Managed nodes

This article describes the concept of a node with a managed life cycle. It aims to document some of the options for supporting manage d-life cycle nodes in ROS 2. It has been written with consideration for the existing design of the ROS 2 C++ client library, and in particular the current design of executors.

Authors: Geoffrey Biggs (https://github.com/gbiggs) Tully Foote (https://github.com/tfoote)

Date Written: 2015-06

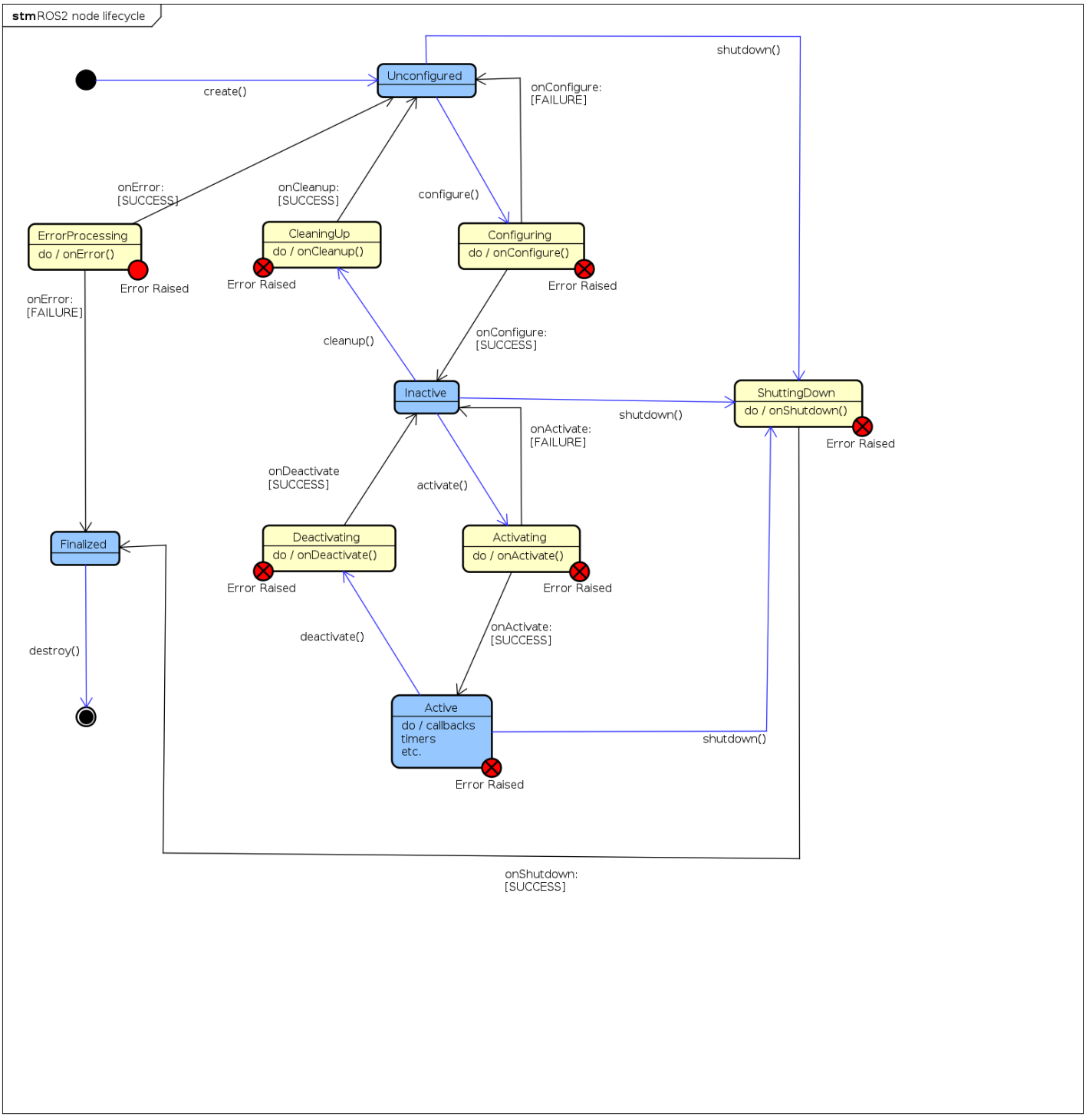
Last Modified: 2021-02

Background

A managed life cycle for nodes allows greater control over the state of ROS system. It will allow roslaunch to ensure that all components have been instantiated correctly before it allows any component to begin executing its behaviour. It will also allow nodes to be restarted or replaced on-line.

The most important concept of this document is that a managed node presents a known interface, executes according to a known life cycle state machine, and otherwise can be considered a black box. This allows freedom to the node developer on how they provide the managed life cycle functionality, while also ensuring that any tools created for managing nodes can work with any compliant node.

Life cycle



There are 4 primary states:

- `Unconfigured`
- `Inactive`
- `Active`

- `Finalized`

To transition out of a primary state requires action from an external supervisory process, with the exception of an error being triggered in the `Active` state.

There are also 6 transition states which are intermediate states during a requested transition.

- `Configuring`
- `CleaningUp`
- `ShuttingDown`
- `Activating`
- `Deactivating`
- `ErrorProcessing`

In the transitions states logic will be executed to determine if the transition is successful. Success or failure shall be communicated to lifecycle management software through the lifecycle management interface.

There are 7 transitions exposed to a supervisory process, they are:

- `create`
- `configure`
- `cleanup`
- `activate`
- `deactivate`
- `shutdown`
- `destroy`

The behavior of each state is as defined below.

Primary State: Unconfigured

This is the life cycle state the node is in immediately after being instantiated. This is also the state in which a node may be retuned to after an error has happened. In this state there is expected to be no stored state.

Valid transition out

- The node may transition to the `Inactive` state via the `configure` transition.
- The node may transition to the `Finalized` state via the `shutdown` transition.

Primary State: Inactive

This state represents a node that is not currently performing any processing.

The main purpose of this state is to allow a node to be (re-)configured (changing configuration parameters, adding and removing topic publications/subscriptions, etc) without altering its behavior while it is running.

While in this state, the node will not receive any execution time to read topics, perform processing of data, respond to functional service requests, etc.

In the inactive state, any data that arrives on managed topics will not be read and or processed. Data retention will be subject to the configured QoS policy for the topic.

Any managed service requests to a node in the inactive state will not be answered (to the caller, they will fail immediately).

Valid transitions out of Inactive

- A node may transition to the `Finalized` state via the `shutdown` transition.
- A node may transition to the `Unconfigured` state via the `cleanup` transition.
- A node may transition to the `Active` state via the `activate` transition.

Primary State: Active

This is the main state of the node's life cycle. While in this state, the node performs any processing, responds to service requests, reads and processes data, produces output, etc.

If an error that cannot be handled by the node/system occurs in this state, the node will transition to `ErrorProcessing`.

Valid transitions out of Active

- A node may transition to the `Inactive` state via the `deactivate` transition.
- A node may transition to the `Finalized` state via the `shutdown` transition.

Primary State: Finalized

The `Finalized` state is the state in which the node ends immediately before being destroyed. This state is always terminal the only transition from here is to be destroyed.

This state exists to support debugging and introspection. A node which has failed will remain visible to system introspection and may be potentially introspectable by debugging tools instead of directly destructing. If a node is being launched in a respawn loop or has known reasons for cycling it is expected that the supervisory process will have a policy to automatically destroy and recreate the node.

Valid transitions out of Finalized

- A node may be deallocated via the `destroy` transition.

Transition State: Configuring

In this transition state the node's `onConfigure` callback will be called to allow the node to load its configuration and conduct any required setup.

The configuration of a node will typically involve those tasks that must be performed once during the node's life time, such as obtaining permanent memory buffers and setting up topic publications/subscriptions that do not change.

The node uses this to set up any resources it must hold throughout its life (irrespective of if it is active or inactive). As examples, such resources may include topic publications and subscriptions, memory that is held continuously, and initialising configuration parameters.

Valid transitions out of Configuring

- If the `onConfigure` callback succeeds the node will transition to `Inactive`
- If the `onConfigure` callback results in a failure code (TODO specific code) the node will transition back to `Unconfigured`.
- If the `onConfigure` callback raises or results in any other result code the node will transition to `ErrorProcessing`

Transition State: CleaningUp

In this transition state the node's callback `onCleanup` will be called. This method is expected to clear all state and return the node to a functionally equivalent state as when first created. If the cleanup cannot be successfully achieved it will transition to `ErrorProcessing`.

Valid transitions out if CleaningUp

- If the `onCleanup` callback succeeds the node will transition to `Unconfigured`.
- If the `onCleanup` callback raises or results in any other return code the node will transition to `ErrorProcessing`.

Transition State: Activating

In this transition state the callback `onActivate` will be executed. This method is expected to do any final preparations to start executing. This may include acquiring resources that are only held while the node is actually active, such as access to hardware. Ideally, no preparation that requires significant time (such as lengthy hardware initialisation) should be performed in this callback.

Valid transitions out if Activating

- If the `onActivate` callback succeeds the node will transition to `Active`.
- If the `onActivate` callback raises or results in any other return code the node will transition to `ErrorProcessing`.

Transition State: Deactivating

In this transition state the callback `onDeactivate` will be executed. This method is expected to do any cleanup to start executing, and should reverse the `onActivate` changes.

Valid transitions out of Deactivating

- If the `onDeactivate` callback succeeds the node will transition to `Inactive`.
- If the `onDeactivate` callback raises or results in any other return code the node will transition to `ErrorProcessing`.

Transition State: ShuttingDown

In this transition state the callback `onShutdown` will be executed. This method is expected to do any cleanup necessary before destruction. It may be entered from any Primary State except `Finalized`, the originating state will be passed to the method.

Valid transitions out of ShuttingDown

- If the `onShutdown` callback succeeds the node will transition to `Finalized`.
- If the `onShutdown` callback raises or results in any other return code the node will transition to `ErrorProcessing`.

Transition State: ErrorProcessing

This transition state is where any error can be cleaned up. It is possible to enter this state from any state where user code will be executed. If error handling is successfully completed the node can return to `Unconfigured`, If a full cleanup is not possible it must fail and the node will transition to `Finalized` in preparation for destruction.

Transitions to `ErrorProcessing` may be caused by error return codes in callbacks as well as methods within a callback or an uncaught exception.

Valid transitions out of ErrorProcessing

- If the `onError` callback succeeds the node will transition to `Unconfigured`. It is expected that the `onError` will clean up all state from any previous state. As such if entered from `Active` it must provide the cleanup of both `onDeactivate` and `onCleanup` to return success.
- If the `onShutdown` callback raises or results in any other result code the node will transition to `Finalized`.

Destroy Transition

This transition will simply cause the deallocation of the node. In an object oriented environment it may just involve invoking the destructor. Otherwise it will invoke a standard deallocation method. This transition should always succeed.

Create Transition

This transition will instantiate the node, but will not run any code beyond the constructor.

Management Interface

A managed node will be exposed to the ROS ecosystem by the following interface, as seen by tools that perform the managing. This interface should not be subject to the restrictions on communications imposed by the lifecycle states.

It is expected that a common pattern will be to have a container class which loads a managed node implementation from a library and through a plugin architecture automatically exposes the required management interface via methods and the container is not subject to the lifecycle management. However, it is fully valid to consider any

implementation which provides this interface and follows the lifecycle policies a managed node. Conversely, any object that provides these services but does not behave in the way defined in the life cycle state machine is malformed.

These services may also be provided via attributes and method calls (for local management) in addition to being exposed ROS messages and topics/services (for remote management). In the case of providing a ROS middleware interface, specific topics must be used, and they should be placed in a suitable namespace.

Each possible supervisory transition will be provided as a service by the name of the transition except `create`. `create` will require an extra argument for finding the node to instantiate. The service will report whether the transition was successfully completed.

Lifecycle events

A topic should be provided to broadcast the new life cycle state when it changes. This topic must be latched. The topic must be named `lifecycle_state` it will carry both the end state and the transition, with result code. It will publish every time that a transition is triggered, whether successful or not.

Node Management

There are several different ways in which a managed node may transition between states. Most state transitions are expected to be coordinated by an external management tool which will provide the node with it's configuration and start it. The external management tool is also expected monitor it and execute recovery behaviors in case of failures. A local management tool is also a possibility, leveraging method level interfaces. And a node could be configured to self manage, however this is discouraged as this will interfere with external logic trying to managed the node via the interface.

There is one transition expected to originate locally, which is the `ERROR` transition.

A managed node may also want to expose arguments to automatically configure and activate when run in an unmanaged system.

Extensions

This lifecycle will be required to be supported throughout the toolchain as such this design is not intended to be extended with additional states. It is expected that there will be more complicated application specific state machines. They may exist inside of any lifecycle state or at the macro level these lifecycle states are expected to be useful primitives as part of a supervisory system.

View Source (https://github.com/ros2/design/blob/gh-pages/articles/node_lifecycle.md)

Edit in Github (https://github.com/ros2/design/edit/gh-pages/articles/node_lifecycle.md)

Pull Requests

Open

Closed

Please Login to View Pull Requests (https://github.com/login/oauth/authorize?client_id=efa6dd8eae9106381720)

Contributors

Please Login to View Contributors (https://github.com/login/oauth/authorize?client_id=efa6dd8eae9106381720)
