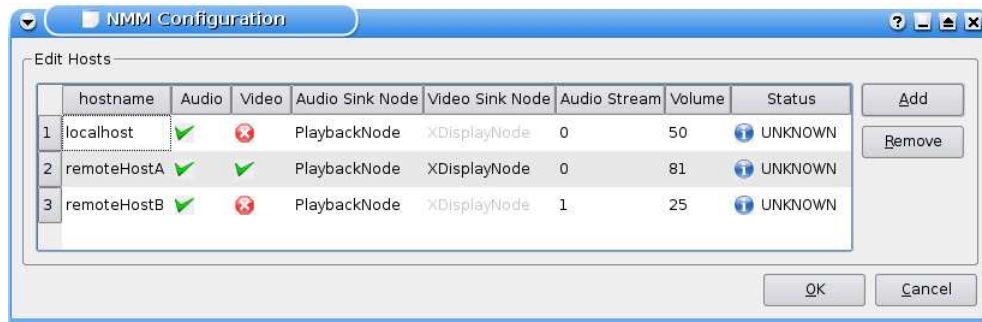


Figure 4.8: The NMM configuration application.

4.2.2 Implementation

The table that is used for displaying the (remote) hosts and their settings (see Figure 4.8 on the preceding page) was implemented using the class `QTableView` provided by Qt. This class allows the developer to separate the data from its presentation according to the *Model/View/Controller (MVC)* pattern [GHJV94].

Figure 4.9 gives an overview about the classes used for implementing the configuration application.

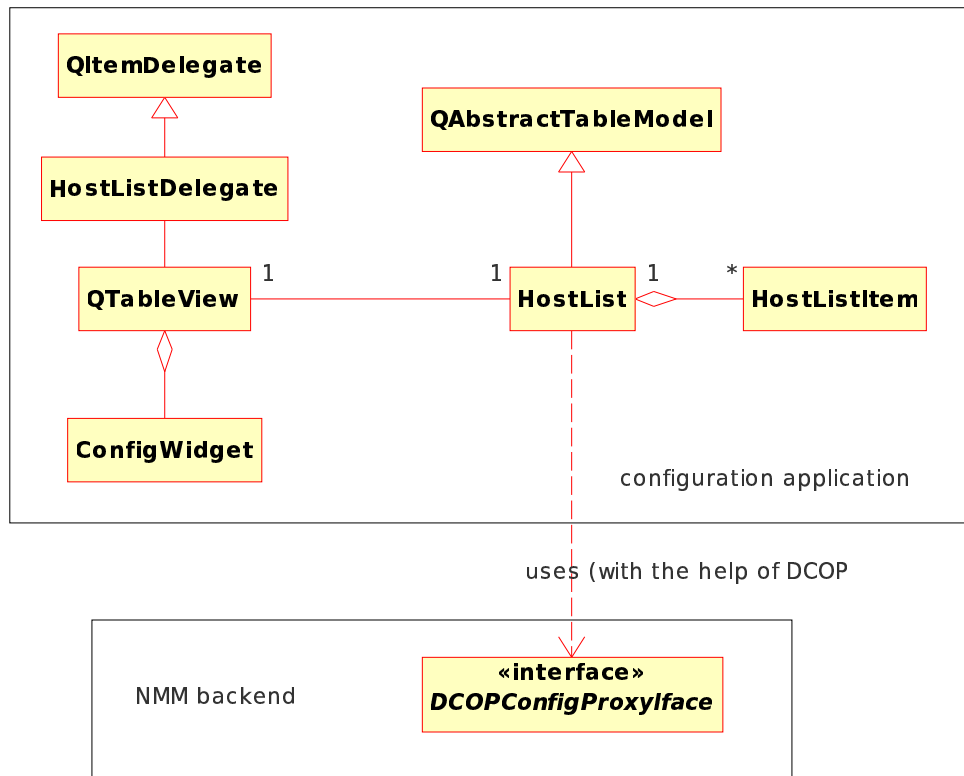


Figure 4.9: The classes used for implementing the configuration application.

ConfigWidget: This Widget is used to aggregate all GUI elements, like buttons and the `QTableView` mentioned above. The configuration application uses this class as so-called *main widget*. For details on GUI programming using Qt see [BS04], [Troa], or [Trob].

HostListDelegate: The `QTableView` allows the developer to assign a delegate. A delegate is used to customize the behavior for editing the data displayed in the table. Therefore, the use of a delegate makes it

possible to provide individual widgets for editing of a certain item of the table. Thus, a `QComboBox` is used to select the audio and video sinks, for example. In the example given, the user can easily select one of the offered choices and does not have to enter the whole name of the desired sink. Using proper editors for all settings increases the usability by eliminating as much wrong values of settings as possible.

HostList: This class is derived from the abstract class `QAbstractTableModel`. Therefore, it implements the data model for the `QTableView`. Internally, a list of `HostListItem`s is used to store all data necessary. Additionally to providing the methods defined by the abstract base class it uses the DCOP interface `DCOPConfigProxyIface` (see Subsection 4.2.3) to retrieve the data of all branches currently used when an object of the class `HostList` is instantiated. When the adjusted settings are finally saved, the DCOP interface is used again in order to notify the backend of the changes.

HostListItem: This class is used to store all data concerning one host. In detail this contains the hostname, whether audio and/or video should be rendered, the nodes that should be used for the audio and the video sink, the selected audio stream, the volume for audio playback and finally some information on the status of the connection to the host. Furthermore, the concept that Qt used for implementing the class `QTableView` and the abstract base class for the data model (`QAbstractTableModel`) makes it possible to mark certain items of such a table *editable* or *not editable*. This can be used to disable some items, when changing the underlying data would not make sense. When the media data does not contain a video stream, for instance, selecting a sink node for video display is unnecessary; therefore, the video sink property can be safely disabled.

DCOPConfigProxyIface: This is the DCOP interface the NMM backend provides to offer a subset of the functionality of the `GraphHandler` to other DCOP using applications. More details about this interface and how it is integrated into the backend can be found in Subsection 4.2.3.

At the current stage of the implementation all information concerning the selected properties are only stored in the object of the class `GraphHandler`. Thus, all settings will be lost when this object is destroyed. This means that the user has to adjust these settings every time another instance of the class `GraphHandler` is used. Since this is not user-friendly at all, it is planned

to add additional methods to the `DCOPConfigProxyIface` that will make it possible to save the settings persistently.

4.2.3 Integration in Backend

As stated in Subsection 3.2.2, a DCOP interface (`DCOPConfigProxyIface`) is used to make some advanced features of NMM, that are not covered with the Phonon API, available. A brief introduction to implementing such an interface is given in [BE99]. However, the issue of how to integrate this interface into the backend needs to be resolved.

The class `GraphHandler` has been introduced (see Section 4.1) as the tool for creating and manipulating the corresponding NMM flow graph of a `MediaObject` and its paths and outputs. Since, the `DCOPConfigProxyIface` provides the functionality to alter the flow graph, this interface and the `GraphHandler` are closely related. Figure 4.10 on the following page shows the integration of the `DCOPConfigProxyIface` into the backend. The class `ConfigProxy` implements the DCOP interface; therefore, it has to translate the call from the DCOP methods to calls of the methods from the `GraphHandler`.

The reason why the interface of the methods defined by `DCOPConfigProxyIface` (see Figure 4.11 on the next page) differs from the interface of the methods defined by the `GraphHandler` lies in the concept of DCOP [BE99]. During a DCOP method call every parameter is serialized; thus, custom types or classes can not be used unless they implement the serialization means defined by DCOP. Since some classes of Qt already provide these means, those classes are used in the DCOP interface. Another restriction of DCOP is that using in-and-out parameters is not possible. Therefore, only the return value can be used as a reply.

To sum up, the `DCOPConfigProxyIface` interface is used to make a selection of methods provided by the class `GraphHandler` available. The class `ConfigProxy` implements the `DCOPConfigProxyIface` and acts as translator between the DCOP interface calls and the `GraphHandler`. Notice that the `DCOPConfigProxyIface` can be used by any application that uses DCOP, for instance the configuration application introduced in Section 4.2.

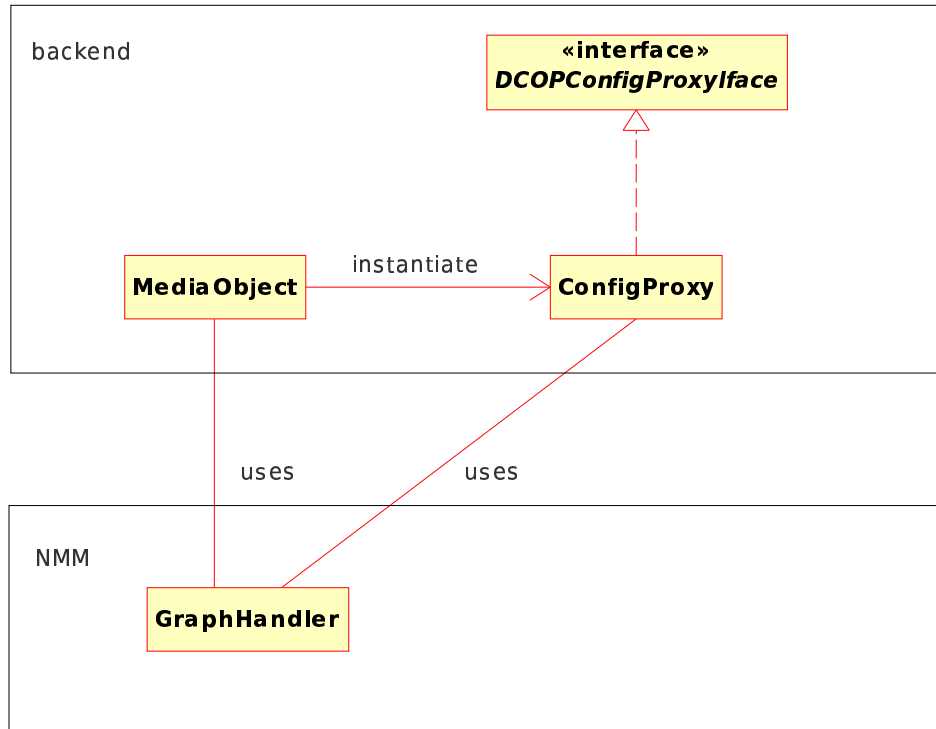


Figure 4.10: Integration of the DCOPConfigProxy interface into the backend.

4.3 Summary

This chapter addressed how the backend and the additional facilities to make the advanced features of NMM available are implemented. In detail, the class `GraphHandler` was introduced as the means to build up and manipulate the NMM flow graph as Phonon requires. Furthermore, the implementation and integration of the DCOP interface `DCOPConfigProxy`, as stated in Subsection 3.2.2, was shown.

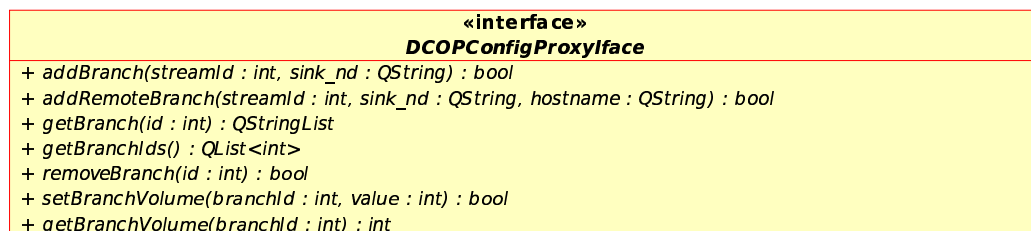


Figure 4.11: The methods defined by the interface `DCOPConfigProxy`.

Chapter 5

Examples

This chapter gives an idea of possible scenarios a user of Phonon-based applications combined with the NMM backend for Phonon can create.

5.1 Distributed Audio Playback

This scenario, as shown in Figure 5.1 on the following page, uses multiple additional hosts for audio playback. The source of the media data is located on *host A*; furthermore, the Phonon-based application used for playback and the NMM configuration application (see Section 4.2) runs on *host A* as well.

In order to build up a scenario like shown in Figure 5.1 on the next page first a data source needs to be defined. This is usually done with the Phonon-based application by specifying an URL. As stated in the paragraph on the core classes of Phonon in Subsection 2.2.2, the `MediaObject` uses URLs to specify the data source. Remember that the `GraphHandler`, as introduced in Section 4.1, that is internally used by the `MediaObject` to build up a NMM flow graph, uses URLs as well. The NMM configuration application (see Section 4.2) can then be used to define the additional hosts. Figure 5.2 on the following page shows the corresponding NMM flow graph to this scenario.

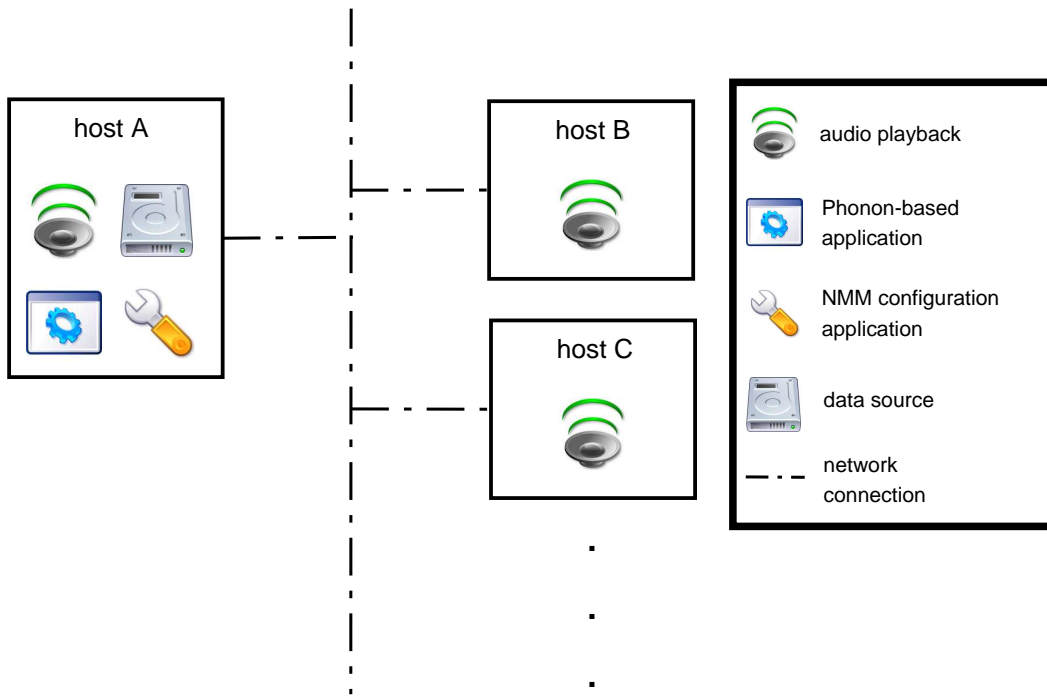


Figure 5.1: A scenario for distributed audio playback.

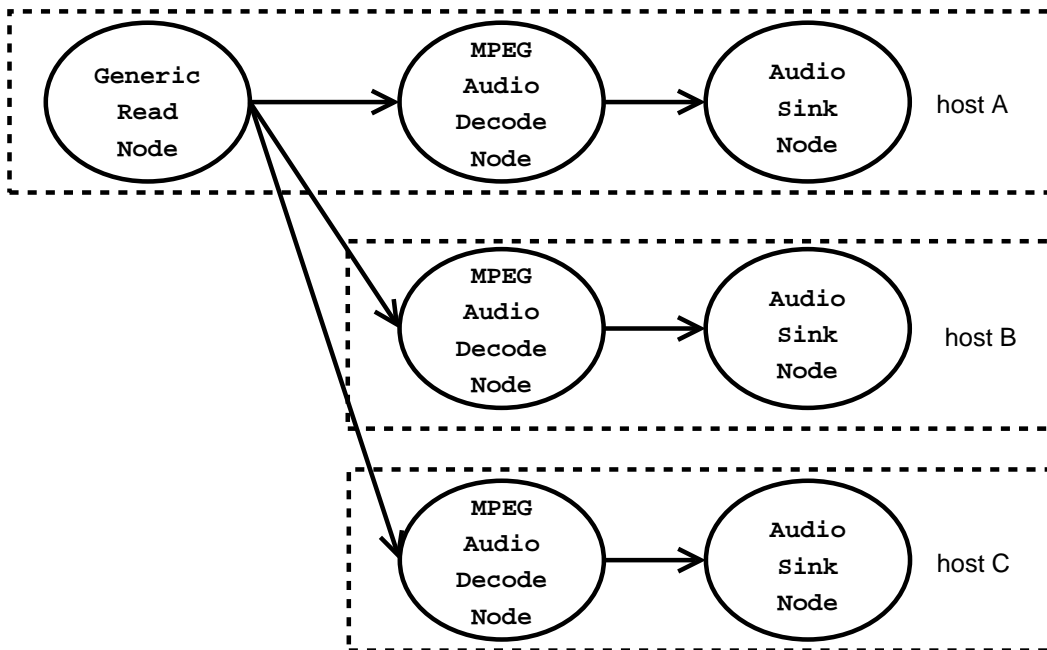


Figure 5.2: The NMM flow graph used for distributed audio playback.

5.2 Distributed Audio and Video Playback

In this scenario the media data is comprised of one video stream and several audio streams. As well as in scenario presented before (see Section 5.1), the data source, the Phonon-based application and the NMM configuration application are located on *host A*. Since there are multiple audio streams present, every participating host can individually choose one of them. Or decide not to play back any audio at all, like *host C*.

Consider having a DVD with different language audio tracks. Such a scenario can be used to watch this DVD with the English audio track on one PC (e. g. *host A*) and with the German audio track on another PC (e. g. *host D*). Furthermore, a computer connected to a HiFi system (like *host B*) can be used for high quality audio playback. Additionally, if a PC is connected to a beamer or some other big screen (e. g. *host C*), this host can be used to benefit from a bigger screen than the usual PC monitor.

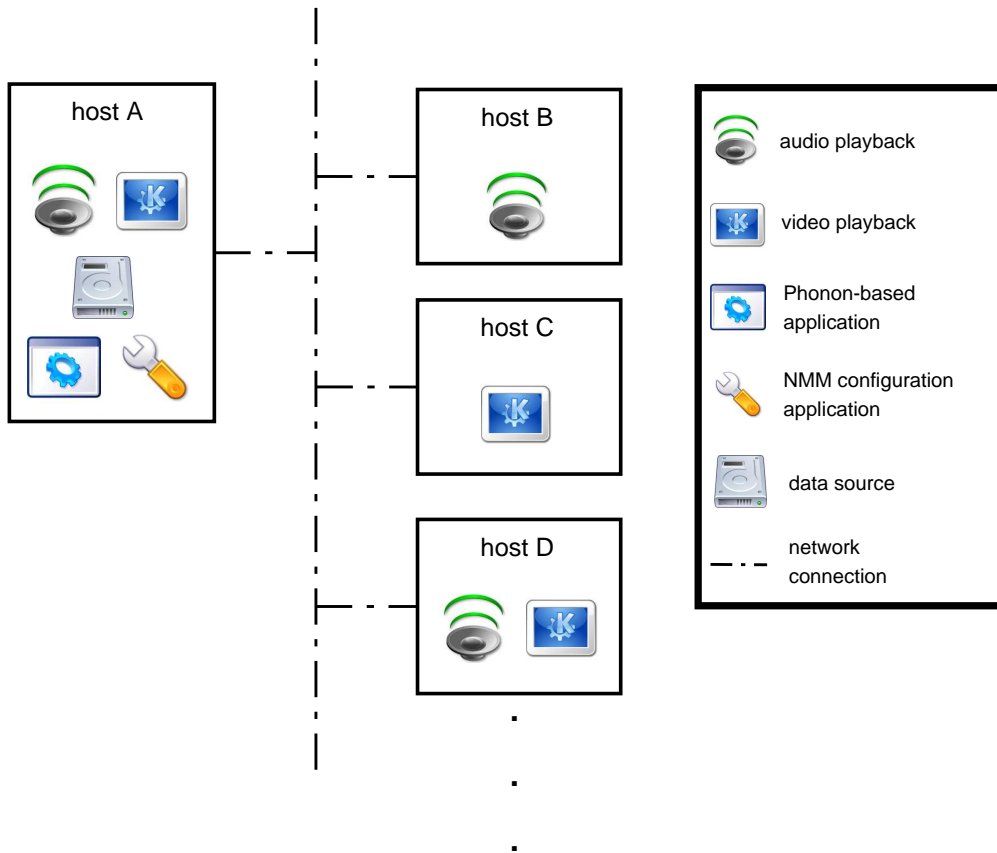


Figure 5.3: A scenario for distributed audio and video playback.

The setup for this scenario is similar to the setup for the scenario introduced in Section 5.1. Since a video stream is present and there are multiple audio streams available, the user can select a certain audio stream and whether the video stream is supposed to be displayed by means of the NMM configuration application. The NMM flow graph required for this scenario is shown in Figure 5.4.

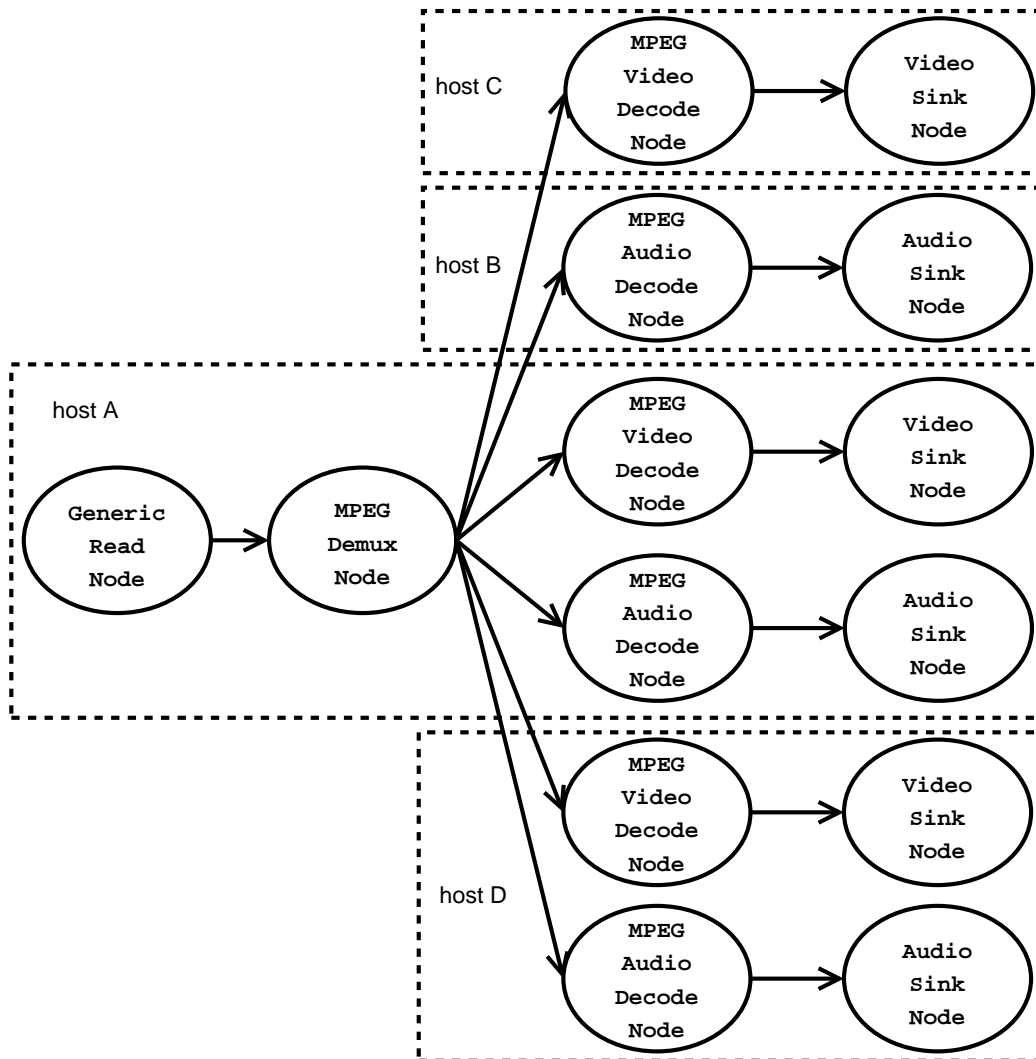


Figure 5.4: The NMM flow graph used for distributed audio and video playback.

5.3 Using a Remote Data Source

Contrary to the scenarios already presented, the media data source (*host B*) is **not** located on the same host as the Phonon-based application and the NMM configuration application (*host A*). Using a remote data source can be easily achieved by specifying a proper URL (see [wgd]), like `file://hostB/data/myfile.mpeg`. Most important is that there is no need to set up any filesharing mechanism, since NMM itself can distribute the data as required.

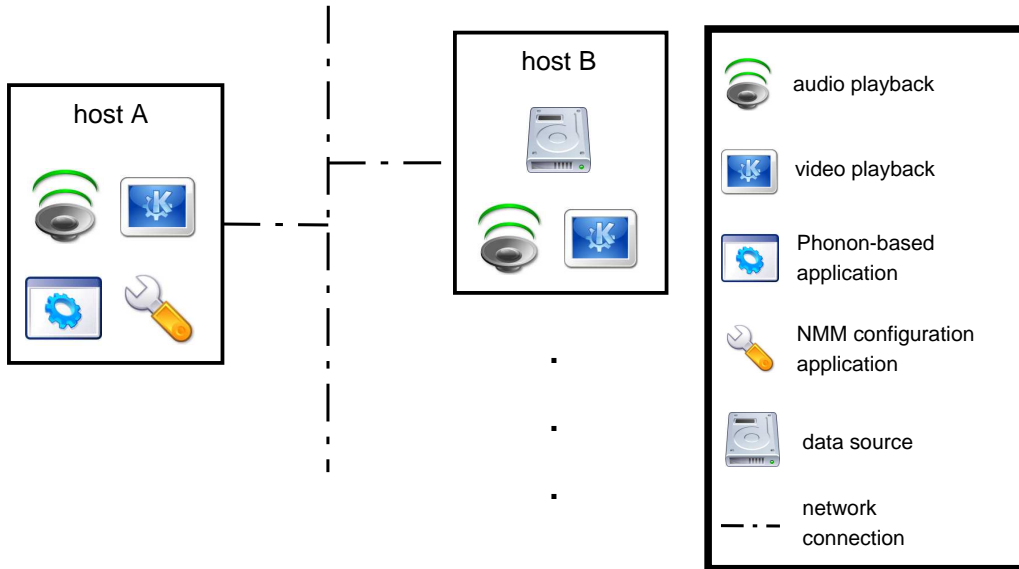


Figure 5.5: In this scenario a remote data source is used.

The fact that the source of the data is not located on the same host as the Phonon-based application does not influence the setup of this scenario; therefore, there is no difference to the setup as stated for the scenario for audio and video playback (see Section 5.2). The `GraphHandler` will build up a flow graph as shown in Figure 5.6 on the following page.

5.4 Using a Remote Live Data Source

Again *host B* acts as source for the multimedia data. Additionally, this scenario (see Figure 5.7 on page 59) shows that even a live data source, like a TV board or a camera, can be used. Notice that it is not mandatory that the host providing the data is also used for any playback. It is required that a serverregistry (see Subsection 2.1.5) is running on that host, though.

The NMM flow graph used is shown in Figure 5.8 on page 59.

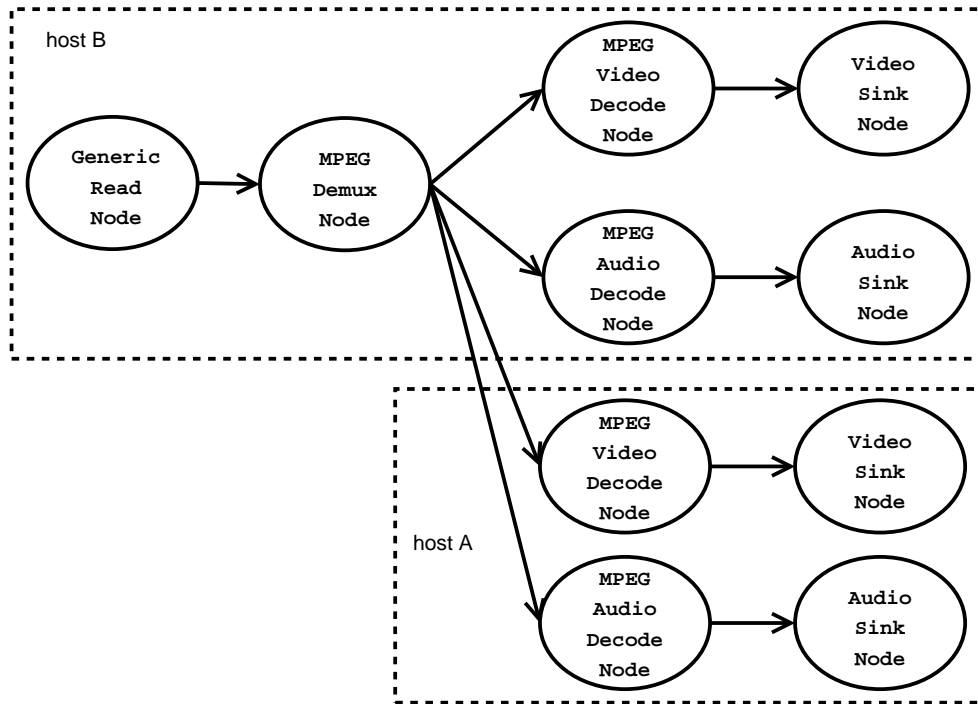


Figure 5.6: The NMM flow graph used to create this scenario.

5.5 Summary

Notice that the scenarios presented in this chapter are only examples. The usage of the NMM backend for Phonon is in no way restricted to these scenarios. Furthermore, the number of participating hosts is only limited by the network capacity.

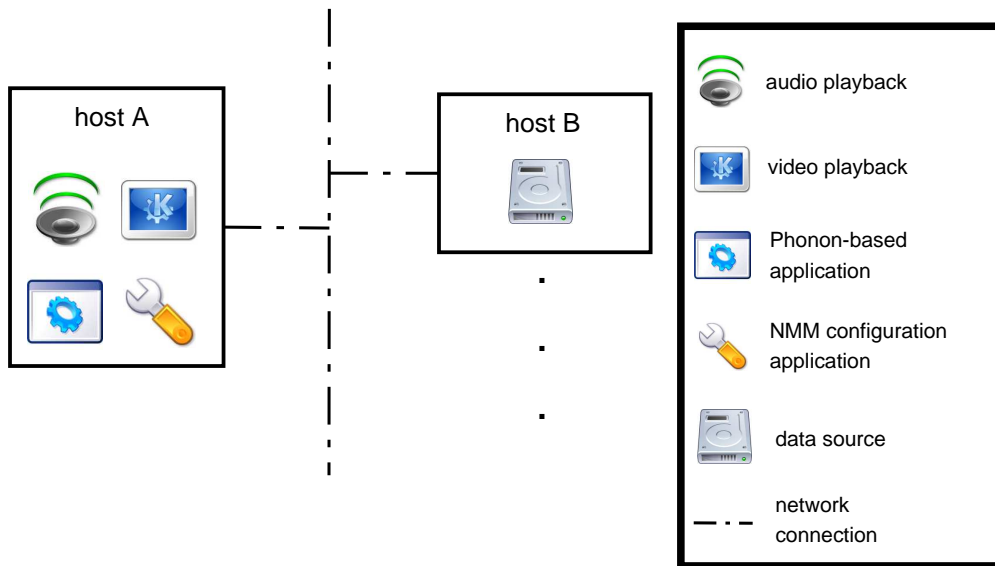


Figure 5.7: This scenario uses a remote live data source.

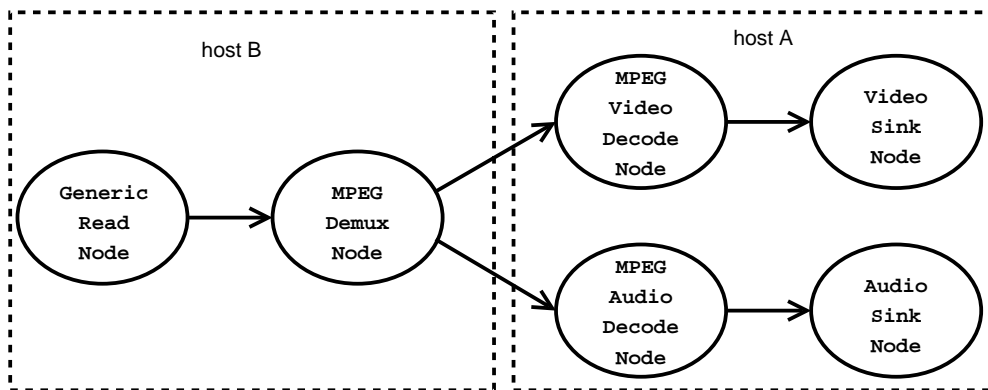


Figure 5.8: The NMM flow graph used to create this scenario.

Chapter 6

Conclusion

This chapter will provide a short overview of the achievements (see Section 6.1) and the still unresolved issues of this work (see Section 6.2).

6.1 Achievements

The two primary goals of this work are the implementation of a NMM backend for Phonon and to integrate certain advanced features of NMM into the backend in order to make them available to users of Phonon-based applications.

Thus, the first task was to implement a backend based on NMM. The backend has to build up a flow graph automatically and furthermore provide the means to alter this flow graph at any time. Even though NMM offers a very powerful class to build up a flow graph automatically, the class `GraphBuilder`, it can not comply with the requirements of Phonon. Therefore, a new class, the class `GraphHandler`, needed to be developed.

The `GraphHandler` (see Section 4.1) makes it possible to automatically build up a flow graph with numerous sinks nodes. Moreover, this flow graph can be manipulated even while the flow graph is running.

The big challenge while implementing the `GraphHandler` was to provide sufficient flexibility during the graph build up and for manipulating an existing flow graph. As a conclusion to this, a lot of details needed to be considered and properly taken care of. Since with the help of the `GraphHandler` an arbitrary number of different flow graphs can be created, a lot of testing was necessary as well. Furthermore, the `GraphHandler` is not supposed to be error-prone. This means that if adding a branch, for example, causes an exception, this should not affect the rest of the flow graph.

Since the objective of this work was not only to provide an NMM back-

end for Phonon, but also to integrate certain NMM specific features into the backend, a concept to make those feature available to Phonon users and developers needed to be elaborated. A very important goal of this concept was **not** to by-pass the backend independence of Phonon-based applications. Therefore, the DCOP interface `DCOPConfigProxyIface` (see Subsection 4.2.3) was implemented. With the help of this interface the NMM flow graph can be manipulated by means of the `GraphHandler`. Additionally, a proof-of-concept prototype, the configuration application as introduced in Section 4.2, that uses this interface was implemented.

The class `GraphHandler` is available as part of the NMM distribution. Since Phonon as well as the NMM backend for Phonon is still under development, the source code of this backend and the configuration application can be found in the multimedia repository of KDE [KDEa].

6.2 Future Work

Backend

Even though the current implementation of the NMM backend for Phonon provides all means for playback, there are still features unimplemented. Furthermore, as mentioned in Subsection 2.2.4, Phonon itself is under development and therefore subject to changes. In detail, the open tasks for the backend are the following.

- Implementation of the a/v capturing API of Phonon.
- Implementation of the widgets for user interaction (like the video widget, as introduced in the paragraph on the core classes of Phonon in Subsection 2.2.2).
- Implementation of software volume control.
- Implementation of the filter/effect API of Phonon. Notice that the `GraphHandler` already supports inserting and removing filters/effects (see Section 4.1), but those features still have to be integrated into the backend.
- Adapt to changes of Phonon, KDE and Qt.
- Implementation of future extensions of Phonon.

Integration of Advanced Capabilities of NMM

As stated in Subsection 3.2.1, the integration concept, as implemented at this time, is limited to providing the means to use multiple (distributed) sink nodes for playback. Since NMM offers a lot of other useful features, e. g. session sharing or seamless handover, it is likely that some of them will be integrated in the future. This can be easily accomplished by extending the `DCOPConfigProxyIface` with proper new methods.

Configuration Application

The configuration application implemented as part of this work can be seen as proof-of-concept prototype. Therefore, there are some things that need to be added or improved.

- The current implementation only supports one Phonon-based application and one `MediaObject`. Thus, the configuration application should be extended in order to make it possible to configure numerous Phonon-based applications with multiple `MediaObjects`.
- Even though the GUI supports displaying status information concerning the connection to the (remote) host (and therefore to the server-registry), this functionality is not implemented yet.
- The configuration application should allow the user to save sets of hosts and load such sets, since it is undesirable to set up the host list for each `MediaObject` from scratch every time.
- New NMM-specific features that are added to the `DCOPConfigProxyIface` need to be integrated as well.

Bibliography

- [BCP03] A. Buchmann, G. Coulson, and N. Parlavantzas, *Introduction to middleware*, IEEE Distributed Systems Online, <http://dsonline.computer.org/middleware/>, 2003.
- [BE99] P. Brown and M. Ettrich, *DCOP: Desktop COmmunications Protocol*, <http://developer.kde.org/documentation/other/dcop.html>, 1999.
- [BS04] J. Blanchette and M. Summerfield, *C++ GUI Programming with Qt 3*, Prentice Hall in association with Trolltech Press, 2004.
- [FKRL06] B. Fuchshumer, M. Kretz, M. Repplinger, and M. Lohse, *Phonon and NMM – Multimedia Architectures for the Desktop and Beyond*, LinuxTag 2006 (Wiesbaden, Germany), LinuxTag e.V., 2006.
- [GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison Wesley, 1994.
- [gst] *GStreamer Website*, <http://gststreamer.freedesktop.org/>.
- [hel] *Helix Website*, <https://helixcommunity.org/>.
- [KDEa] KDE community, *KDE Multimedia Repository*, <svn://anon-svn.kde.org/home/kde/trunk/KDE/kdemultimedia>.
- [KDEb] ———, *KDE Website*, <http://www.kde.org>.
- [Kre] M. Kretz, *Phonon Website*, <http://phonon.kde.org/>.
- [Loh05a] M. Lohse, *The Network-Integrated Multimedia Middleware (NMM) : Basic Introduction*, <http://graphics.cs.uni-sb.de/-NMM/current/Docs/intro/>, 2005.

- [Loh05b] ———, *Network-Integrated Multimedia Middleware, Services and Applications*, Ph.D. thesis, Lehrstuhl für Computergraphik, Universität des Saarlandes, Saarbrücken, Germany, 2005.
- [LRS02] M. Lohse, M. Repplinger, and P. Slusallek, *An Open Middleware Architecture for Network-Integrated Multimedia*, IDMS/PROMS'2002 Joint International Workshops on Interactive Distributed Multimedia Systems / Protocols for Multimedia Systems (Coimbra, Portugal), November 2002.
- [LRS03a] ———, *Dynamic Distributed Multimedia: Seamless Sharing and Reconfiguration of Multimedia Flow Graphs*, 2nd International Conference on Mobile and Ubiquitous Multimedia (MUM 2003) (Norrköping, Sweden), December 2003.
- [LRS03b] ———, *Session Sharing as Middleware Service for Distributed Multimedia Applications*, First International Workshop on Multimedia Interactive Protocols and Systems, MIPS 2003 (Naples, Italy), November 2003.
- [LS01] M. Lohse and P. Slusallek, *Extended Format Definition and Quality-driven Format Negotiation in Multimedia Systems*, EUROGRAPHICS Workshop (Manchester, United Kingdom), September 2001.
- [LS05] ———, *Towards Automatic Setup of Distributed Multimedia Applications*, The 9th IASTED International Conference on Internet and Multimedia Systems and Applications (IMSA) 2005 (Honolulu, Hawaii, USA), August 2005.
- [Mil92] D. L. Mills, *Network Time Protocol (Version 3) Specification, Implementation and Analysis (RFC 1305)*, 1992.
- [mpe] *Information technology—Generic coding of moving pictures and associated audio information: Systems(iso/iec 13818-1:2000)*, ISO/IEC.
- [ntp] *The Network Time Protocol*, <http://www.ntp.org>.
- [pal98] *Recommendation ITU-R BT.470-6, Conventional Television Systems*, International Telecommunications Union, 1998.
- [Rep03] M. Repplinger, *Eine Architektur zur Anbindung und Kontrolle von im Netz verteilten Multimedia-Geräten*, Diploma thesis,

- Lehrstuhl für Computergraphik, Universität des Saarlandes, Saarbrücken, Germany, 2003.
- [RP02] P. Rechenberg and G. Pomberger, *Informatik-Handbuch*, 3rd ed., Hanser, 2002.
- [RWLS05] M. Repplinger, F. Winter, M. Lohse, and P. Slusallek, *Parallel Bindings in Distributed Multimedia Systems*, The 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2005) (Columbus, Ohio, USA), June 2005.
- [SN95] R. Steinmetz and K. Nahrstedt, *Multimedia: Computing, Communications and Applications*, Prentice Hall, 1995.
- [Str97] B. Stroustrup, *The C++ Programming Language*, 3rd ed., Addison Wesley, 1997.
- [Troa] Trolltech, *Qt 4.1 Whitepaper*, <http://www.trolltech.com/developer/documentation/whitepapers/qtwhitepapers/>.
- [Trob] ———, *Qt Online Reference Documentation*, <http://doc.trolltech.com/>.
- [Wes] S. Westerfeld, *aRts Website*, <http://www.arts-project.org/>.
- [wga] NMM work group, *Hello World! Welcome to NMM Application Development*, <http://graphics.cs.uni-sb.de/NMM/current/Docs/helloworld/>.
- [wgb] ———, *List of available plug-ins for NMM*, <http://graphics.cs.uni-sb.de/NMM/current/Docs/pluginlist/>.
- [wgc] ———, *Network-Integrated Multimedia Middleware (NMM)*, <http://www.networkmultimedia.org>.
- [wgd] ———, *Using the Graph Builder*, <http://graphics.cs.uni-sb.de/NMM/current/Docs/clic/x142.html>.
- [xin] *Xine Website*, <http://xinehq.de/>.