Synchronization in the Network-Integrated Multimedia Middleware (NMM)

Stephan Didas
didas@studcs.uni-sb.de

Advanced practical project
according to a topic from
Prof. Dr.-Ing. Philipp Slusallek
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung 6.2 – Informatik
Universität des Saarlandes, Saarbrücken, 2002

Disclaimer

The software described in this document is under continuous development. Some concepts and software components therefore may be different in the software distributions available on the internet. For further information about the current state of the project and the software distribution see [1].

Contents

1 Introduction **3**
1.1 Media classification . . . 3
1.2 Introduction to synchronization . . . 4
1.3 Synchronization requirements . . . 5
1.3.1 Hard and soft real-time constraints . . . 6
1.3.2 The special case of lip synchronization . . . 6

2 Introduction to NMM **7**
2.1 Nodes . . . 8
2.2 Messages . . . 9

3 Synchronization in NMM **11**
3.1 Overview . . . 11
3.2 Elementary data types . . . 13
3.2.1 Rational numbers . . . 13
3.2.2 Time representation - `Time`, `Interval` and `UserTime` . . . 13
3.2.3 Time information in the multimedia stream - `Timestamp` 14
3.3 Objects in the graph . . . 15
3.3.1 `Clock` and `TimedElement` . . . 15
3.3.2 Timestamp creation - `StreamTimer` . . . 16
3.3.3 `GenericSyncSinkNode` and its subclasses . . . 16
3.3.4 `Controller` and its subclasses . . . 18
3.3.5 `Synchronizer` . . . 21
3.4 Events related to synchronization . . . 22
3.4.1 `sync_enable` and `sync_disable` . . . 22
3.4.2 `sync_reset` . . . 23

4 Plug-ins and Applications **25**
4.1 Source nodes . . . 25
4.1.1 Live sources . . . 25
4.2 Filter nodes . . . 26
4.3 Converter nodes . . . 26
4.4 Sink nodes . . . 27
4.4.1 Video sinks . . . 27
4.4.2 Audio sinks . . . 27
4.5 Audio visualization . . . 29

4.5.1 The mp3vis and mp3vis2 applications 29
4.5.2 `ScopeNode` and `SAnalyzerNode` 30
4.5.3 Synchronization in the application code 31
4.6 MPEG decoding 31

A Source Files **35**

Chapter 1

Introduction

Multimedia is becoming a more and more important field of application for computers. The traditional media like radio and television have been supplemented and even partionally replaced by streaming media and multimedia internet sites. Even the traditional receiver devices change: Not only the personal computer, but also the personal digital assistant and the cellular phone can take part in multimedia applications. Typically these components are able to communicate via a network connection.
The aim of the Network-Integrated Multimedia Middleware (NMM) for Linux project is to create a middleware that allows to access different kinds of media on a variety of devices. The network is considered as an integral part of the multimedia system, and the application programmer should be able to use and control distributed devices in a network-transparent way. See [1] for further information about the project, its goals and its current state. The subject of this practical project is to extend this already existing multimedia software system by basic facilities for the synchronization of multimedia streams. These synchronization abilities are especially important for dealing with multiple media streams.

1.1 Media classification

One can distinguish between *continuous* and *discrete media*. In this case continuous means the same as time-dependent and discrete means the same as time-independent. A book is an example for a discrete medium, a video sequence for a continuous medium. A system for an electronic lecture-room is an example of increasing popularity that mixes both media types: The audio and video data that shows the lecturer as continuous media and the texts he presents as discrete media.

In the literature the notion *continuous media* is also used for media which consists of presentation units with equal durations (see [13], page 567).

Although a medium is continuous it can not be stored and worked up like this: All what can be done is to divide it into discrete parts so that a computer is capable to work it up. This division has to be fine enough that a human observer can not notice it. For example, a continuous motion is split up in 25

or 30 video frames per second. The audio signal of a 1kHz sine wave is sampled 44100 times per second, and only the value at the sample times is stored. That means even continuous media are stored as discrete values. A multimedia data stream is a sequence of such values which do not in any case provide useful information for themselves. For example, one audio sample without the context of other samples does not tell us anything about the song it is taken from. To characterize a multimedia stream, it is necessary to divide it into sensible parts which are called *logical data units*, or short *LDU's*.

There is no unique way to divide a stream in LDU's. Regard a stream of raw audio data (Stereo, 16 bits per sample with a sample rate of 48000 kHz): One could say that each audio sample in the stream is an LDU. We have seen that this does not make sense in most of the cases. Another possibility is to group a reasonable number of samples, let us say 1024, together to one LDU. With the above parameters, 1024 samples are 4096 bytes. So this LDU would be a unit that can be practically transported and worked up by a computer. If the presentation consists of multiple songs, one can even regard a whole song as an LDU. In a video stream it can make sense simply to view each video frame as an LDU. However even in this case there could be other sensible possibilities: In an MPEG stream, several following video frames form a so-called *group of pictures* or *GOP*. This GOP can also be seen as the LDU of the MPEG video stream.

So there are different levels of granularity how to regard media data. It depends on the application which point of view to choose.

1.2 Introduction to synchronization

Now we want to introduce some useful notions related to multimedia and synchronization. In the context of multimedia systems *synchronization* in general refers to the content, spatial and temporal relations between different media objects. Like in most of the literature we will use the notion *synchronization* with the main focus on the *temporal relations between the media objects*.

The notion can be further specialized according to the regarded media objects:

- *Intra-object synchronization:* Intra-object synchronization refers to the temporal relations between several presentation units of the same time-dependent media object. An example for this is the presentation of a video with a constant rate of 25 frames per second which is shown in figure 1.1.

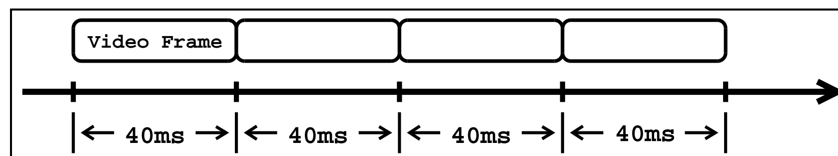


Figure 1.1: Intra-object synchronization

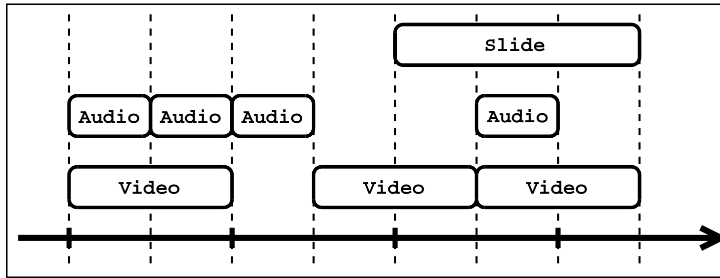


Figure 1.2: Inter-object synchronization

- *Inter-object synchronization:* Inter-object synchronization refers to the temporal relations between different media objects. The easiest example is the synchronization between the visual and acoustical information in television. Figure 1.2 shows another example with three multimedia streams.

We will see in chapter 3 that this differentiation even has an effect on the software components that are used. Intra-object synchronization is strongly connected with the notion of *jittering*. Jittering means that the temporal distance between two following objects in a stream with constant rate varies. One can state that one main aim of intra-object synchronization is the reduction of jittering.

Regarding to the origin of temporal relations one can make another distinction:

- *Live synchronization:* The aim is to reproduce the temporal correlations of different media objects as they existed during the capturing process. This is the case for a video recording program or a video conference system.

- *Synthetic synchronization:* Here the temporal relations are artificial specified. One can speak of synthetic synchronization if one adds a new audio stream to a given video stream.

1.3 Synchronization requirements

What are the requirements that we have for the synchronization of multimedia streams?

At first one can note that the requirements on principle depend on the actual types of media to synchronize. A slide in the electronic lecture-room can have a greater skew according to the audio as the speakers lips, for example.

Another obvious fact is that the requirements result from the capabilities of the observer's senses. On the one hand this makes it impossible to precisely define the temporal constraints: The reactions to a certain error in the presentation time are individually different for various observers. On the other hand that gives us the possibility to limit the dimension of the required accuracy of the presentation time to a range of some 10 milliseconds. Shorter periods of

time are normally not noticeable for a human observer. [12] and [13] give a more detailed description of synchronization requirements for different types of media.

1.3.1 Hard and soft real-time constraints

Synchronization always deals with real-time constraints. From the operating system's point of view such real-time constraints can be divided into *hard* and *soft real-time constraints*. The real-time constraints are called hard if it is necessary for the correct behaviour of the software or system that a given deadline is kept. This can even lead to a danger for human's life (one may think of industrial robots or airbag control). But what happens in a multimedia application if a deadline cannot be kept? We will see later that in several cases a delay of the presentation is not even noticeable to the observer. Therefore in the case of multimedia applications one can speak of soft real-time constraints. This makes it possible to avoid using special software like a real-time operating system (see also [3]). At the moment the NMM project uses a standard Linux operating system. That means the operating system does not support any guaranteed reaction times.

1.3.2 The special case of lip synchronization

Lip synchronization refers to a presentation of audio and video and the temporal relations between them for the particular case of human speaking.

In this case it is also difficult to gain clear results: Even a test sequence without any skew between video and audio is declared as incorrect by some test users (see [12] for more details.) Moreover there are many conditions that have an influence on the recognition of synchronization errors that complicate a general statement. Regard an audio/video sequence that shows a human speaker: The brightness of the images and the distance to the speaker have an influence on the detection of synchronization errors.

Experiments have shown that in most cases it is sufficient to keep the skew between audio and video in a region between -80 milliseconds (audio behind video) and 80 milliseconds (audio ahead of video). Regard an example application that plays MPEG-encoded audio/video streams. This application has to ensure that a single video frame is displayed at most 80 milliseconds before or after the intended presentation time. For further information please take a look at [12], pages 580 - 584, or [13], pages 588 - 592.

It is interesting that people are more tolerant against audio/video skews if the audio follows the video. As a result of the different velocities of sound and light we make the experience of a slightly delayed acoustic impression every day.

Chapter 2

Introduction to NMM

The Network-Integrated Multimedia Middleware (NMM) for Linux is being developed for the last 2 years by the computer graphics group of Saarland University. Information about the project in general and its current state can be found on the internet (see [1]). NMM is free software implemented in C++; the current distribution is available for download at the project's homepage.

One main aim of NMM is to simplify implementing multimedia applications by providing easy access to different kinds of hardware and different multimedia formats.

Central objects in NMM are `Nodes` and `Messages`. Nodes serve to encapsulate certain functionality, Messages are data or information units that are created, processed, or consumed by nodes. Figure 2.1 shows how a node can use messages to communicate with other nodes and with the application. Multimedia data is sent between two nodes with a message type called `Buffer` – indicated as `B`. To exchange information with other nodes and the application there is a second message type called `Event` – indicated as `E`. These notions will be further explained in the next two sections.

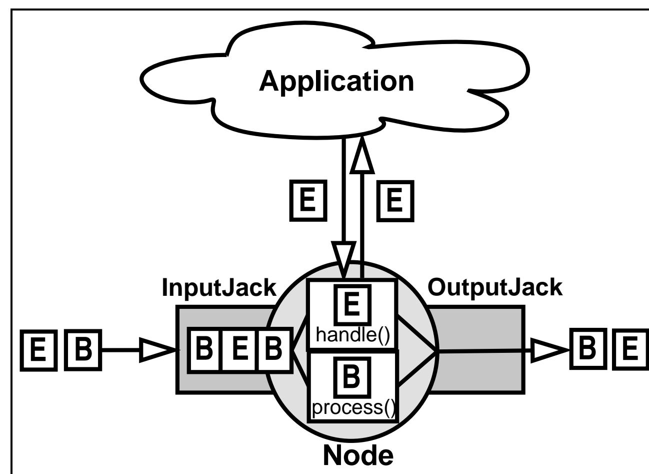


Figure 2.1: The relationship between Nodes, Messages and the application

2.1 Nodes

On the one hand a node can provide an interface to a concrete hardware device like a soundcard or a camera. On the other hand it can also be a software object like a video colorspace converter or an audio filter.

To realize connections with each other nodes have so-called `Input-` and `OutputJacks`. One could compare the nodes to things like a guitar, an amplifier or effects and the `Jacks` to the real jacks that connect them.

In principle there is no restriction on the number of input and output jacks. Not every node must have both Input- and OutputJacks. One may think of a file reader node as a data producer (*source node*) that does not need an InputJack or a video display *sink node* without an OutputJack a a pure data consumer. Multiple OutputJacks (*demultiplexer node* with one input only) or multiple InputJacks (*multiplexer node* with one output only) are allowed. The above types together with *filter node* and *converter node* make the six types of nodes that are distingiushed. There are so-called `Formats` to describe the data content of a multimedia stream. A filter node usually has the same input and output data format. An example is a node that adds an effect to an audio stream. A converter node explicitly changes the format of incoming data. One may think of a node that decodes compressed MPEG video data into single video frames.

There is a group of classes that represents the different generic node types: `GenericSourceNode`, `GenericProcessorNode`, `GenericMultiplexerNode`, `GenericDemultiplexerNode`, `GenericConverterNode`, `GenericFilterNode` and `GenericSinkNode`. They simplify the programming of new nodes, so that a programmer only needs to implement the functionality of the new node. For example, the `GenericSourceNode` has a member function with the type

```
Buffer* produceBuffer()
```

and the `GenericProcessorNode` has a function called

```
Buffer* processBuffer(Buffer*)
```

which serves to encapsulate the functionality of the nodes[1]. A `Buffer` is a container for multimedia data. This will be further explained in section 2.2.

With their jack connections nodes are able to form general *flow graph* structures. Figure 2.2 shows an example graph for a simple MPEG video player. It consists of six nodes: The only source in the graph is an `MPEGReadNode` that reads the MPEG-encoded data from the hard disc. Then the data is split up into the audio and video components by the `MPEGDemuxNode`. The `MPEGVideoDecodeNode` converts the MPEG video stream into a raw video stream, the same is done by the `MPEGAudioDecodeNode` for the audio stream. The `XDisplayNode` and the `PlaybackNode` finally present the video and audio data to the user.

One path in such a flow graph describes a path of data or information flow in the graph; the messages that flow along such a path are called a *stream*.

[1]For more information about the node types and a list of currently available nodes see http://www.networkmultimedia.org/NMM/Status/Plugins/index.html.

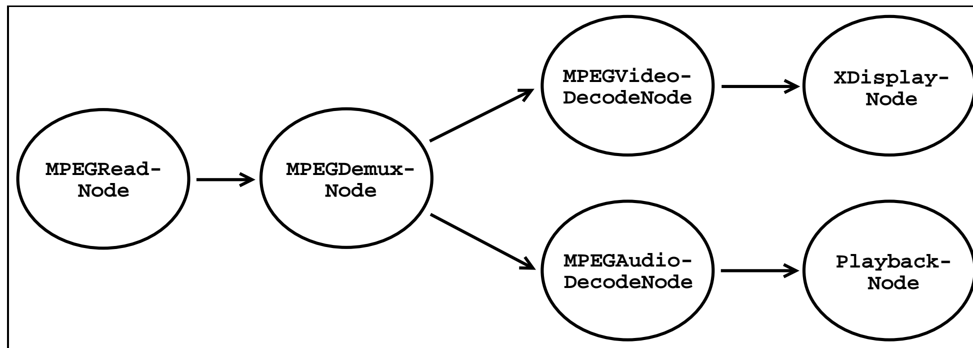


Figure 2.2: A simple node graph for an MPEG video player

Especially for nodes that provide an interface for a hardware device, there are operations that should only be executed in certain states of the whole system. For example, the audio device should not be closed during a presentation. To allow the node programmer to define such contexts for operations, nodes have different *states*. These states can be questioned and changed by the application. The node programmer can restrict certain actions (e.g. changing the input file for a reader node or changing the output device for a sound playback node) to the states that make sense. He can also implement node-specific functionality for the state transitions. This makes it possible to reserve an exclusive used hardware device or to allocate needed memory before the application starts the presentation. The possible states of a node are `CONSTRUCTED`, `INITIALIZED`, `ACTIVATED` and `STARTED`[2].

2.2 Messages

Messages can either contain multimedia data or information for the nodes. Messages which carry multimedia data are called `Buffers`, the other Message type is called `CEvent`[3] . (In figure 2.1, the two types are indicated as `B` and `E`.) At the level of application programming in NMM, a data buffer is the usual size of an LDU (see section 1.1). Events can not only occur between the data buffers in a multimedia stream (then they are called *in-stream-events*), but they can also be used as a way of communication between the application and the nodes (as so-called *out-of-band-events*.)

There are `BufferManagers` that can allocate the necessary memory if a node requests a new buffer. Each message can be used multiple times in different paths of the application graph: A reference counter makes sure that the message object is not returned to its BufferManager for later reuse until the last user releases it.

[2]Further information about the state model and the accompanying methods is given at http://www.networkmultimedia.org/NMM/Docs/state.html.

[3]The subclass of `Message` is called `CEvent`, which stands for Compound Event. A `CEvent` can include several `Events`. In the following text we will simply use the term `Event` to distinguish events from data buffers.

Each node has generic methods for the handling of events and processing of data buffers. We have already seen the `Buffer* processBuffer(Buffer*)`-method in the last section. To handle incoming events, each node has an `EventDispatcher`. The node programmer has the ability to define the node's reaction to a certain event by registering event handler functions at its dispatcher. For each incoming event the dispatcher calls the accompanying handler function (if it exists).

Every node has got its own thread, so several nodes can work in parallel. For this reason it makes sense that messages can be stored in a `StreamQueue` at the InputJack. One can choose the maximal number of messages that this queue should be able to store in the node's constructor. There are also different modes for the queue: `MODE_KEEP_OLDEST`, `MODE_KEEP_NEWEST` and `MODE_SUSPEND`. The modes describe the behaviour of the queue if it is full and the predecessor node tries to put in another message. For synchronized applications, the queues should always use `MODE_SUSPEND`. The other modes can cause that messages are discarded. As we will see later these queues can play an important role for synchronization for two reasons:

- *Reduction of jittering:* The message queues form a data storage between two nodes that work in parallel. A sink node can present the data stored in its queue, even if its predecessor node can not produce any new data for a short time.
- *Latency:* Message queues and multiple data buffers in the graph increase the time a buffer needs to "travel" along the graph from source to sink. This can be a problem with time-critical messages: It would not be a good idea to sent a stop-signal for all nodes as an in-stream-event, because it would take too long to get to the sink node.

This section should only introduce the notions referring to NMM that are necessary to understand the main concepts and the synchronization extensions. NMM is still work in progress, and changes to the described notions may occur. Please refer to [1] to get to know more about the current state of the project.

Chapter 3

Synchronization in NMM

This chapter presents the synchronization concepts and components introduced in NMM as part of this practical project.

3.1 Overview

To get a first impression of the new software components we will take a look at a generic flow graph with two synchronized sink nodes.

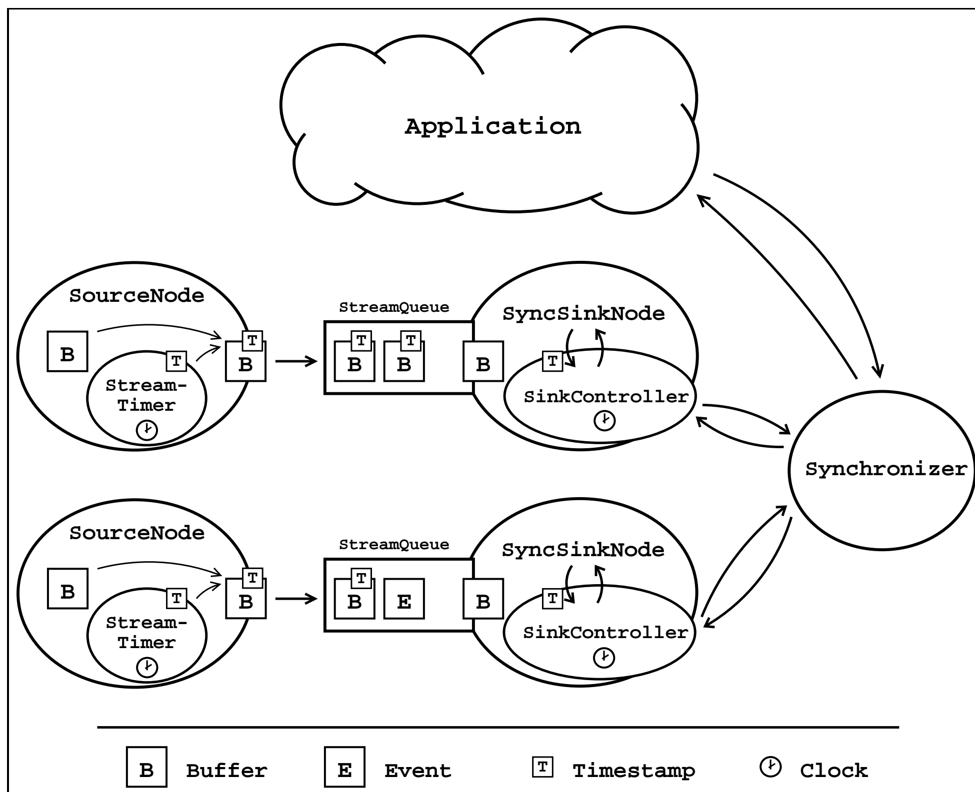


Figure 3.1: General overview of synchronization components

This graph only contains four nodes: two source nodes and two sink nodes. One can see the fundamental objects that will be explained in this chapter: `Clocks`, `Timestamps`, `StreamTimer`, `Controller`, `SyncSinkNodes` and `Synchronizer`.

All components that want to have access to a `Clock` are subclasses from `TimedElement`; they share a common `Clock` that is a static member of `TimedElement`. `Timestamps` contain the time information for a data buffer. The `StreamTimer` is an object that can be used by nodes to simplify timestamp creation. All synchronized sink nodes are subclasses of `GenericSyncSinkNode`. These nodes come along with a `Controller`. They pass the timestamps of incoming buffers to their controller which decides what to do with this buffer: Present it immediately, wait some time until presentation or discard the buffer because it arrived too late at the sink node. One can say that the controller realizes the intra-stream synchronization. If you want to combine multiple streams, the controllers of the sink nodes are connected to a `Synchronizer` object which realizes the inter-stream synchronization. It also serves as an interface for the application which can pause or wakeup the presentation via the synchronizer.

How does the controller decide which buffer to present and which to discard? The main idea is that a controller tries to present all buffers as if they had the same *latency*. Figure 3.2 shows that one can imagine the latency as the time a buffer takes from source to sink. This notion is defined as

$$latency := presentation\ time - sync_time$$

with sync_time as the time value included in a buffer's timestamp. One can also express it the other way round: If a latency is given, the controller tries to present all buffers to the time $sync_time + latency$. The given latency is called *theoretical latency*. In practice the controller has to compute the latency for each incoming buffer (its *real latency*). If the latency exceeds a pre-defined value depending on the theoretical latency, the buffer is too old and declared as invalid. If the real latency is smaller than the theoretical latency, the node has to wait some time until he can present the buffer. The aim of the intra-stream synchronization is to keep the latency constant for the buffers of the stream. A constant latency means that the temporal distance between two buffers is

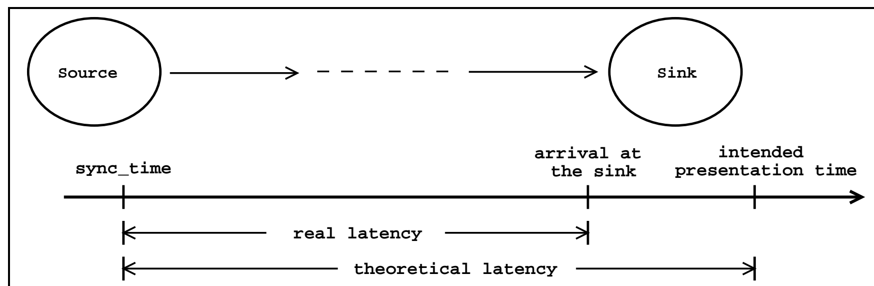


Figure 3.2: The notion of latency

equal to the difference of their sync_time, and so they are presented with the intended velocity.

The controller also computes an average value of the latency of incoming buffers. With this value it is possible to compare the latencies of two or more multimedia streams. This is the part of the synchronizer: If the average latency of a stream changes, the synchronizer computes a new theoretical latency to take these changes into account. The aim of this inter-stream synchronization is that the latencies for the different streams are equal. The easiest way to reach this is to set the theoretical latency to the maximal average latency of all streams: Then the streams with a small latency have to wait until their latency has reached the value from the "slower" ones.

This should give a first impression of how synchronization works in NMM. Before going into details, we first have to take a look at some elementary data structures that are used by synchronization.

3.2 Elementary data types

To represent points of time, time durations and frame rates and to put some time information into a multimedia stream there are some elementary datatypes in NMM.

3.2.1 Rational numbers

All frame rates or time durations in multimedia applications are measured by clocks and are therefore of limited precision. One can express them as rational numbers. Therefore the class `Rational` for representation and basic arithmetic functions of rational numbers has been added to NMM. It stores a rational number as

```
class Rational {
  long int numerator;
  long int denominator;
};
```

Note that the numerator carries the sign information for the fraction; the denominator is always greater than zero. If the denominator reaches the value zero during a computation, a `RationalDivisionByZeroException` is thrown.

3.2.2 Time representation - `Time`, `Interval` and `UserTime`

There are two types for representation of time in NMM. `Time` represents a point of time, `Interval` stands for a duration (which can be considered as the difference between two points of time.) Both `Time` and `Interval` have a precision of one nanosecond. They are internally stored as

```
struct Time {
  long int sec;
  long int nsec;
};
```

```
struct Interval {
  long int sec;
  long int nsec;
};
```

where the variable nsec is bounded by the values 0 and $10^9 - 1$ and sec can take all possible values, both positive and negative. To bound the nsec value makes comparison and elementary arithmetical operations for Time and Interval faster by reducing the number of cases to distinguish. A precision of one nanosecond was chosen to gain compatibility to the OpenML – Standard (see [7]). One should note that the precision of the clocks available at the moment is smaller than one microsecond. We have seen in section 1.3 that such a precision suffices for the main synchronization purposes.

As described in [14] (page 81), it is guaranteed that a long int has at least 32 bits. This ensures that both of the types have a range of about 68 years in the past and the future. At the moment this should be sufficient for every multimedia application.

The main idea behind the separation between points of time and intervals was to gain a typesafe interface for time calculations. A set of standard operators comes along with the two types, and so it should not be necessary to manipulate them at low level.

There is also a type to provide a human readable time representation: UserTime. This is meant to be used in time displays in various applications. A UserTime consists of the following members:

```
struct UserTime {
  int hour;
  int min;
  int sec;
  int msec;
};
```

The precision of one millisecond for UserTime should suffice for the intended usage as time display.

3.2.3 Time information in the multimedia stream - Timestamp

In most of the applications it is necessary to send some time information within the data stream. For this reason the class Message has a Timestamp as member variable. A Timestamp is build up like this:

```
struct Timestamp {
  Time sync_time;
  long int stream_counter;
  bool is_valid;
};
```

The sync_time carries the time information. Usually it marks the intended beginning of the presentation of the buffer's media content. The stream_counter simply counts the data buffers in the stream. The flag is_valid indicates if the time value has been set correctly. This is necessary because you do not always have enough information to set the timestamp for each outgoing buffer correctly. For example, a generic source node that reads different kinds of formats from

a hard disc does not have the knowledge to extract the time information for every format. With this flag it can indicate that a buffer's timestamp contains no useful time information. A specific decoder node then can compute the right time values. You can get and set a message's timestamp with the two methods

```
void setTimestamp(const Timestamp timestamp);
Timestamp getTimestamp();
```

from the class `Message`. Although the timestamps are contained in a general message, they are only in use for data buffers and not for events at the moment (as shown in figure 3.1). The concept of a timestamp is similar to SGI's Unadjusted System Time (see [11] for details).

3.3 Objects in the graph

3.3.1 `Clock` and `TimedElement`

`TimedElement` is the superclass for all objects that want to have access to a time source. All TimedElements of one application share one global and static `Clock` object and therefore have access to the same time. The shared access to one clock is the reason why there is no possibility to change the clock's value. They can get the current time with the method

```
Time getTime();
```

The `Clock` is a static member of the class `TimedElement` and is therefore automatically created and destroyed. At the moment there's only one kind of `Clock` that uses the internal system clock.

On a local system it might be interesting to use another timebase, especially in the context of continuous media played by an internally buffered device. The simplest and also the best example for this situation is given by a sound device. The soundcard chip uses its own internal timebase that is not synchronized with the computers clock. So there can be a slight skew between these two clocks. If we only rely on the computer's internal clock, this skew can be noticeable over a long presentation. This problem could be solved by using the sound devices clock instead of the computers clock as timebase for audio and video.

The first and obvious problem with this approach is that in most of the cases one does not even have direct access to the sound devices internal clock. Another problem appears in context of a network environment: Here the two sink nodes must not necessarily be located on the same computer. The question arises how to synchronize the video sink computer with the clock of the sound device, because this includes synchronization via network connections.

It suffices as a first step to use the computer's hardware clock as timebase. The *Network Time Protocol (NTP)* is used to reduce the clock skew between the different machines in the network.

3.3.2 Timestamp creation - StreamTimer

Nodes that want to create timestamps for a data stream can use a `StreamTimer`. A `StreamTimer` has two different modes, `REAL_TIME` and `CONST_RATE`. You can choose the mode with the method

```
Result setMode(const Mode mode);
```

In the `REAL_TIME` mode the StreamTimer uses the common clock to create the timestamps. This mode is meant to be used in nodes for life-sources like a camera or a microphone. Although most of these sources offer the possibility to choose a frame- or buffer rate, the chosen rates cannot always be kept. So it seems to be better to use the computer's clock to get the timestamp information for incoming buffers instead of relying on what we have chosen as framerate. This is just what the StreamTimer does in `CONST_RATE` mode. You can set a frame rate or an inter-frame-gap, and the timestamps are simply computed as increasing values from 0 on and regarding this constant rate. This can be useful if you know the exact frame rate, e.g. in a video stream read from disc, but the stream does not have timestamps yet. The interval between two following buffers can be set with the two methods

```
Result setRate(const float rate);
Result setInterval(const Interval interval);
```

where the first method uses the buffer rate and the second the interval between two following buffers.

For both of the modes the StreamTimer sets the timestamps into the messages when the following method is called:

```
Result setTimestamp(Message* message);
```

Note that this method has the side-effect of increasing the internal timestamp value of the StreamTimer to make it right for the next buffer. So it should be called only once for each buffer.

3.3.3 GenericSyncSinkNode and its subclasses

The superclass for all sink nodes that are capable to be synchronized is called `GenericSyncSinkNode`. Instead of the `Buffer* processBuffer(Buffer*)` -methods from the other nodes, these have the two methods

```
void prepareBuffer(Buffer* in_buffer);
void presentBuffer(Buffer* in_buffer);
```

In the `prepareBuffer(Buffer*)`-method, all time-wasting preparations for the presentation of the buffer should happen. For example, a video display node could copy the next frame in a shared memory region used by the X window system. In other cases, this method is not used and then simply should not be overloaded.

In the `presentBuffer(Buffer*)`-method the presentation should happen as soon as possible. The idea behind this is that the `presentBuffer(Buffer*)`-method can be triggered right at the time the presentation should happen. The video display node sets the flag for presentation of the next frame in this method. Surely this is only an approximation: There is always a small duration between the `presentBuffer(Buffer*)`-call and the actual presentation, but this time interval is neglected.
The method

```
setSynchronized(bool);
```

can be used to turn the synchronization on and off. Per default it is turned off, so that all applications that do not use synchronization do not have to care about it.

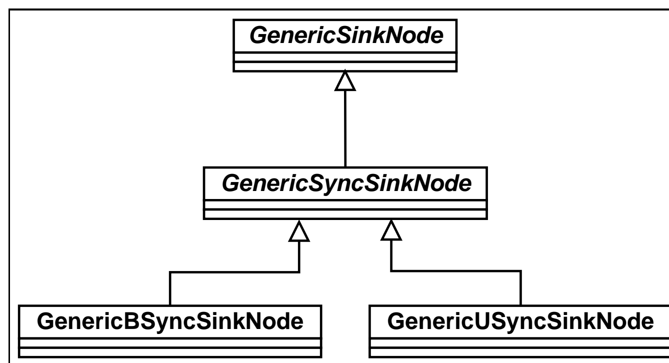


Figure 3.3: `GenericSyncSinkNode` in the NMM class hierarchy

As shown in figure 3.3, there are two subclasses for `GenericSyncSinkNode` with names that only differ in one letter: `GenericBSyncSinkNode` stands for *Generic Buffered Synchronized Sink Node* and `GenericUSyncSinkNode` stands for *Generic Unbuffered Synchronized Sink Node*.

This differentiation has been made to allow for the fact that some sink nodes can present multimedia data directly while other nodes just give the data to low level hardware drivers with an own internal buffer. An audio playback node is an example for such a buffered sink node. Not the node itself sends the data to the soundcard's chip, but it only writes the data to the sound driver's internal cache memory. So there is a delay between the write command of the node and the actual presentation of the audio data. This delay depends on the amount of data that is stored in this buffer at the time you write the new data in and is limited by the total size of this buffer. To take this into account the `GenericBSyncSinkNode` has got a method called

```
Interval getOutputDelay();
```

that returns the period of time between a writing command and the presentation of the written data at this moment. The computation of this returned value will be further explained at the example of the `PlaybackNode` in section 4.4.2.

We have already seen in section 3.1 that each GenericSyncSinkNode has a `Controller` as member variable. The main difference between the *unbuffered* and *buffered* node is that they have different controllers. These will be further explained in section 3.3.4.

The following code sample taken out of GenericSyncSinkNode shows what such a node does with an incoming buffer:

```
timestamp = in_buffer->getTimestamp();

if( !(controller -> isBufferValid(timestamp))) {

  // if the buffer is too old or invalid, release it
  in_buffer -> release();
} else {

  // otherwise, prepare it for presentation
  prepareBuffer(in_buffer);

  // wait until the time for presentation has come ...
  suspendThread(controller -> waitToPresent(timestamp));

  // and present it!
  presentBuffer(in_buffer);
}
```

3.3.4 `Controller` and its subclasses

As we have seen in the last section, each GenericSyncSinkNode has its own `Controller` which tells the node what to do with an incoming buffer: discard it, present it directly or present it after waiting some time. This controller is a member of its parent node and therefore a local object from the node's point of view. So the communication between node and controller can use simple and fast function calls. Like the corresponding sink node types there are different classes for *Buffered Sink Controller* and *Unbuffered Sink Controller*.

We have already described some of the internal parameters like real latency in section 3.1. Now a more detailed description will be given.

The controller gets the timestamp of each incoming buffer with the call of the method `bool isBufferValid(Timestamp)` by the GenericSyncSinkNode shown in the source code example above. We now explain what the controller does in this method:

For each incoming buffer the controller computes the real latency as the difference between the earliest possible presentation time and the buffer's sync time. An `USinkController` assumes that the node has no internal buffer and can present the data immediately. The controller therefore uses the current time as approximation of the presentation time and computes the real latency as follows:

$$real_latency \quad := \quad current_time \; - \; sync_time.$$

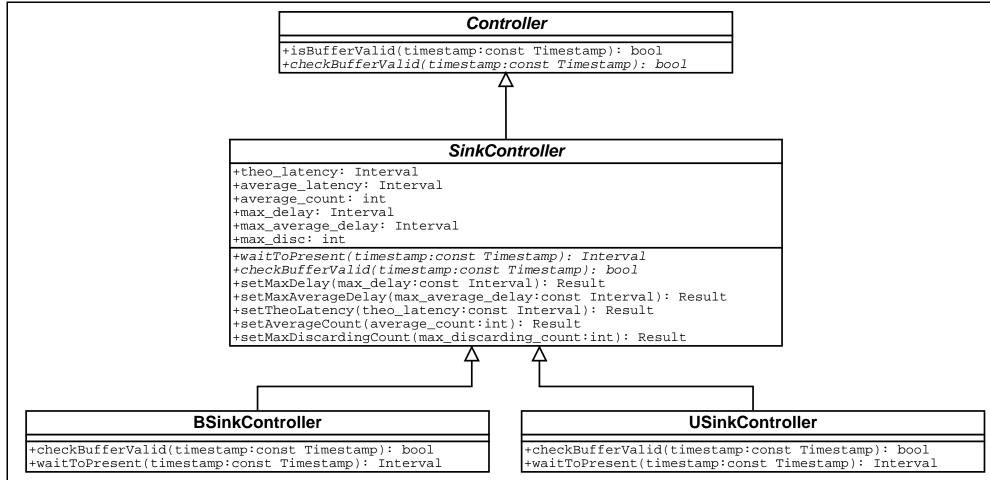


Figure 3.4: The class hierarchy for `Controller` and its subclasses

A `BSinkController` assumes that its parent node has an internal buffer. A data buffer can not be presented before the presentation of the internal buffer's data has been finished. We will call the duration of this presentation the *output_delay*. Therefore a `BSinkController` uses the formula

$$\mathit{real_latency} \quad := \quad \mathit{current_time} \; + \; \mathit{output_delay} \; - \; \mathit{sync_time}$$

to compute the real latency of each buffer.

After the computation the real latency is used to decide whether the buffer should be presented or discarded: If the real latency is greater than

$$\mathit{theo_latency} + \mathit{max_delay}$$

the buffer is declared as invalid. Here *max_delay* is the limit for the maximal delay of one single buffer, and *theo_latency* is a value set by the synchronizer (see section 3.3.5). *Max_delay* can be set with the member function

```
Result setMaxDelay(Interval);
```

If we want to keep the requirements for lip synchronization in section 1.3.2, a typical value for *max_delay* is 80 milliseconds. The *theo_latency* is updated by the synchronizer with the member function

```
Result setTheoLatency(Interval);
```

Both of the controller types compute an average value out of the incoming buffer's real latencies. If $r_1, \ldots, r_{average_count}$ are the real latencies for the last *average_count* buffers, then the average latency is computed as

$$\mathit{average_latency} \quad := \quad \frac{r_1 + \ldots + r_{average_count}}{\mathit{average_count}}.$$

The number of buffers that are taken into account for such a computation can be set by the member function

```
Result setAverageCount(int);
```

The number of buffers surely has an influence on the reaction time of the synchronization system. Typical values are 10 buffers for video streams and 20 buffers for audio streams.

Similar to the real latency, the controller also checks if the average latency is greater than $theo_latency + max_average_delay$. In this case it calls the method

```
Result setAverageLatency(Interval, SinkController*);
```

from the synchronizer. This has no influence on the question if the last buffer is considered as valid or invalid. The average latency value is used by the synchronizer to update the value for the theoretical latency (see section 3.3.5). Typical values for the *max_average_delay* range from 20 to 40 milliseconds. A BSinkController also informs the synchronizer if its average latency is getting smaller. For example, this can happen if the soundcard's internal clock is a bit slower than the computer's clock and the sample rate is not kept exactly. So the synchronizer has to take this into account to keep audio and video together.

Not every buffer that has been declared as invalid so far should be discarded. To avoid discarding of too much immediately following buffers the controller counts how many buffer it has considered as invalid after each other. If this number exceeds the value *max_disc*, the buffer will be presented even if it is too old. This makes sense for a video presentation: If each third frame has to be presented, one gets an impression of the motions in the video even if not every frame is displayed. This value has no influence on discarding all buffers in the state `WAIT_FOR_RESET` (see section 3.4.2). It is set with the member function

```
Result setMaxDisc(int);
```

Note that a value of zero causes the controller to declare all buffers as valid. This is useful for audio playback: Discarding an audio buffer causes unwanted noise output.

Now we have seen what a controller does to check if a buffer is valid. The other key method of the controllers is the method

```
Interval waitToPresent(const Timestamp);
```

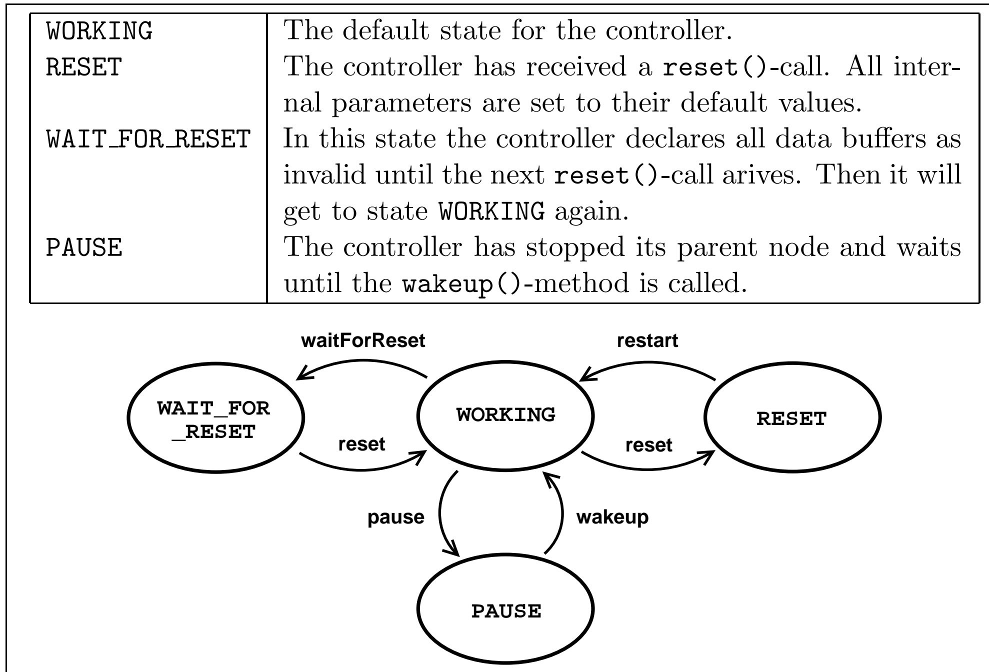
Here a BSinkController always returns zero: The node does not have to wait until the intended presentation time. It can write the data into the internal buffer, because the output delay has already been considered. An USinkController returns the value

$$sync_time \ + theo_latency - current_time$$

and the node waits until the presentation should happen.

The controller is always in one of the states shown in figure 3.5. The state diagram shows which transitions between these states are allowed and which member functions of the controller cause such a transition. None of these functions is called by the application; they are internally called by the parent node and the synchronizer. The states `RESET` and `WAIT_FOR_RESET` are made necessary by the event `sync_reset`, further explanations follow in section 3.4.2.

Note that we are in a multi-threaded environment, so the state variable is made thread-safe with mutual exclusion.



`WORKING`	The default state for the controller.
`RESET`	The controller has received a `reset()`-call. All internal parameters are set to their default values.
`WAIT_FOR_RESET`	In this state the controller declares all data buffers as invalid until the next `reset()`-call arives. Then it will get to state `WORKING` again.
`PAUSE`	The controller has stopped its parent node and waits until the `wakeup()`-method is called.

Figure 3.5: States and state transitions of a `Controller`

3.3.5 `Synchronizer`

So far we have only considered synchronization that refers to one multimedia stream and not introduced the link between multiple streams. This link is constructed via the `Synchronizer` object which is connected to the controllers of all sink nodes that should be kept in sync. The synchronizer realizes inter-stream synchronization it is therefore not related to a special node. It is created once for the application with two or more sink nodes for which it encapsulates the synchronization strategy. One can build a new subclass of `Synchronizer` if the existing ones do not fit to the requirements of the stream constellation in the actual application. At the moment the only subclass of `Synchronizer` is the `AudioVideoSynchronizer` which is designed for general audio/video presentations. The synchronizer is also thought as a central access point for run-time synchronization control: For example, the application can sent pause and wakeup signals to the synchronizer.

The main task of the synchronizer is to keep the average latencies of all connected sink controllers equal. For this reason it has to react to incoming average latency values. If a controller calls the method

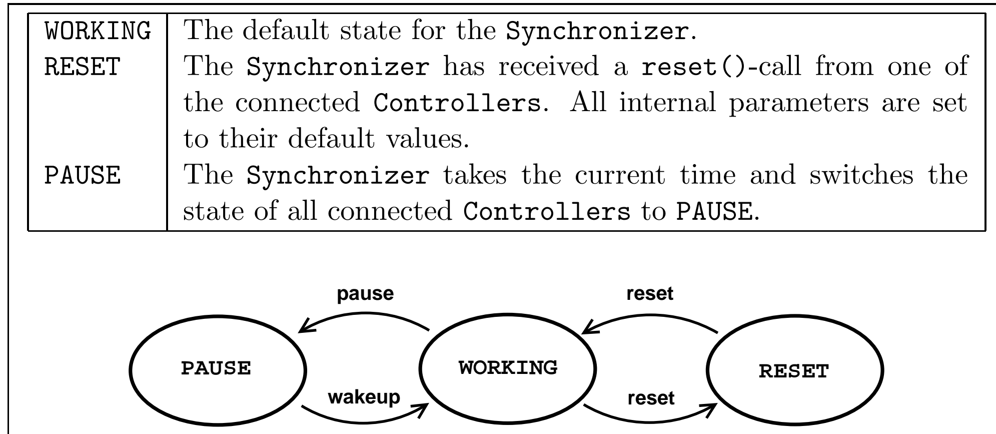
```
Result setAverageLatency(Interval, SinkController*);
```

the synchronizer checks if it is necessary to update the *theo_latency*. We have already seen in section 3.1 that one strategy is to simply take the maximum of the old value for *theo_latency* and the incoming average latency.

The AudioVideoSynchronizer also takes into account that the connected sink nodes present audio and video: Here the theoretical latency is set to the

audio average latency to make the video stream fit to the audio stream.

Like a controller, a synchronizer also has got different states. These are shown in figure 3.6. Note that the `pause()`- and `wakeup()`-methods are called by the application; the `reset()`-method is called internally by the connected controllers. See section 3.4.2 for further information about the state `RESET`.



`WORKING`	The default state for the `Synchronizer`.
`RESET`	The `Synchronizer` has received a `reset()`-call from one of the connected `Controllers`. All internal parameters are set to their default values.
`PAUSE`	The `Synchronizer` takes the current time and switches the state of all connected `Controllers` to `PAUSE`.

Figure 3.6: States and state transitions of a `Synchronizer`

3.4 Events related to synchronization

At the moment there are three events that refer to synchronization. They serve to enable and disable synchronization or reset the internal parameters of synchronization components. They are typically sent in-stream: It is the only way to make sure that they can bring their effects after the presentation of a certain buffer.

3.4.1 `sync_enable` and `sync_disable`

The first two events, `sync_enable` and `sync_disable`, have only an effect on synchronized sink nodes (i.e. all subclasses of `GenericSyncSinkNode`). They have the same functionality as a call of the `setSynchronized(bool)`-method. This way it is possible to disable or enable the synchronization after or before the presentation of a certain buffer. In the constructor of `GenericSyncSinkNode` the two accompanying event handler methods are registered which are called

```
Result eventSyncEnable();
Result eventSyncDisable();
```

These methods simply call the `setSynchronized(bool)`-method with the right arguments.

3.4.2 sync_reset

The third event, sync_reset, is also handled by many other nodes (including MPEGDemuxNode, MPEGVideoDecodeNode, MPEGAudioDecodeNode, AC3DecodeNode etc.). All of these register a handling method called eventSyncReset(). See [1] and [4] to get to know more about these nodes. Section 4.6 describes the synchronization related aspects that are connected with them.

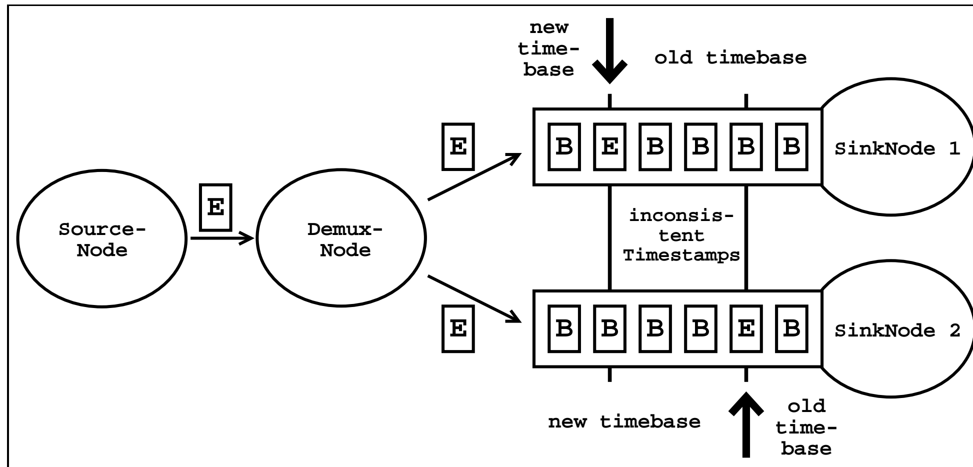


Figure 3.7: Problems with different path lengths

The event sync_reset is used to indicate that the parameters of synchronization should be reset. For example, this is necessary if you switch to another channel on TV or if you choose a new chapter in the DVD application. In both cases, the timestamps in the new MPEG stream are not related to the timestamps read before. Therefore one resets all internal parameters and uses the new values from now on.

Figure 3.7 shows the problem that occurs when an in-stream-event generated by a single source is duplicated by a demultiplexer node. The different path length causes that the both copies of the event appear at the sinks at different times. That can cause problems for all events that need to be worked up at all sinks at the same time. In the case of a sync_reset-event this leads to inconsistent timestamp values: One controller has already reset internal parameters and receives new timestamps, the other still receives old timestamps.

That is the reason for the state WAIT_FOR_RESET: If the first controller handles an event sync_reset, its state switches from WORKING to RESET (also for the synchronizer). The internal parameters are then set to their default values. The synchronizer now calls waitForReset() at the second controller. This controller switches to state WAIT_FOR_RESET and declares all buffers as invalid until the event sync_reset is also received by its parent node. Figure 3.8 shows the participating components in this state. Some buffers with corrupt timestamps are discarded and the presentation can continue with the new stream. The second controller calls restart() at the synchronizer which calls the same method at the first controller. All states turn back to WORKING again.

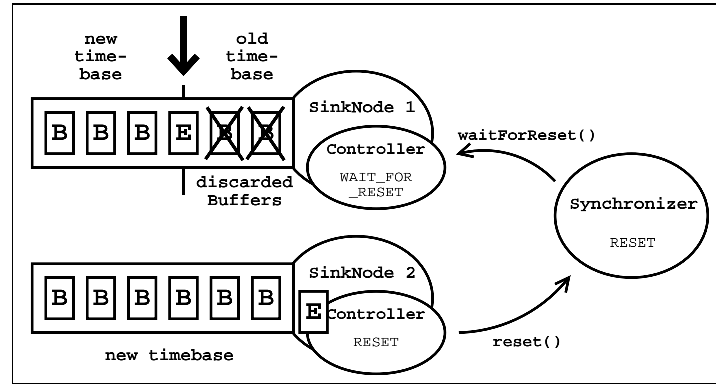


Figure 3.8: Sink nodes, controllers and synchronizer during a `sync_reset`-event

This handling for `sync_reset` has the disadvantage that data buffers are discarded at one sink. Another strategy would be to stop the sink node which receives the first `sync_reset`-event until the second node also receives such an event. This could be done with a conditional variable in the controller that is checked in the method `waitToPresent(Timestamp)`. If the state is `WORKING` or `WAIT_FOR_RESET`, the controller returns the interval the node should wait. In the state `RESET` the node's thread has to wait at the conditional variable until the other sink node receives the event, too. Even if no data is lost with this strategy, it has the disadvantage that one stream has to be stopped. If the presentation of the audio stream is stopped, the observer notices this as an awkward noise.

One could try to avoid this with another approach: Both of the sink nodes do not stop the presentation. The inter-stream synchronization is switched off at the moment the first sink receives the `sync_reset`-event. The second event then switches it on again. This could simply be done with the state `RESET` of the synchronizer: In this state the synchronizer would not react to incoming average latency values.

Chapter 4

Synchronization in Plug-ins and Applications

After describing the underlying datatypes and architecture we will have a look at some practical examples of how synchronization takes place in plug-in nodes and applications. The functionality of these nodes is described rather shortly here. For further information how the nodes work, please refer to [4]. At first we will describe the timestamp handling for some general node classes. We will do this in the order of appearance of the nodes in a flow graph: From source nodes over filter or converter nodes to the sink nodes. We will especially describe the sink nodes as concrete node examples because almost every example application contains one of the presented sink nodes. Then we will have a closer look at two examples for synchronization in application programming: The audio visualization examples have been written as part of this practical project, the MPEG decoding examples were just extended by the synchronization part. See [4] to learn more about the MPEG decoding example and the participating nodes.

4.1 Source nodes

One can distinguish between live sources and "artificial" sources. Live sources get their data from a device that captures it at that moment like a video camera or a microphone. Artificial sources read data that was created before and stored from hard disc, DVD or from other data sources. The main difference according to synchronization is that live sources have to create timestamps for outgoing buffers while the data read from artificial sources needs to contain some time information in order to allow synchronization. Therefore we will have a closer look at the timestamp creation in live sources.

4.1.1 Live sources

The live sources always use a StreamTimer for timestamp creation as described in the last chapter. For this reason the nodes simply create a StreamTimer object in the `REAL_TIME`-mode and tell it to set the timestamp of each outgoing

buffer in the `produceBuffer()`-method. The following code sample shows how this works for a plug-in node called `Source`:

```
Source() {
  // ...
  StreamTimer timer = new StreamTimer(StreamTimer::REAL_TIME);
}

~Source() {
  // ...
  delete timer;
}

Buffer* produceBuffer() {
  Message* out_buffer = getNewBuffer(out_buffer_size);
  // produce the data content and
  // put it into the out_buffer
  timer->setTimestamp(out_buffer);
  return out_buffer;
}
```

An example for such a live source is the `RecordNode` that uses the OSS audio driver (see [2]) to read audio data from the sound device. Another live source node example is the `GenericFireWireNode` which can be used to record a video stream from a camera connected to the IEEE 1394 bus.

4.2 Filter nodes

This class of nodes is characterized by the fact that the format of incoming and outgoing data and buffer quantities are equal. For example, a node that displays a logo in the upper left corner of each incoming video frame belongs to this cathegory. Another obvious example is a sound effects node. Both of them sent one buffer with the same format for each buffer they receive.

These nodes normally do not need to know something about presentation time or duration of incoming buffers. They only copy the timestamp of an incoming buffer into the corresponding outgoing buffer they produce. This is done automatically by the member function

```
Buffer* getNewBuffer(const size_t size, Buffer* in_buffer);
```

which is defined in the class `IExternalBufferManager` and returns a new buffer with the expected `size` and the same timestamp as the `in_buffer`.

4.3 Converter nodes

It is characteristic for a converter node that its input format is not equal to its output format. This holds for a large spectrum of nodes. Not all of the

converters have to change something at the buffer's timestamps: One may think of a node that converts video frames from an YUV colorspace to RGB. From the synchronization point of view, these examples behave like the filter nodes in the last section.

The example shown above only deals with raw formats: There are also compressed formats like MPEG or AVI which come along with their own format specific time information. Some nodes have to convert this time information into values that can be used as timestamps in NMM or otherwise. The other difference to a filter node is that for a converter node there is not always a one-to-one correspondence between incoming and outgoing buffers.

In section 4.5 we will describe two nodes for which the assumption of such a correspondence fails: the audio visualization nodes. In section 4.6 we will focus on the nodes taking part in MPEG decoding.

4.4 Sink nodes

Most of the changes due to synchronization have been done in the sink nodes. Some of these changes have been explained already in the last chapter, but now we will see their influence on already existing nodes.

4.4.1 Video sinks

At the moment there are three video sinks available for NMM which are inherited from `GenericUSyncSinkNode`: The `XDisplayNode`, the `MatroxDisplayNode` and the `GLDisplayNode`. The XDisplayNode is the usual display node that communicates with the X server while the MatroxDisplayNode uses a special device and driver. See [4] to get to know more about these sink node. The GLDisplayNode finally uses OpenGL to display incoming frames. The synchronization related part of these nodes is very similar. With the inheritance from `GenericUSyncSinkNode` there is no need for big changes in the plug-in nodes.

The main idea is to copy the next frame into a suitable memory region in the `prepareBuffer(Buffer*)`-method. For the XDisplayNode and the MatroxDisplayNode this is done by a real memory copy operation. They copy the image data into an X shared memory or a special memory region of the driver. The GLDisplayNode creates a texture out of the data in this method.

The `presentBuffer(Buffer*)`-method then just gives the command to change the presented image. The nodes do this with special functions from the X server, the Matrox driver or OpenGL.

4.4.2 Audio sinks

The NMM audio sink is called `PlaybackNode`. It uses the Open Sound System (OSS) audio drivers to get access to the sound device. See [2] for details about how to use the OSS drivers.

In this case we have an example for a buffered sink node since the OSS driver allocates an internal data buffer. Therefore PlaybackNode is a subclass of `GenericBSyncSinkNode`.

Besides the extensions for synchronization, the PlayBackNode has been extended by automatical recognition of supported audio formats. Raw audio formats are build up with the following parameters:

- sample rate: The number of samples per second. Usual values are between 8kHz and 96kHz.
- bit per sample: The number of bits that build one sample. Normally this value will be 16.
- number of channels: This number can vary between 1 and 6.
- sample representation: This refers to the byte order depending on the processor architecture and is often refered as *big endian* or *little endian*.

The PlaybackNode automatically tests which combinations of these values are supported by the soundcard.

To get synchronized we need some methods to get to know the delay caused by the data in the internal buffer. Because the sampling rate does not change while the driver is playing this is equivalent to get to know the number of bytes that are stored in the internal buffer at the moment. The OSS drivers offer different `ioctl()`-calls to solve this problem. The are called `GETOSPACE`, `GETOPTR` and `GETODELAY`. I have made the experience that they are not equivalently. The `GETODELAY`-call gave the best results. All of the functions have one problem in common: The internal buffer is separated in so-called *fragments* that are used by the driver for double-buffering to avoid a data leak at the soundcard. When you start to write audio data to the device, the driver starts the presentation after the first fragment is filled. The delay values returned by the three methods are zero until the presentation has been started, although data has been already written to the device. Therefore it is sensible to work with small fragment sizes. Per default the driver uses two fragments with 65536 bytes per fragment for all sound cards we have tested. The PlaybackNode tries to configure the driver to use an internal buffer of up to 16 fragments with 4096 bytes per fragment. The sound driver can override these values depending on the hardware's capabilities. The internal buffer size means that $16 * 4096/(48000 * 2 * 2) = 0.341$ seconds of audio can be stored in this buffer with a sample rate of 48 kHz, stereo and 16 bit per sample. We note that the consideration of this internal buffer plays an important role to gain audio/video synchronization.

With the number of bytes *bytes_in_buffer* in the internal buffer one can simply compute the duration *t_buf* of playing the internal buffer's content:

$$t_buf = \frac{bytes_in_buffer}{sample_rate \cdot bytes_per_sample}$$

Here the value *bytes_per_sample* is defined as

$$bytes_per_sample = number_of_channels \cdot bit_per_sample.$$

If we write a data buffer to the device at the time *now*, the observer will hear it at the time $now + t_buf$. So this interval *t_buf* is computed and returned by the `getOutputDelay()`-method.

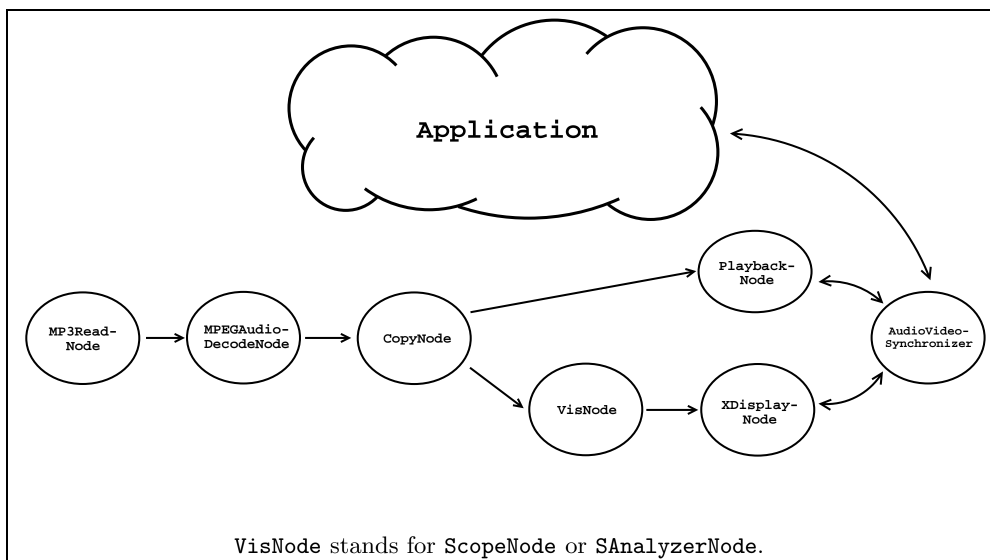
See [2] and the source code file `PlaybackNode.cpp` in the NMM distribution for more details about the programming internals.

In the PlaybackNode there is nothing to do for a `prepareBuffer` - method, and it is therefore left empty. The only thing the PlaybackNode does with incoming audio data is to write it to the sound device in the `presentBuffer(Buffer*)`-method.

4.5 Audio visualization

4.5.1 The mp3vis and mp3vis2 applications

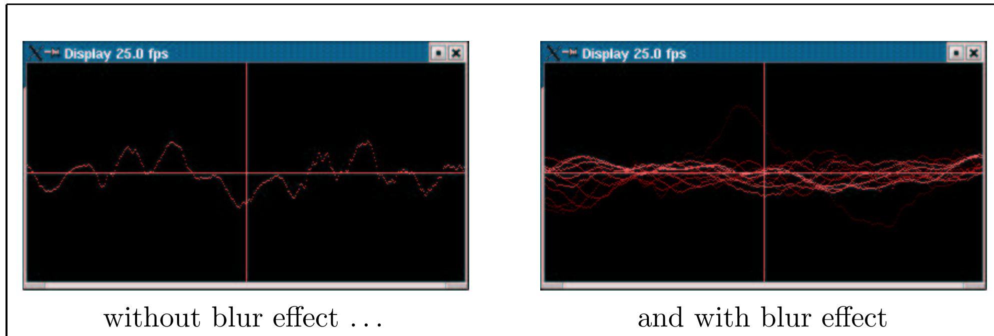
There are two example applications that play MPEG 2 Layer 3 encoded audio files and visualize the audio data. Both of the applications are build up with six nodes, and their graphs only differ in the node used for visualization. The application `mp3vis` uses the `ScopeNode` while `mp3vis2` makes use of a `SAnalyzerNode`. Figure 4.1 shows the graph of `mp3vis` or `mp3vis2`, respectively. You simply need to replace VisNode with a ScopeNode or a SAnalyzerNode to get the graph of `mp3vis` and `mp3vis2` out of it.



`VisNode` stands for `ScopeNode` or `SAnalyzerNode`.

Figure 4.1: The graph of the MP3 visualization examples

The source node is an `MP3ReadNode` to read the compressed audio data from the hard disc. This reader is connected to an `MPEGAudioDecodeNode` that decodes the MP3-compressed data into raw audio data. After the decoder the multimedia stream has to be split up into two streams, one for the audio output and one for the visualization. This is done by a `CopyNode` which simply sends incoming buffers to both of its output jacks. One of them is connected directly to the `PlaybackNode`, an audio sink node which uses the OSS audio driver. The other output jack leads to the `VisNode`. The `VisNode` itself is connected to the video sink, the `XDisplayNode`. The `MPEGAudioDecodeNode` and the `VisNode` are the two nodes in this graph that create and set the timestamps into the



without blur effect ... and with blur effect

Figure 4.2: Screenshots of mp3vis (ScopeNode)

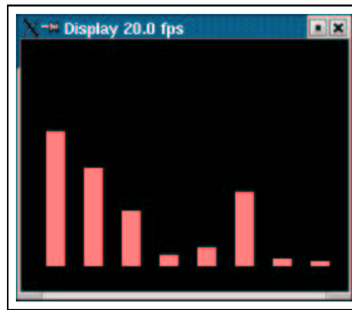


Figure 4.3: Screenshot of mp3vis2 (SAnalyzerNode)

buffers. The `MPEGAudioDecodeNode` is further described in the section 4.6, and we will at first describe the `VisNodes`.

4.5.2 `ScopeNode` and `SAnalyzerNode`

Both of these nodes create a raw video data stream out of a raw audio stream. The `ScopeNode` draws points in a coordinate system with the time as x-value and the audio sample value as y-value. This gives the impression of watching the audio signal on an oscilloscope. With the two methods

```
Result setBlurEnabled(bool)
Result setBlurStep(int step)
```

one can add a blur effect to the visualized signal. `step` describes the intensity of the blur effect. Small values at a range from 1 to 10 cause a long blur effect, great values up to 40 a shorter duration.

The `SAnalyzerNode` shows the components of the audio signal at different frequencies. To compute these values a Fourier transform of the audio data must be computed. For this purpose the RFFTW library is used. Details about this library and its API can be found in [10] where also an example similar to the code in `SAnalyzerNode.cpp` is shown. The screenshots in this section (figure 4.2 and 4.3) should give an impression about what the nodes do.

The nodes use a StreamTimer in the `CONST_RATE`-mode to set timestamps to the outgoing buffers.

4.5.3 Synchronization in the application code

We will now take a look at the application source code to understand what an application programmer has to know about synchronization. At first the synchronization must be switched on for all sink nodes. Then an AudioVideoSynchronizer is created and the sinks are registered to it.

```
XDisplayNode* display = new XDisplayNode(...);
display -> setSynchronized( true );

PlaybackNode* playback = new PlaybackNode(...);
playback -> setSynchronized( true );

AudioVideoSynchronizer* sync = new AudioVideoSynchronizer();
sync -> setVideoSink( display );
sync -> setAudioSink( playback );
```

These few lines of code suffice in these two examples to get a synchronized audio/video presentation. Additionally the two examples offer a pause-functionality which simply uses the two function calls `sync -> pause();` and `sync -> wakeup();`

4.6 MPEG decoding

In figure 4.4 we see the flow graph for a general MPEG video player similar to the graph shown in figure 2.2. By replacing the source nodes and with small modifications this graph can be used to play MPEG encoded files from the hard disc, from DVD's or to watch TV over DVB. The timestamp handling in the participating nodes is a bit more complicated as in the nodes we have seen before.

The source node in this graph is the `MPEGReadNode`. This node only reads data content and does not write any timestamp into a buffer. The `MPEGDemuxNode` splits the MPEG stream into the video and the audio components. This node has the ability to read the time information encoded in the MPEG stream and convert them into `Time` objects. The node stores the first time value read during a presentation and subtracts it from all following values before putting them into the buffers. So the `sync_time` values of the buffers start at zero. At the level the `MPEGDemuxNode` regards the stream, there is not enough information to mark each audio or video packet. That is the point which makes the `is_valid`- flag in the timestamps necessary (see section 3.2.3). The `MPEGDemuxNode` marks all buffers without a valid timestamp with a flag set to false.

The successor nodes in the graph are the `MPEGVideoDecodeNode` and the `MPEGAudioDecodeNode`. Depending on the audio format in the MPEG stream an `AC3DecodeNode` is used instead of an `MPEGAudioDecodeNode`.

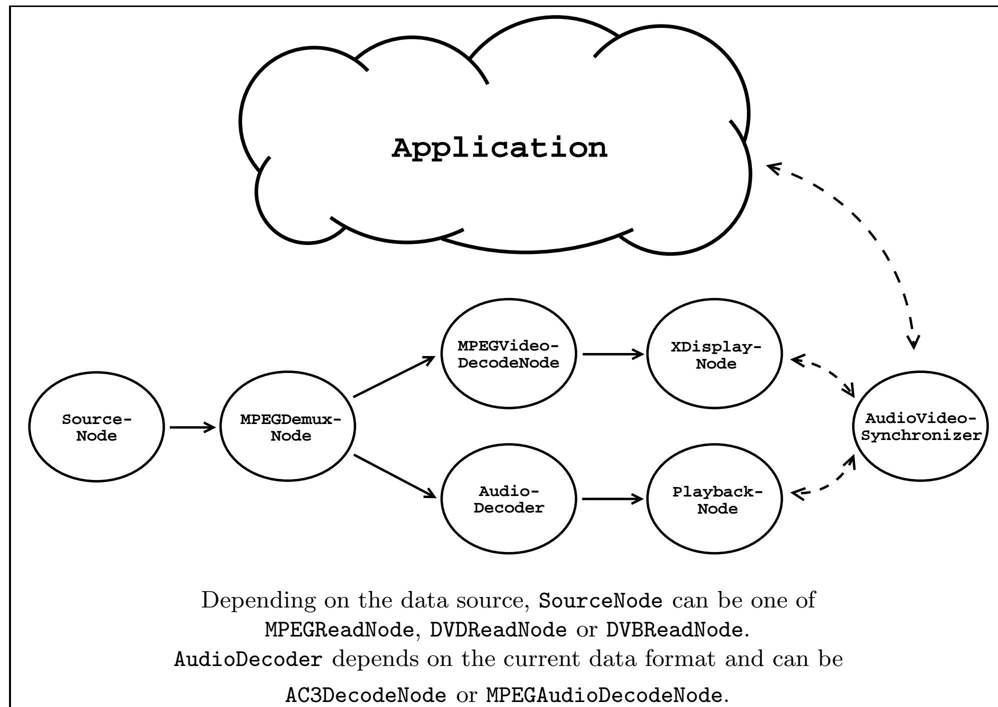


Figure 4.4: A general flow graph for MPEG decoding applications

Creating the timestamps for the audio data is easy: If the incoming buffer contains a valid timestamp, take this for the next outgoing buffer. Store the last written time internally and add the duration for each outgoing buffer to this value to get the time for the next buffer.

The `MPEGVideoDecodeNode` can extract the current duration of the presentation for one video frame out of the MPEG data. Together with the timestamps of the incoming buffers this suffices to reconstruct the timestamps for all video frames. What makes this a bit more complicated is that the compressed MPEG data contains the frames in another order as they are finally displayed. This makes sense because the MPEG format supports inter-frame compression. For this reason an MPEG video stream is divided into *groups of pictures* or *GOP*'s that consist of `I`-, `P`- and `B`-frames. The `I`-frames are encoded `I`ndependent of other frames where `P`- and `B`-frames are encoded related to the information of other frames. The `P`-frames only use forward prediction while the `B`-frames use bidirectional prediction. See [6], page 167, for more details. The library that is used by the node internally changes the order of the frames so that the output is in right order for presentation. Therefore one looses the correlation between the incoming and the outgoing buffers and does not know how to associate timestamps with outgoing buffers. The `MPEGVideoDecodeNode` solves this problem by only taking the incoming timestamps for the `I`-frames. These frames are decoded immediately after their data has been given to the library. The timestamps for `P`- and `B`-frames are generated using the last `I`-frame's timestamp. This does not cause a significant loss of precision because groups of pictures have a limited maximal length. Usually they will contain up to 12 pictures.

Then the next group of pictures will start and the next exact timestamp out of the original stream will be taken.

Besides this we have already discussed in section 3.4.2 that selecting a new DVD chapter or TV channel makes it necessary to reset all internally stored time values in the nodes above. This is done when a `sync_reset`-event occurs.

Appendix A

Source Files

There is not enough space here to show the whole source code that has been implemented for the practical project. I will only give a list of the source files which can be found in the NMM software distribution[1].

The classes that build the underlying architecture are located in a special directory (nmm/base/sync). These files contain:

- nmm/base/sync/Rational.hpp
- nmm/base/sync/Rational.cpp
- nmm/base/sync/Types.hpp
- nmm/base/sync/Types.cpp
- nmm/base/sync/Clock.hpp
- nmm/base/sync/Clock.cpp
- nmm/base/sync/TimedElement.hpp
- nmm/base/sync/TimedElement.cpp
- nmm/base/sync/Controller.hpp
- nmm/base/sync/Controller.cpp
- nmm/base/sync/SinkController.hpp
- nmm/base/sync/SinkController.cpp
- nmm/base/sync/BSinkController.hpp
- nmm/base/sync/BSinkController.cpp
- nmm/base/sync/USinkController.hpp
- nmm/base/sync/USinkController.cpp

[1]The NMM software is available for download at http://www.networkmultimedia.org.

- nmm/base/sync/GenericSyncSinkNode.hpp
- nmm/base/sync/GenericSyncSinkNode.cpp
- nmm/base/sync/GenericBSyncSinkNode.hpp
- nmm/base/sync/GenericBSyncSinkNode.cpp
- nmm/base/sync/GenericUSyncSinkNode.hpp
- nmm/base/sync/GenericUSyncSinkNode.cpp
- nmm/base/sync/Synchronizer.hpp
- nmm/base/sync/Synchronizer.cpp
- nmm/base/sync/AudioVideoSynchronizer.hpp
- nmm/base/sync/AudioVideoSynchronizer.cpp

I wrote some examples for timestamp handling in plug-in nodes and synchronization in application programs (described in chapter 4.5). These include the following files:

- nmm/plugins/audio/visualization/ScopeNode.hpp
- nmm/plugins/audio/visualization/ScopeNode.cpp
- nmm/plugins/audio/visualization/SAnalyzerNode.hpp
- nmm/plugins/audio/visualization/SAnalyzerNode.cpp
- nmm/examples/mp3dec/mp3vis.cpp
- nmm/examples/mp3dec/mp3vis2.cpp

Besides these, synchronization handling has been added to many of the plug-in nodes that already existed before. Most of the extensions consisted only of a few lines of additional code. I would only like to mention the sink nodes and especially the OSS audio sink node because there were significant changes to the source code (see also chapter 4.4.2). It can be found in the files

- nmm/plugins/audio/PlaybackNode.hpp
- nmm/plugins/audio/PlaybackNode.cpp

List of Figures

1.1 Intra-object synchronization 4
1.2 Inter-object synchronization 5

2.1 Nodes and Messages 7
2.2 A simple node graph for an MPEG video player 9

3.1 General overview of synchronization components 11
3.2 The notion of latency 12
3.3 `GenericSyncSinkNode` in the NMM class hierarchy 17
3.4 The class hierarchy for `Controller` and its subclasses 19
3.5 States and state transitions of a `Controller` 21
3.6 States and state transitions of a `Synchronizer` 22
3.7 Problems with different path lengths 23
3.8 Sink nodes, controllers and synchronizer during a `sync_reset`-event 24

4.1 The graph of the MP3 visualization examples 29
4.2 Screenshots of mp3vis (ScopeNode) 30
4.3 Screenshot of mp3vis2 (SAnalyzerNode) 30
4.4 A general flow graph for MPEG decoding applications 32

Bibliography

[1] *Network-Integrated Multimedia Middleware (NMM).* Project's homepage: http://www.networkmultimedia.org.

[2] 4Front Technologies. *Open Sound System Programmer's Guide*, 2000. http://www.opensound.com.

[3] Be, Inc. *The Be Book for BeOS Release 5*, 2000.

[4] Patrick Becker, Patrick Cernko, Wolfgang Enderlein, Marc Klein, and Markus Sand. *Design and Development of a Multimedia Home Entertainment System for Linux.* Universität des Saarlandes, 2002. Advanced practical project.

[5] Gordon Blair and Jean-Bernard Stefani. *Open Distributed Processing and Multimedia.* Addison-Wesley, 1. edition, 1998.

[6] John F. Koegel Buford. *Multimedia Systems.* ACM Press, 1. edition, 1994.

[7] Steve Howell. *OpenML Version 1.0 Final Specification.* Khronos Group, 2001. http://www.khronos.org/openml.

[8] Keith Jack. *Video Demystified.* LLH Technology Publisher, 2. edition, 1996.

[9] Helmut Kopka. *LaTeX – Einführung Band 1.* Addison-Wesley Verlag, 3. edition, 2000.

[10] Massachusetts Institute of Technology. *FFTW Tutorial (for Version 2.1.3)*, 1999. http://www.fftw.org.

[11] Chris Pirazzi. *Introduction to UST and UST/MSC.* Internet site: http://www.lurkertech.com/lg/time/intro.html.

[12] Ralf Steinmetz. *Multimedia-Technologie.* Springer-Verlag, 2. edition, 1999.

[13] Ralf Steinmetz and Klara Nahrstedt. *Multimedia: Computing, Communications and Applications.* Prentice Hall, 1. edition, 1995.

[14] Bjarne Stroustrup. *Die C++ Programmiersprache.* Addison-Wesley Verlag, 4. edition, 2000.

[15] W3C. *Synchronized Multimedia Integration Language (SMIL 2.0)*, 2001. http://www.w3.org/TR/smil20.