Buffer Management in NMM

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

April 2010

This document describes the buffer management framework of NMM. It provides instructions for using buffer managers in plug-ins and transport strategies and for defining new buffer managers using custom memory allocation strategies or different memory pools.

1. Overview

This document describes the buffer management framework of NMM and provides instructions and examples on using and extending it. Section 2 describes the concepts of buffer management in NMM. Section 3 describes how to use buffer managers in plug-ins and transport strategies. Section 4 describes how to define custom buffer managers.

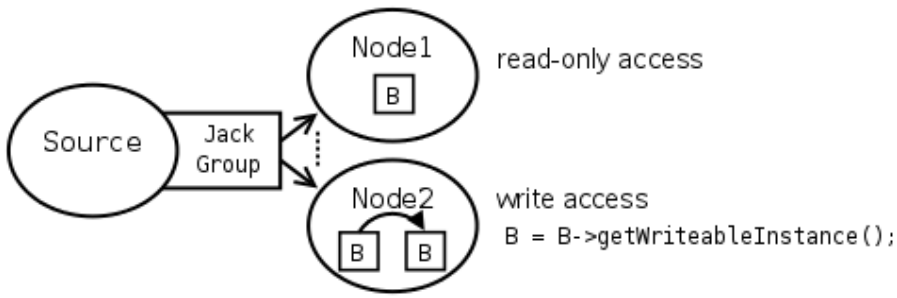
2. Buffer Management in NMM

This section describes the concepts of buffer management in NMM. It assumes basic knowledge of the messaging system and flow-graph architecture of NMM. It describes the reference counting and copy-on-write patterns used by NMM buffers and the concept of buffer managers.

2.1. Buffers

NMM messages (buffers and composite events) provide an interface for realizing reference counting. In particular, the methods `increaseCount()` and `release()` provide the basic operations for this mechanism. Furthermore, NMM provides *buffer managers* that administrate a certain amount of memory and offer the interface for requesting a buffer with a particular size. Such a buffer then initially has a reference count of one. Whenever this counter reaches a value of zero, the buffer is returned to its corresponding buffer manager.

Figure 1. Passing a Buffer to multiple Nodes



When streaming buffers between connected co-located processing elements of a flow graph, only the corresponding references are forwarded; no copy operation is performed (see Figure 1). However, since a single buffer might be forwarded to multiple receivers (e.g. when using a jack group), special care must be taken. First of all, the reference count needs to be increased according to the number of receivers. Then, additional checks must be provided for accessing the memory area of such a buffer. While concurrent read-only access imposes no limitations, writing access needs to be restricted. In order to efficiently realize this functionality, we chose to use the same instance of a buffer as long as possible, and only to create a copy of a buffer if explicitly requested. This feature is realized via the method `getWriteableInstance()`.

```
/** Returns writeable instance of this buffer.
 *
 * This method returns a writeable instance of the buffer.
 * Since Buffer-objects can be shared along the flowgraph,
```

```
 * an operation, which modifies the data of the Buffer-object,
 * must call this method before getData().
 * Typical use:
 * @code
 * myBuffer = myBuffer->getWritableInstance();
 * char *data = myBuffer->getData();
 * @endcode
 *
 * @return pointer to writeable instance of this buffer
 */
Buffer *getWriteableInstance();
```

This method requests a new buffer and creates a copy of the current buffer, if the reference count of the buffer is greater than one, i.e. the buffer is concurrently used within at least two different branches of the flow graph. If not, the buffer itself is returned.

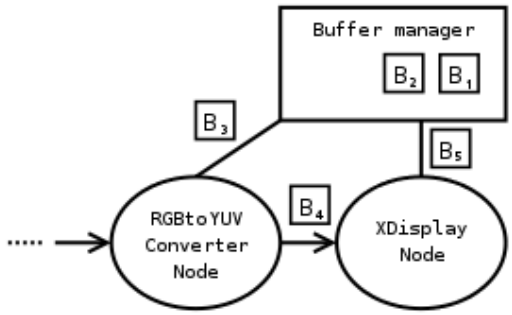
This allows all parts of the flow graph that only require read-only access to the buffer to operate on the same instance; a copy is only generated for nodes that explicitly need to modify the contents of the buffer. Notice that this also includes modifications of the header information, e.g. timestamps. Since converters, such as encoders or decoders, typically do request a new buffer for transforming incoming data to outgoing, copy operations are mainly performed for concurrently operating filter nodes that operate in-place, i.e. using the memory area of the incoming buffer.

2.2. Buffer Managers

In order to reduce the number of allocation and deallocation operations, a number of preallocated buffers is administrated by a buffer manager. If all buffers were already requested, new buffers are allocated upon request. However, a buffer manager then tries to limit the total amount of allocated memory by deallocating returned buffers when this amount is exceeded. Since packet sizes of multimedia streams can be irregular, the sizes of requested buffers vary. To avoid unnecessary allocations of memory, a buffer out of the pool of available buffers is also returned if its size is larger than the requested size. Therefore, NMM buffers also allow to set and query the used amount of memory of buffer objects.

In addition, plug-in nodes can provide specific implementations of buffer managers. This allows to provide access to memory areas administrated by software libraries or device drivers used as a basis within plug-in nodes. For example, the X window system offers shared memory pages. Access to such specific buffer managers then further reduces the number of time-consuming operations.

Figure 2. Buffer Managers



As a further optimization, NMM allows to share buffer managers between different co-located nodes. For example, the buffer manager provided by a node for rendering video via the X window system can be used by its predecessor. Figure 2 shows this scenario with a color space converter as predecessor of the sink node. In this case, the sharing of buffer managers allows to directly use the memory provided by the X window system within the converter node. Therefore, no copy operation at all is performed when streaming data between these two nodes.

Buffer managers are also automatically shared between nodes and components that perform the deserialization of buffers, in particular transport strategies configured within a communication channel.

2.3. Events and Composite Events

For composite events, no such services are provided by NMM. These objects are created manually and are copied when being forwarded to multiple receivers. This choice was made due to following observations. First, composite events store very specific data, namely different events; reuse of memory needed for events is not applicable in this case. Secondly, composite events allow to insert, modify, and remove internal events. Providing such features in the presence of concurrent access would require to copy composite events upon certain conditions, e.g. the deletion of an event within another branch of the flow graph. Implementing this feature is complicated and would require coarse-grained locking mechanisms.

Together, this effort would not be worthwhile the possible improvements of the performance - this is especially true when considering the comparatively small amounts of memory needed for composite events and events. However, if larger amount of memory are to handled by events, our current implementation would need to be extended.

3. Using Buffer Managers

This section describes how to use buffer managers. The instructions and source code examples in this section are useful to plug-in developers and developers of transport strategies.

3.1. Using Buffer Managers in Plug-in Nodes

Every node has its own buffer manager from which it allocates buffers. For the convenience of the plug-in developer, the class `GenericNode` provides its own interface for allocating buffers. The `getNewBuffer` method can be used within the `processBuffer` method of a node class for allocating a buffer from the node's buffer manager.

```
Message* MyNode::processBuffer(Buffer* in_buffer)
{
  // ...

  buffer* out_buffer = getNewBuffer(size);
  if(!out_buffer) {
    // Failed to allocate buffer!
    return 0;
  }

  // ...

  return out_buffer;
}
```

Please refer to the document "Developing Plug-ins for NMM" for more information about implementing plug-in nodes. It contains many code examples that also do buffer allocation.

The `GenericNode` class, like all components that generate buffers, implements the messaging interface `IExternalBufferManager`. This interface (defined in `nmm/comm/messaging`) provides not only the `getNewBuffer` method but also methods for getting and setting the buffer manager used by the component.

```
/** Sets BufferManager.
 *
 * Use this buffer manager. The buffer manager passed will be returned
 * in getBufferManager and will be used for all getNewBuffer calls.
 * This method MUST be called before the node's init method if called
 * from the outside. The subclass itself should use this method from
 * within its own setup method to change/set the buffer manager. If
 * no one sets a buffer manager, a basic Allocator will be created on
 * the first call of getNewBuffer.
 *
 * @param manager The new buffer manager to use.
```

```
 */
void setBufferManager(BufferManager *manager);

/** Gets used BufferManager.
 *
 * Get this node's buffer manager. If possible, a subclass should use
 * this method to check for the presence of a buffer manager before
 * it installs its own.
 *
 * @return The buffer manager currently in use or 0 if non such exists.
 */
const BufferManager *getBufferManager() const;
```

By using the `getBufferManager` method, a node can access its own buffer manager. Initialization or configuration methods of the node can set a new buffer manager by calling `setBufferManager`. The following example (taken from the `XDisplayNode` plug-in in the open source version of NMM) shows how to set a custom buffer manager as the buffer manager of a node:

```
void XDisplayNode::useSharedMemoryManager(unsigned int shared_pages)
{
  // Set number of shared memory pages
  shared_memory_pages = shared_pages;

  // Unset the old buffer manager
  if(buffermanager) {
    setBufferManager(0);
  }

  // Create a new buffer manager
  if (shared_memory_pages > 1) {
    buffermanager = new ShmBufferManager();
  }
  else {
    buffermanager  = 0;
  }

  // Set the buffer manager to be used
  setBufferManager(buffermanager);

  // Initialize the buffer manager
  if (image) {
    image->updateMemoryManager(buffermanager, shared_memory_pages);
  }
}
```

The method shown in this example can be called through an interface to enable or disable the use of the shared memory buffer manager. It is also called by the initialization methods of the node, because the shared memory buffer manager (`ShmBufferManager`) is used by default.

3.2. Using buffer Managers in Transport Strategies

A transport strategy is a component of the communication framework of NMM which implements a certain transport mechanism, such as a network protocol for transmitting NMM messages. A transport strategy which receives NMM buffers over a network connection must allocate these buffers in the same way as a source node does. Therefore, a transport strategy that receives buffers must have access to a buffer manager.

Transport strategies, similar to nodes, implement the `IExternalBufferManager` interface. See Section 3.1 for details. Please refer to the document "Communication Framework of NMM" for more information about implementing transport strategies. The subsection "Buffer Management" of the section "Message Transport" explains how to use the buffer manager in a transport strategy.

3.3. Sharing a Buffer Manager

A node in an NMM flowgraph may share a buffer manager with another node in the same flowgraph. This has been described in Section 2.2. An application can explicitly enable the sharing of a buffer manager by using the `IBufferManager` interface. This interface is implemented by all node classes and all other distributed components that have buffer managers (note that `IBufferManager`, as opposed to `IExternalBufferManager`, is a true NMM interface for distributed communication. It is defined in `nmm/multimedia`).

```
/** Set the buffer manager to use by this node.
 *
 * This method configures this node to use the given buffer manager.
 * The given buffer manager must belong to a node which is local to this node,
 * otherwise this method will fail.
 *
 * @param manager Interface to the buffer manager to use
 * @return @c SUCCESS if the method succeeded, otherwise @c FAILURE
 */
Result useBufferManager(in IBufferManager manager);
```

As can be seen, the method accepts another `IBufferManager` interface as its argument. Essentially, this method tells the component on which it is called to use the buffer manager of the other component which is identified by the second `IBufferManager` interface. The following example (taken from the Multimedia Box application) shows how to use this method:

```
#include "nmm/multimedia/IBufferManager.hpp"

// Use the buffer manager of the display node (display)
// for allocating buffers created by an decoder (decoder)
IBufferManager_var bm1(display->getParentObject()->
  getCheckedInterface<IBufferManager>());
IBufferManager_var bm2(decoder->getParentObject()->
  getCheckedInterface<IBufferManager>());
```

```
bm2->useBufferManager(*bm1.get());
```

In this example, the application has two interfaces of type `INode` of two subsequent nodes in a flowgraph (a video decoder and a display sink). It requests an `IBufferManager` interface to each of the nodes and then calls `useBufferManager` to tell the video decoder to allocate its buffers of uncompressed video data using the buffer manager of the display sink.

The above will work only if the two nodes are local to each other (i.e. within the same address space). Otherwise, the one node can not use the buffer manager of the other node, and the `useBufferManager` method will fail.

Note that buffer managers are automatically shared between transport strategies and nodes. Consider a distributed flowgraph consisting of an audio source and an audio sink with a network edge in between. The receiving transport strategy allocates buffers and fills them with data received from the network. These buffers are allocated using the buffer manager of the audio sink. This design has been chosen for network transparency. In this example, it ensures that, by default, the buffer manager of the sink node will be used, regardless whether there is a network edge before it or not.

4. Writing Custom Buffer Managers

This section describes how to write a custom buffer manager. Custom buffer managers can be used in applications that require different strategies of memory allocation or allocate memory from a memory pool other than system memory. The implementation of a shared memory buffer manager for Linux is used as an example. [1]

4.1. Requesting and Releasing Buffers

A custom buffer manager is a subclass of the `BufferManager` class. This class can be found in the directory `nmm/comm/messaging`. Every class that realizes a custom buffer manager must at least implement the following virtual methods:

```
/** Requests a buffer.
 *
 * Request an unused buffer of a certain minimal size.
 * If the buffer manager is able to comply with the request
 * (by reusing a buffer that is large enough, or by allocating
 * a new one), such a buffer is returned. Otherwise, null
 * is returned. This call does not block until an adequate
 * Buffer is available.
 *
 * The reference count is already increased to one inside
 * this function. A Node should NOT increase the reference
```

```
 * count of the returned Buffer unless it is really passing
 * it on multiple times!
 *
 * @param size the minimal size of the buffer.
 * @return a pointer to a Buffer object, or null if no buffer is available.
 */
virtual Buffer* requestBuffer(const size_t size) const = 0;

/** Release a buffer.
 *
 * This is called from the Buffer class when
 * the corresponding buffer's reference count has dropped to 0.
 * @param buffer The buffer to be released.
 */
virtual void releaseBuffer (Buffer* buffer) const = 0;
```

The `requestBuffer` method implements allocation of a new buffer, and the `releaseBuffer` method implements deallocation of a buffer whose reference count has reached zero. Note that the pool-based allocation strategy, described in Section 2.2, is not implemented in the base class. If a custom buffer manager should use a similar allocation strategy, it must provide its own implementation. The default buffer manager class, `B_Allocator`, may serve as a reference. It can be found in the same location as the `BufferManager` class. Of course, a custom buffer manager may also use any other allocation strategy that is most suitable for its particular purpose.

The implementation of these two methods is free in the choice of allocation strategy and memory pool to use for allocating buffers. However, there are certain requirements that must be met by both methods. The requirements for the `requestBuffer` method are as follows:

- The method must increase the reference count of the buffer by one. This is achieved by calling `increaseCount(1)` on the `Buffer` object before returning it.
- The method must return a null pointer if buffer allocation fails. No exceptions may be thrown.
- The method must return a null pointer if the `erase` method has been called before. See Section 4.2 for details.
- The method must return a buffer whose size is at least the requested size. If this is not possible, the method must fail. The returned buffer may be larger than requested.

The following example code, taken from the shared memory buffer manager for Linux, shows the general structure of an implementation of the `requestBuffer` method: [2]

```
Buffer* ShmBufferManager::requestBuffer(const size_t size) const {
  MutexGuard mg(mutex);

  if (erase_flag) {
    // This BufferManager will be erased. Deny all requests
    return 0;
  }
```

```
  // Initialize the buffer pool.
  // Allocate as many buffers of the requested size as possible.
  if (buffers.empty() && shm_segment) {
    buffer_size = size;
    for (char* buf = shm_segment;
          buf + size < shm_segment + shm_size;
          buf += size) {
      Buffer* buffer = new Buffer(buffer_size, buf, this);

      // add buffer to list
      buffers.push_front(buffer);

      // add buffer to freebuffers list
      freebuffers.push_front(buffer);
    }
  }

  // Check if a buffer of the requested size is available
  if (size > buffer_size) {
    // No buffer of the requested size available.
    // Either no shared memory has been allocated yet,
    // or the requested buffer size is too large.
    return 0;
  }

  // Check if there are any buffers left
  if (freebuffers.size() == 0) {
    // Out of shared memory buffers
    return 0;
  }

  // Remove buffer from the list of free buffers
  Buffer* buf = freebuffers.front();
  freebuffers.pop_front();

  // Increase the reference count
  buf->increaseCount(1);

  // increase the number of allocated buffers
  ++num_of_buffers;

  return buf;
}
```

This example shows a buffer manager which allocates buffers using memory in a shared memory segment (at address `shm_segment`). The allocation strategy is very simple: When the first buffer is requested, the buffer manager preallocates as many buffers of the requested size as fit into the shared memory segment (whose size is `shm_size`). The request for a buffer fails if the requested buffer size is greater than the size of the preallocated buffers or if there are no more free buffers left or if no shared memory segment is

available yet. Note that the method also increases the reference count of the buffer before returning it, and that it keeps track of the number of allocated buffers.

The requirements for the `releaseBuffer` method are as follows:

- The buffer manager must delete itself if and only if the `erase` method has been called before, and there are no more buffers that have been allocated by the buffer manager. See Section 4.2 for details.

The following example code, taken from the shared memory buffer manager for Linux, shows the general structure of an implementation of the `releaseBuffer` method.

```
void ShmBufferManager::releaseBuffer(Buffer* buf) const {
  MutexGuard mg(mutex);

  // Decrease the count of allocated buffers
  num_of_buffers--;

  // Add buffer to the list of free buffers
  freebuffers.push_front(buf);

  // The object will delete itself if and only if
  // all allocated buffers have released
  // and erase has been called
  if(erase_flag && (num_of_buffers == 0)) {
    // Delete buffer manager
    delete this;
  }
}
```

In this example, the buffer manager adds the released buffer to the pool of free buffers, from where it can be used again by the `requestBuffer` method that has been shown above. The method also keeps track of the number of buffers. Finally, it deletes the buffer manager when the last buffer has been released and `erase` has been called.

4.2. Initialization and Cleanup

Initialization of the buffer manager can be performed in the constructor or in `requestBuffer`. The implementation of the buffer manager is free to choose the right time for initialization.

As described in Section 2.2, multiple nodes can share buffer managers. To achieve this feature, the lifetime of a buffer manager is managed by reference counting. The following methods are invoked by nodes to increase and decrease the reference count:

```
/** Returns pointer to this BufferManager object.
 *
 * @return pointer to this BufferManager object
```

```
 */
virtual BufferManager* request();

/** Decrease reference count and release BufferManager, if it drops to 0.
 */
virtual void release();
```

A custom buffer manager typically does not reimplement these methods. However, the `release` method invokes the `erase` method when the reference count of the buffer manager reaches zero.

```
/** Erase this BufferManager object.
 *
 * This method is protected, since it is called by release.
 */
virtual void erase() const = 0;
```

This method must be implemented by all subclasses of `BufferManager`. It must check if all buffers allocated by the buffer manager have been released and, if so, delete the buffer manager. The buffer manager must not delete itself if there are still buffers allocated by it, as these buffers will call `releaseBuffer` on the buffer manager when their reference count reaches zero.

The following example code, taken from the shared memory buffer manager for Linux, shows the general structure of an implementation of the `erase` method.

```
void ShmBufferManager::erase() const {
  MutexGuard mg(mutex);

  // Set flag to indicate that this method has been called
  erase_flag = true;

  // The object will delete itself if and only if
  // all allocated buffers have released
  // and erase has been called
  if (num_of_buffers <= 0) {
    // Delete buffer manager
    delete this;
  }
}
```

This special reference counting strategy ensures that a buffer manager is deleted if and only if it is no longer used by a node and all buffers have been released. Note that if some buffers allocated from the buffer manager are never released, then the buffer manager itself will never be released, causing a memory leak. However, in such a situation, there already is a memory leak somewhere else. An application should normally guarantee that all buffers are eventually released. If buffers are only used within the flowgraph and not stored externally, this is achieved by simply flushing the graph. References to buffers

stored outside the flowgraph must be eventually released by the component that is responsible for these references.

De-initialization of the buffer manager should be performed in the destructor.

Notes

1. The complete source code of the shared buffer memory manager is provided in the open source version of NMM as the class `ShmBufferManager`. It can be found in the directory `nmm/plugins/video/display/X`.
2. Debug output (and other less relevant code) has been removed to show only the essential parts of the implementation. Also, the complete implementation uses a fallback buffer manager which allocates system memory whenever no shared memory buffer can be allocated. The fallback buffer manager has been removed to keep this example simple. Some comments have been added to clarify the implementation.