States and State Transitions of NMM Nodes

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

April 2010

1. Introduction

One major goal for the development of the NMM architecture is to allow for an easy development of plug-in nodes. The state machine of NMM defines strict guidelines for the life-cycle of a node; from being constructed to the point where data is flowing through the processing element. In order to allow for an integration of various different multimedia processing routines, the state machine needs to be generic enough to cover all different cases that might occur.

2. States and State Transitions

Figure 1. States of a Node

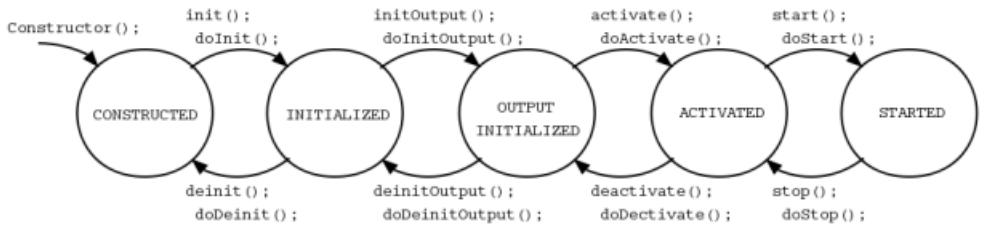

Figure Figure 1 shows the states and methods for performing state transitions. For each of these state transition methods, a do-method is called internally, e.g. for `init()` the method `doInit()`.

This design is according to the Template Method pattern, which delegates specific parts of an algorithm to subclasses that implement a template method that is called by the superclass. Applied to the state machine of NMM, this allows to implement code specific for a particular plug-in to be executed during state transitions.

The state transitions themselves are already provided by the node base class and internally use the facilities of the event dispatcher. To this end, the state machine is defined in NMM IDL by using the keyword `limited` as additional modifier for state transition methods, e.g. the method `init()` is limited to the state `CONSTRUCTED`. For a state transition to complete successfully, the invocation of corresponding state transition method has to be allowed for the current state, all plug-in specific required methods have to be completed, and the corresponding do-method has to be processed with returning SUCCESS and without throwing an exception. Otherwise, an exception is thrown indicating that the state transition failed and the state is not changed.

The different states that specify the life-cycle of a plug-in node are defined as follows.

- `CONSTRUCTED` This state is reached after the creation of a node. The internal status of the node should be initialized within the constructor, which acts as do-method for this state. Within this state, all resources that have influence on the supported formats or features of the node have to be specified but are not allocated. For example, the handle to a particular multimedia device has to be given.
- `INITIALIZED` When reaching this state by calling the method `init()`, resources have to be requested and allocated. Furthermore, the static input and output formats of a node have to be fully specified

within its static input and output property, respectively. Often, these formats depend on the allocated resources, e.g. the formats supported by a particularly chosen multimedia device. In addition, if the number of inputs of the node also depends on configurations to be made in constructed state, input jacks have to be created; otherwise, this can be done during the construction of the node. Please note that certain subclasses of the generic nodes already provide the correct number of input jacks and output jacks, which correspond to the particular node type. For example, a sink node already provides a single input jack and no output jack. Since all these steps are specific for a particular plug-in, they are to be implemented within the `doInit()`-method.

Within this state, input jacks can be requested from the node by specifying a format and optionally the name of the jack if more than one input is available. Only if the name and format is supported, an input jack is provided and the corresponding format is stored internally. A requested input jack can be connected to an output jack to be provided by another node that is within the state `OUTPUT_INITIALIZED`.

- `OUTPUT_INITIALIZED` Upon calling the method `initOutput()`, the working properties of the node have to be specified for reaching the next state. In particular, the working input property has to be filled in respect to the chosen input formats as specified when input jacks were requested in state `INITIALIZED`. Correspondingly, the working output property has to be filled with all output formats that are available for the currently configured input formats. Furthermore, the correct number of output jacks with corresponding names have to be created. Again, this is done by providing a custom implementation within the method `doInitOutput()`. Please note that certain subclasses of the generic nodes already provide the correct number of input jacks and output jacks, which correspond to the particular node type. For example, a sink node already provides a single input jack and no output jack. Output jacks can then be requested by providing a format and optionally the name of the jack. This specification seems to be similar to the requirements for state `INITIALIZED`. However, an additional property of multimedia streams has to be taken into account. Depending on the chosen input formats, the supported output formats and the number of outputs of a processing element can only be determined by considering the actual content of the input streams. For example for multiplexed streams, the number and formats of elementary streams is not known a priori and can only be determined by analyzing the stream of data. A similar property can be observed for streams of compressed data, where the precise output format of the decoding routine is first known after some decompressed buffers were generated.

  However, fully specified output formats are needed in order to be able to connect output jacks and input jacks. Furthermore, the process of analyzing incoming data in order to determine the correct output formats is also needed for handling different multimedia streams with the same flow graph. A simple example for this case would be the playback of several files of the same file format by using a single flow graph. Therefore, the generic processing model of NMM allows to handle both cases transparently. Correspondingly, the development of plug-in nodes is greatly simplified. Together, for a node to analyze data, its predecessors within the flow graph -- also called upstream nodes -- need to provide enough buffers for determining the format. Therefore, all upstream nodes need to be in the state `ACTIVATED` to be discussed next.

- `ACTIVATED` When calling the method `activate()` the input and output formats of all requested jacks are determined and the jacks are connected. Within the method `doActivate()` resources that depend on the chosen formats have to be reserved. From this point on, the node can be started to perform the actual multimedia processing. Since this is also the first time during the life-cycle of a node that

all required resources have been requested, the state `ACTIVATED` needs to be reached for all upstream nodes before a call of `initOutput()` for a specific downstream node can be performed.

- `STARTED` Finally, by calling the method `start()` the internal loop of the node is triggered and the node starts to perform the intended processing by handling messages. If needed, plug-in nodes can provide custom functionality to be performed prior to being started within the method `doStart()`.

As can be seen in Figure Figure 1, state transition methods also exist for reaching preceding states, namely `stop()`, `deactivate()`, `deinitOutput()`, and `deinit()`, together with corresponding template methods. In general, all operations and reservations performed should be reverted during these transitions.

3. State Transitions - Quick Reference

The application programmer can call following methods:

3.1. State Transitions for the Application Developer

Constructor() : is invoked upon creation of node within NMM registry; node specific functionality; no reservation of resources

init() : reserve resources; after init() all generally supported formats have to be specified (the static formats).

getInputJack() (only in state INITIALIZED) : request input jack by specifying format and optionally jack tag.

initOutput() : reserve resources; after initOutput() all supported formats by this specific instance of the node have to be specified (the working formats).

getOutputJack() (only in state OUTPUT_INITIALIZED) : request output jack by specifying format and optionally jack tag.

activate() : reserve resources depending on chosen formats

start() : start processing and forwarding messages (in-stream events and buffers)

setEnabled(true | false) : only for filter node: enable or disable filtering of data in buffers; is true in the beginning

stop() : stop processing and forwarding messages (in-stream events and buffers)

flush() (only in state ACTIVATED or STARTED) : flush incoming queue(s), clear internal buffers, reset library, etc.

deactivate() : free resources allocated in activate

deinitOutput() : free resources allocated in initOutput

deinit() : free resources allocated in init

3.2. Plug-in Development

Instructions for the plug-in programmer: init(), initOutput(), activate(), start(), stop(), deactivate(), deinitOutput(), deinit() and flush() internally call protected methods : doInit(), doInitOutput(), doActivate(), doStart(), doStop(), doDeactivate(), doDeinitOutput(), doDeinit() and doFlush(). The plug-in programmer can implement these do-methods to implement node specific functionality.

3.3. Exceptions and Return Values

Undefined state transitions cause an IllegalStateTransitionException to be thrown. If a do-method does not return SUCCESS, a StateTransitionFailedException will be thrown. It is also allowed to throw a StateTransitionFailedException (or a subclass thereof) in a do-method. We recommend throwing exceptions to indicate errors, rather than returning `FAILURE`.

4. Connecting and Disconnecting Nodes

4.1. Connecting Nodes

Figure 2. States of a Node

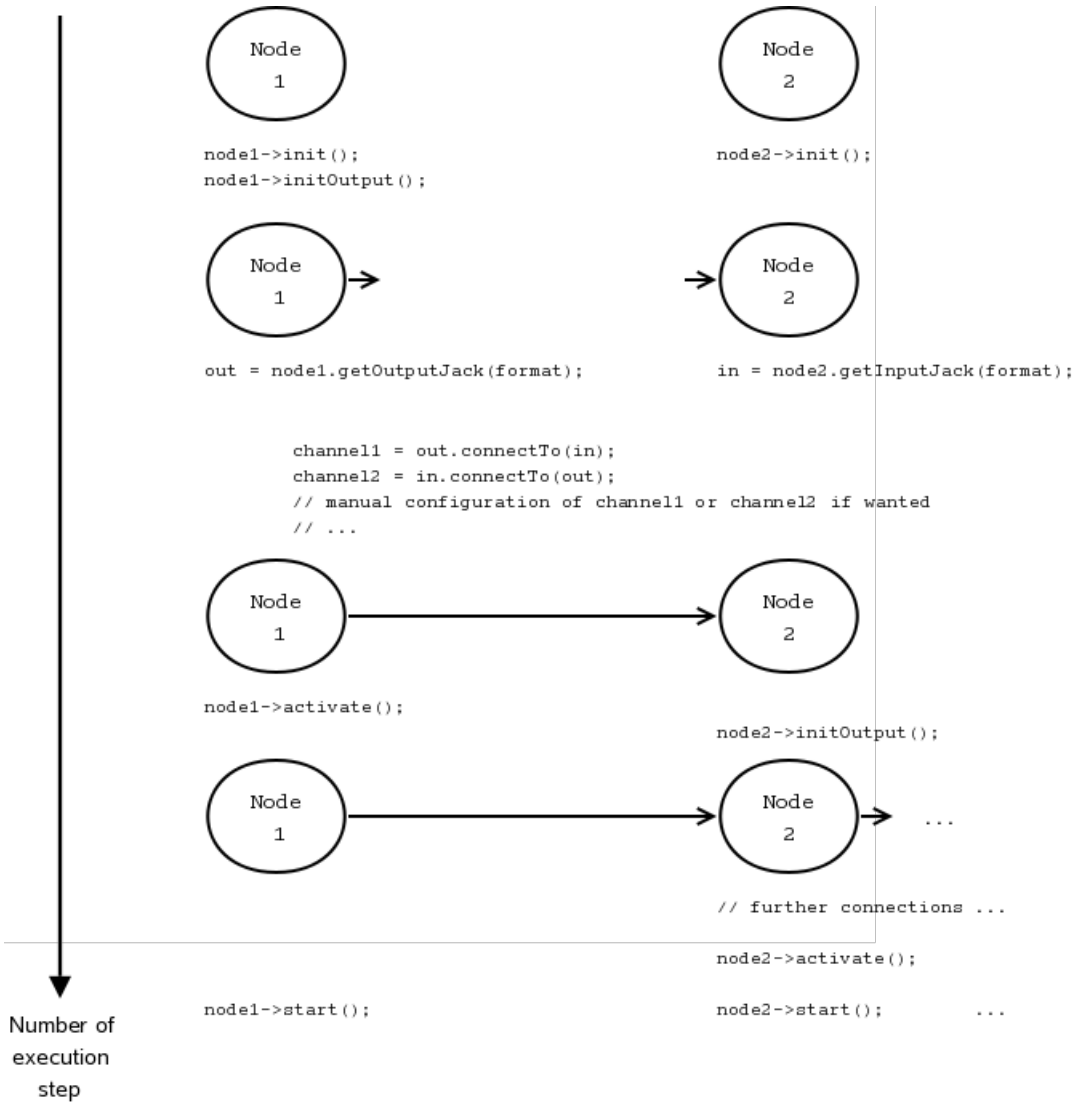

Figure Figure 2 shows an example for the manual connection setup between two nodes. The state transitions methods of NMM nodes are called during the manual connection setup between several nodes. Starting from a source node (Node 1), a converter/filter node is connected (Node 2). These steps are repeated for all other connections within the flow graph. Both calls of `connectTo()` return a communication channel that refers to the same connection and can be configured manually or automatically. Alternatively, the `connect()` method can be called.

In the following, the source code of this sequence using the `connect()` method is shown.

```
node1->init();
node2->init();

node1->initOutput();

// node1 is in state OUTPUT_INITIALIZED
// node2 is in state INITIALIZED
connect(node1, node2);

node1->activate();
node2->initOutput();
node2->activate();

node1->start();
node2->start();
```

4.2. Disconnecting Nodes

For disconnecting two nodes, they need to be in the corresponding states, i.e. in the same states as for connecting them. In the following, the source code of this sequence using the `disconnect_output()` method is shown.

```
node1->stop();
node2->stop();

// flush both nodes to release data stored internally
node1->flush();
node2->flush();

node1->deactivate();
node2->deactivate();

node2->deinitOutput();

// node1 is in state OUTPUT_INITIALIZED
// node2 is in state INITIALIZED
disconnect_output(node1);
```

```
node1->deinitOutput();

node1->deinit();
node2->deinit();
```