FD – Media Object

Content

History

Version	Date	Adjustments
PA1	2007-08-27	First version of the document. Original document was found in MAS. (EMAHAGL)

1 Introduction

This document describes the Media Object (MO) component.

Basically a MO is a container for media. The format of the media is at it would be stored in a file system.

2 Function Structure

2.1 MediaObject

The MO is used in backend to distribute media between components.

For example:

- Prompts, are represented by MO:s and provided by component Media Content Manager.
- Media from a RTP stream can be recorded in a MO.

The media in a MO is always formatted as it would be in a file system. It is the responsibility of the creator of a MO to set the file format on the MO and format the data correspondingly when injecting it into the MO.

The data in a MO is either injected by a client or is fetched by the MO itself.

2.1.1 Overview

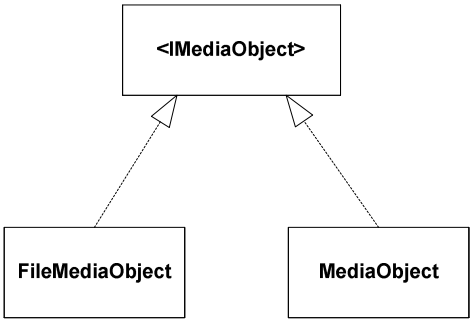

Figure 1 Media Object class diagram

Figure 1 shows a coarse overview of the MO component. There are two distinct types of MOs: The one that fetches its own data from a file (The MediaObject) and the one that gets its data injected (MediaObject).

2.1.1.1 Properties

A MO has a set of properties, as size in byte, length and content-type. The content type property is the most significant as it denotes the actual type of content in the MO and by this the format of the data. The content type is represented as a MIME type, for example the quicktime movie would be represented as the MIME type "video/quicktime". The properties also have a file extension which is the actual extension a MO would have when stored as a file on the file system. In the backend a specific content is always mapped to a specific file extension. For example the a MO above that contains a quicktime movie with content type "video/quicktime" would have file extension "mov".

2.1.1.1.1 Fileformat

A MO's data is always formatted after a specific file format such as WAV and MOV. It is the responsibility of the creator of a MO to set the file format on the MO and format the data correspondingly when injecting it into the MO. If the data is fetched from file by the MO itself, the file is loaded as is, no processing of the data is made. The creator of such a MO must however specify the fileformat along with the path to the file to load. A MO is unaware of what kind of data it is loading.

2.1.1.1.2 Media Codecs

A MO's content type denotes the type of content in the MO but some contents may contain multiple codecs. For example a MOV file can contain video in *video/mpeg* and audio in *audio/pcm.* These codecs are not carried in the MO as properties, so a client must either inspect the data or map the content type to a specific set of codecs.

2.1.2 Memory Data Structure

A MO stores the media data in list of sequential ordered buffers. The reason for storing the data in many buffers is that it makes it possible to keep portions of the data in memory. For example, the FileMediaObject that reads its data from a file will be able to read portions of the file instead of reading it entirely. A MO client that request data doesn't have to wait for the entire file to be read before processing the data. The size of each buffer should be set by the creator of the MO (probably a Factory).

If data is injected into the MO by a client, the client is responsible for injecting each buffer in the correct sequence.

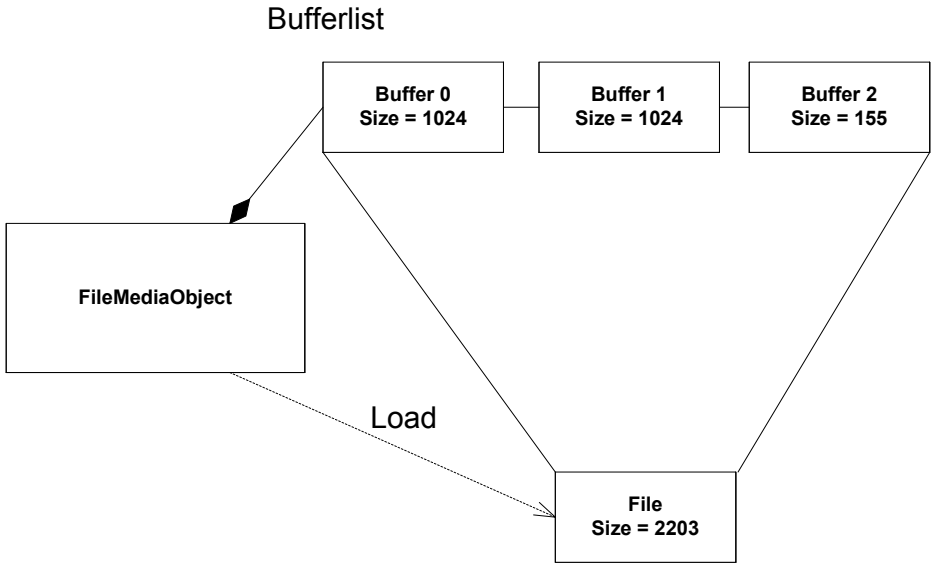

Figure 2 Mapping of file to memory buffer

Figure 2 shows how FileMediaObject load a file and maps it into 3 buffers with default size of 1024 bytes.

2.1.2.1 MediaObject

A MediaObject will store its buffers in the order that the client injects them. The client is responsible for injecting the buffers in an ordered sequence. Once the client has injected the data the MediaObject can be set to be immutable, meaning that no more buffers can be appended to it.

2.1.2.2 FileMediaObject

This section describes how a FileMediaObject loads data from a file. A FileMediaObject is provided with a path to the file to load and will load the file by itself, i.e. the creator is not appending any data. Figure 2 showed how FileMediaObject maps a file into a number of buffers. The number of buffers needed depends on the default (maximum) buffer size. The creator of a FileMediaObject specifies this size.

A FileMediaObject loads only the buffers of data that is asked for. By this a client that is only interested in the first part of a file will only force the FileMediaObject to load the buffers to cover that. A constraint is that the data is loaded sequentially, i.e buffer 0 is always loaded before buffer 1. A client is forced to ask for the data sequentually by the IMediaObject interface, which only provides an iterator for accessing the data. See 3.3 - Read data, for an explanation on this.

Figure 3 shows how a FileMediaObject has mapped only the first 1024 bytes of the file as this is the only data requested so far.

The main purpose of this behavior is to make it possible for a client that request data to begin processing it before the entire file is loaded.

A FileMediaObject is **immutable**, meaning that is not possible for a client to append more buffers to it.

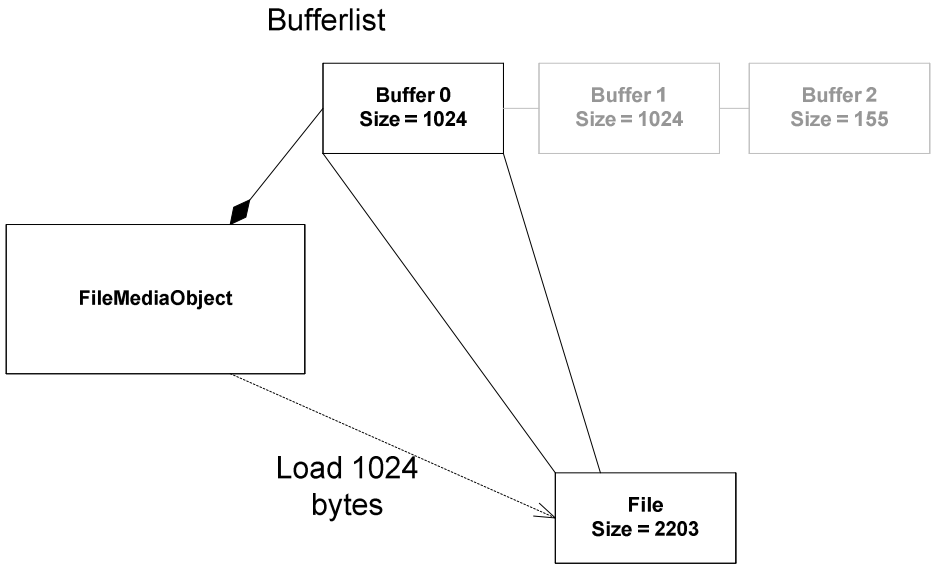

Figure 3 Mapping the 1024 first bytes of the file

2.1.2.3 Direct Byte Buffers

MO uses the *java.nio.ByteBuffer* class to create *direct* byte buffers.

The following is taken from the ByteBuffer Javadoc at
http://java.sun.com/j2se/1.5.0/docs/api/java/nio/ByteBuffer.html

"A direct buffer has the advantage that the Java virtual machine will make a best effort to perform native I/O operations directly upon it. That is, it will attempt to avoid copying the buffer's content to (or from) an intermediate buffer before (or after) each invocation of one of the underlying operating system's native I/O operations direct ByteBuffers as buffer"

The main reason that direct ByteBuffers are used in backend is that JNI supports functionality to get the address to a direct byte buffer. The advantage of this is that no extra copying of the data is necessary when it is passed between Java and C/C++ (JVM and Native side).This is useful for componens written in C++, such as the Stream Component's streaming functionality, as the data can be accessed directly. When the Stream component is streaming a MediaObject it uses JNI to access the direct address of each ByteBuffer in a MO. See [1] for a functional description of the Stream component.

2.2 ContentTypeMapper

2.2.1 Overview

The content type mapper maps one property of a media to another. The methods it provides are:

MimeType mapToContentType(MimeType[] codecs)

String mapToFileExtension(MimeType[] codecs)

MimeType mapToContentType(String fileExtension)

The content type mapper uses its configuration file to create mappings between content types, codecs and file extensions. Figure 4 is an example configuration.

Content Type Mapper configuration

```
<contenttypemapper>

   <contenttype mimetype="video/quicktime">
      <codec mimetype="video/h263"/>
      <codec mimetype="audio/pcmu"/>
      <fileext name="mov"/>
   </contenttype>

   <contenttype mimetype="audio/wav">
      <codec mimetype="audio/pcmu"/>
      <fileext name="wav"/>
   </contenttype>

   <contenttype mimetype="text/plain">
      <codec mimetype="text/plain"/>
      <fileext name="txt"/>
   </contenttype>

</contenttypemapper>
```

Figure 4 Content Type Mapper configuration

Each *contenttype* entry is identified by its *mimetype*. For each content type there is one or many codecs defined. Each codec is also identified by its *mimetype*. For a content type there is also one file extension, *fileext*.

During initialization, the content type mapper reads its configuration and sets up mappings between codecs to content types, file extensions to content types, and content types to file extensions.

3 Function Behavior

3.1 MediaObject creation

For client outside the Media Object module the creation of MediaObjects will be simplified by using a Media Object Factory. The factory will always return the interface IMediaObject to hide the specific implementation of the IMediaObject interface that is really created. The factory will choose implementation based on the create method the client calls. Table 1 shows which concrete implementation of IMediaObject that is created when different create-methods is called. Note that the fileformat of the data must be given when data is injected.

Table 1 Actual implementation created by mediaobject factory

Factory Method	Concrete Implementation	Comment

create()	MediaObject	Creates an empty MediaObject. Data can be injected to it.
create(Buffer[])	MediaObject	Creates an immutable (no more data can be injected) MediaObject injected with the passed buffers.
Create(InputStream)	MediaObject	Creates an immutable (no more data can be injected) MediaObject that is injected with data read from the specified InputStream.
create(File)	FileMediaObject	Creates an immutable (a client can not inject data into it) MediaObject that will load data from the file sequentially when it is requested.
create(String)	MediaObject	Creates an immutable MediaObject that will hold the passed string.

3.2 Injecting data

As could be seen in Table 1, the only way for a client to inject data **after** creation is to create an empty Media Object (the MOs that is injected with data at creation-time is set to immutable). Data is injected into a MO with the *append(buffer)* method. The client **must** append the buffers in the correct sequence or else a consumer of the data will get it in the wrong order.

3.3 Read data

The data in a MO can be read in two ways, either as a stream of bytes from an InputStream, or in a more raw fashion by retrieving the actual ByteBuffer's that holds the data.

3.3.1 Read from InputStream

To read data in a MO with an InputStream is straight forward. Just retreive the InputStream from the MO and use the read methods on it. The InputStream implementation will hide the data structure details of the MO from the client.

3.3.2 Read direct from buffers

In some cases the actual byte buffers that holds the data in the MO is needed when data is to be read. This is the case when data is read from the native side over JNI. Therefore a MO provides an iterator that iterates over the buffers.

The iterator returned from a MO supports the function *next* and *hasNext.* A call to next will return the next ByteBuffer in the iteration.

Figure 5 shows how a client retreives data from a MO. It checks if any more data is available with the *hasNext* method, if so it retreives the next ByteBuffer in the iteration and pulls the data from it.

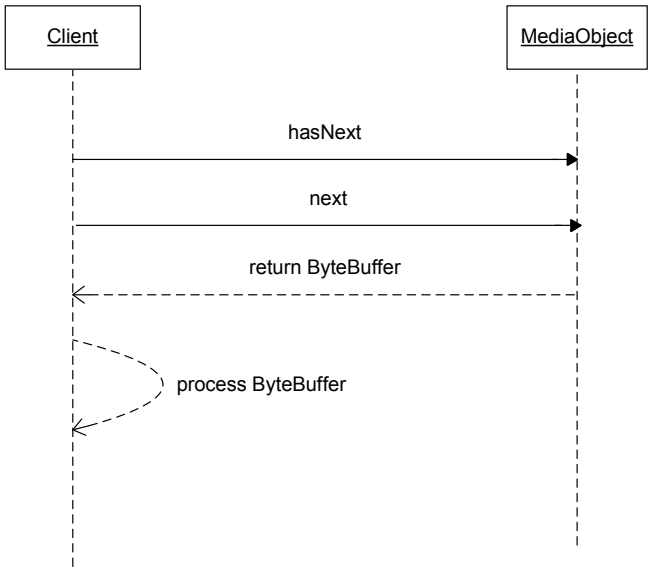

Figure 5 Retrieving data from a Media Object

4 References

[1] FD – Stream
26/0363-PRO049 Uen

5 Terminology

NIO	Non-Blocking IO
JNI	Java Native Interface