

Network-Integrated Multimedia Middleware (NMM) : Basic Introduction

**Motama GmbH, Saarbruecken, Germany
(<http://www.motama.com>)**

April 2010

Copyright (C) 2005-2010
Motama GmbH, Saarbruecken, Germany
<http://www.motama.com>

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being all sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found in the file COPYING.FDL.

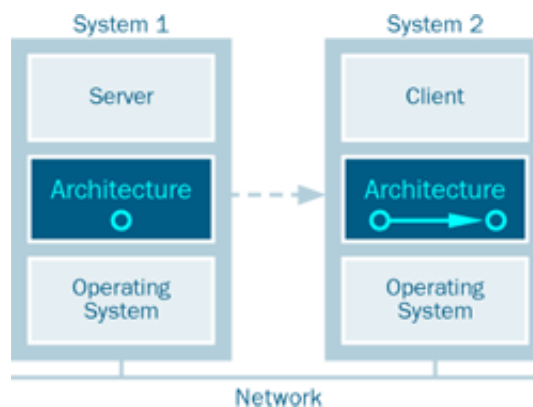
THE DOCUMENT IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE DOCUMENT BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE DOCUMENT OR THE USE OR OTHER DEALINGS IN THE DOCUMENT.

This document gives a basic introduction to the design of the Network-Integrated Multimedia Middleware (NMM). NMM is a multimedia middleware and considers the network as an integral part and enables the intelligent use of devices distributed across a network. Therefore, it can be used as enabling technology for locally operating multimedia applications, but more importantly for all kinds of networked and distributed multimedia systems - spanning from embedded and mobile systems, to PCs, to large-scale computing clusters.

1. Introduction

Besides the PC, an increasing number of multimedia devices -- such as set-top boxes, PDAs, and mobile phones -- already provide networking capabilities. However, today's multimedia infrastructures adopt a centralized approach, where all multimedia processing takes place within a single system. The network is, at best, used for streaming predefined content from a server to clients. Conceptually, such approaches consist of two isolated applications, a server and a client. The realization of complex scenarios is therefore complicated and error-prone -- especially since the client has typically no or only limited control of the server.

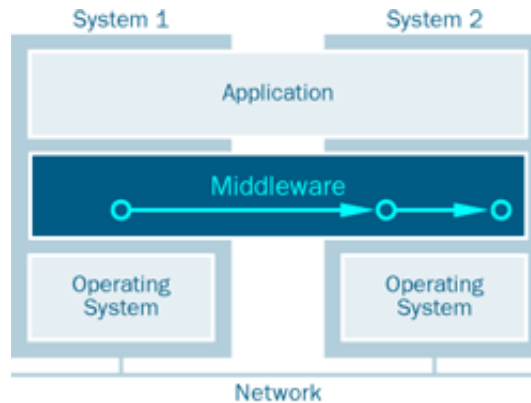
Figure 1. Client/server streaming consists of two isolated applications that do not provide fine-grained control or extensibility:



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

Figure 2. A multimedia middleware is a distributed software layer that eases application

development by providing transparency:



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

NMM is both an active research project at Saarland University in Germany and an emerging Open Source project. NMM runs on a variety of operating systems and hardware platforms (please refer to NMM FAQ for details). NMM is implemented in C++, and distributed under a dual-license: NMM is released under 'free' licenses, such as the GPL, and commercial licenses.

2. General Design Approach

2.1. Nodes, Jacks, and Flow Graphs

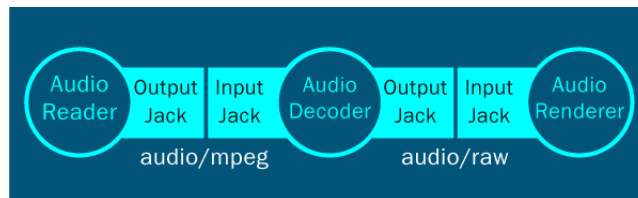
The general design approach of the NMM architecture is similar to other multimedia architectures. Within NMM, all hardware devices (e.g. a TV board) and software components (e.g. decoders) are represented by so called *nodes*. A node has properties that include its input and output ports, called *jacks*, together with their supported multimedia *formats*. A format precisely defines the multimedia stream provided, e.g. by specifying a human readable type, such as 'audio/raw' for uncompressed audio streams, plus additional parameters, such as the sampling rate of an audio stream. Since a node can provide several inputs or

outputs, its jacks are labeled with tags. Depending on the specific kind of a node, its innermost loop produces data, performs a certain operation on the data, or consumes data.

Our system distinguishes between different types of nodes: A *source* produces data and has one output jack. A *sink* consumes data, which it receives from its input jack. A *filter* has one input and one output jack. It only modifies the data of the stream and does not change its format or format specific parameters. A *converter* also has one input and one output jack but can change the format of the data (e.g. from raw video to compressed video) or may change format specific parameters (e.g. the video resolution). A *multiplexer* has several input jacks and one output jack; a *demultiplexer* has one input jack and several output jacks. Furthermore, there is also a generic mux-demux node available. A guide (http://www.motama.com/nmmdocs_plugins.html) provides step-by-step instructions on how to develop of a new node.

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

Figure 3. A flow graph for playing back MP3 files:



Together, more than 60 nodes are already available for NMM, which allows for integrating various input and output devices, codecs, or specific filters into an application. A complete list of available nodes is available online (http://www.motama.com/nmmdocs_features.html).

2.2. Messaging System

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

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

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

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

2.3. Interfaces

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

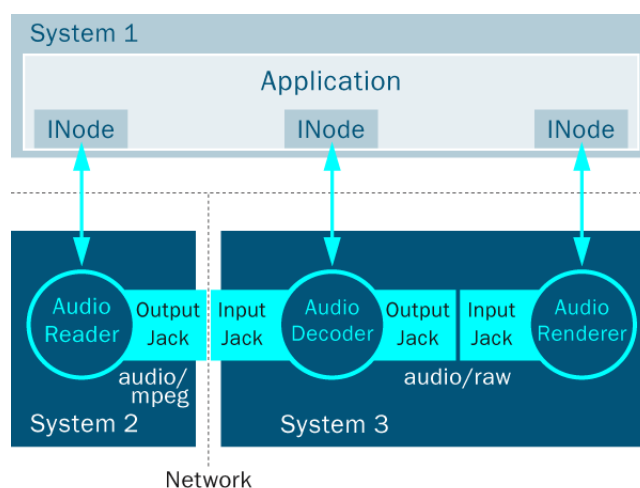
2.4. Distributed Flow Graphs

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

The following shows an example for a distributed flow graph for playing back encoded (compressed) files, e.g. MP3 files. A source node for reading data from the local file system (AudioReader) is connected to a node for decoding audio streams (AudioDecoder). This decoder is connected to a sink node for rendering uncompressed audio using a sound board (AudioRenderer). Once the graph is started, the source nodes reads a certain amount of data from a given file, encapsulates it into buffers, e.g. of size 1024 bytes, and

forwards these buffers to its successor. After being decoded to uncompressed 'raw' audio, the converter node forwards data buffers to the sink node.

Figure 4. A distributed flow graph for playing back MP3 files:



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

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

2.5. Distributed Synchronization

Since NMM flow graphs can be distributed, they allow for rendering audio and video on different systems. For example, the video stream of an MPEG2 file can be presented on a large screen connected to a

PC while the corresponding audio is played on a mobile device. To realize synchronous playback of nodes distributed across the network, NMM provides a generic distributed synchronization architecture (http://www.motama.com/nmmdocs_time.html). This allows for achieving lip-synchronous playback as required for the above described setup. In addition, media presentations can also be performed on several systems simultaneously. A common application is the playback of the same audio stream using different systems located in different rooms of a household -- a home-wide music system.

The basis for performing distributed synchronization is a common source for timing information. We are using a static clock within each address space. This clock represents the system clock that is globally synchronized by the Network Time Protocol (NTP) (<http://www.ntp.org/>) and can therefore be assumed to represent the same time basis throughout the network. With our current setup, we found the time offset between different systems to be in the range of 1 to 5 ms, which is sufficient for our purposes. A primer (http://www.motama.com/nmmdocs_ntp.html) describes how to set up NTP for NMM.

3. Registry Service

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

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

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

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

Before a server registry can be used, it needs to determine which devices and software components are available on a particular host. Therefore, the registry needs to be initialized once using following command.

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

Create config file with plugin information ...
Loading plugins...
AC3DecodeNode          available
AVDemuxNode            available
AVIReadNode            available
... and many further nodes

Config file successfully written.
```