***roscpp overview (/roscpp/Overview)**: Initialization and Shutdown (/roscpp/Overview/Initialization%20and%20Shutdown) | Basics (/roscpp/Overview/Messages) | Advanced: Traits [ROS C Turtle] (/roscpp/Overview/MessagesTraits) | Advanced: Custom Allocators [ROS C Turtle] (/roscpp/Overview/MessagesCustomAllocators) | Advanced: Serialization and Adapting Types [ROS C Turtle] (/roscpp/Overview/MessagesSerializationAndAdaptingTypes) | Publishers and Subscribers (/roscpp/Overview/Publishers%20and%20Subscribers) | Services (/roscpp/Overview/Services) | Parameter Server (/roscpp/Overview/Parameter%20Server) | Timers (Periodic Callbacks) (/roscpp/Overview/Timers) | NodeHandles (/roscpp/Overview/NodeHandles) | Callbacks and Spinning | Logging (/roscpp/Overview/Logging) | Names and Node Information (/roscpp/Overview/Names%20and%20Node%20Information) | Time (/roscpp/Overview/Time) | Exceptions (/roscpp/Overview/Exceptions) | Compilation Options (/roscpp/Overview/Compilation%20Options) | Advanced: Internals (/roscpp/Overview/Internals) | tf/Overview (/tf/Overview) | tf/Tutorials (/tf/Tutorials) | C++ Style Guide (/CppStyleGuide)*

Contents

roscpp (/roscpp) does not try to specify a threading model for your application. This means that while roscpp (/roscpp) may use threads behind the scenes to do network management, scheduling etc., it will never expose its threads to your application. roscpp (/roscpp) does, however, allow your callbacks to be called from any number of threads if that's what you want.

The end result is that without a little bit of work from the user your subscription, service and other callbacks will **never** be called. The most common solution is `ros::spin()`, but you must use one of the options below.

Note: Callback queues/spinning do not have any effect on the internal network communication in roscpp. They only affect when user callbacks occur. They *will* have an effect on the subscription queue, since how fast you process your callbacks and how quickly messages are arriving determines whether or not messages will be dropped.

1. Single-threaded Spinning

The simplest (and most common) version of single-threaded spinning is `ros::spin()`:

Toggle line numbers

```
1 ros::init(argc, argv, "my_node");
2 ros::NodeHandle nh;
3 ros::Subscriber sub = nh.subscribe(...);
4 ...
5 ros::spin();
```

In this application all user callbacks will be called from within the `ros::spin()` call. `ros::spin()` will not return until the node has been shutdown, either through a call to `ros::shutdown()` or a `Ctrl-C`.

Another common pattern is to call `ros::spinOnce()` periodically:

Toggle line numbers

```
1 ros::Rate r(10); // 10 hz
2 while (should_continue)
3 {
4   ... do some work, publish some messages, etc. ...
5   ros::spinOnce();
6   r.sleep();
7 }
```

`ros::spinOnce()` will call all the callbacks waiting to be called at that point in time.

Implementing a `spin()` of our own is quite simple:

Toggle line numbers

```
1 #include <ros/callback_queue.h>
2 ros::NodeHandle n;
3 while (ros::ok())
4 {
5   ros::getGlobalCallbackQueue()->callAvailable(ros::WallDuration(0.1));
6 }
```

and `spinOnce()` is simply:

Toggle line numbers

```
1 #include <ros/callback_queue.h>
2
3 ros::getGlobalCallbackQueue()->callAvailable(ros::WallDuration(0));
```

Note: `spin()` and `spinOnce()` are really meant for single-threaded applications, and are not optimized for being called from multiple threads at once. See the multi-threaded spinning section for information on spinning from multiple threads.

2. Multi-threaded Spinning

roscpp (/roscpp) provides some built-in support for calling callbacks from multiple threads. There are two built-in options for this:

`ros::MultiThreadedSpinner`

> `MultiThreadedSpinner` is a blocking spinner, similar to `ros::spin()`. You can specify a number of threads in its constructor, but if unspecified (or set to 0), it will use a thread for each CPU core.
>
> Toggle line numbers

```
1 ros::MultiThreadedSpinner spinner(4); // Use 4 threads
2 spinner.spin(); // spin() will not return until the node has been sh
utdown
3
```

`ros::AsyncSpinner` *(since 0.10)*

AsyncSpinner API (Jade) (http://docs.ros.org/api/roscpp/html/classros_1_1AsyncSpinner.html)

A more useful threaded spinner is the `AsyncSpinner`. Instead of a blocking `spin()` call, it has `start()` and `stop()` calls, and will automatically stop when it is destroyed. An equivalent use of `AsyncSpinner` to the `MultiThreadedSpinner` example above, is:

Toggle line numbers

```
1 ros::AsyncSpinner spinner(4); // Use 4 threads
2 spinner.start();
3 ros::waitForShutdown();
```

Please note that the `ros::waitForShutdown()` function does **not** spin on its own, so the example above will spin with 4 threads in total.

3. CallbackQueue::callAvailable() and callOne()

See also: CallbackQueue API docs (http://www.ros.org/doc/api/roscpp/html/classros_1_1CallbackQueue.html)

You can create callback queues this way:

```
#include <ros/callback_queue.h>
...
ros::CallbackQueue my_queue;
```

The `CallbackQueue` class has two ways of invoking the callbacks inside it: `callAvailable()` and `callOne()`. `callAvailable()` will take everything currently in the queue and invoke all of them. `callOne()` will simply invoke the oldest callback on the queue.

Both `callAvailable()` and `callOne()` can take in an optional timeout, which is the amount of time they will wait for a callback to become available before returning. If this is zero and there are no callbacks in the queue the method will return immediately.

Through ROS 0.10 the default timeout has been 0.1 seconds. ROS 0.11 makes the default 0.

4. Advanced: Using Different Callback Queues

You may have noticed the call to `ros::getGlobalCallbackQueue()` in the above implementation of `spin()`. By default, all callbacks get assigned into that global queue, which is then processed by `ros::spin()` or one of the alternatives. roscpp (/roscpp) also lets you assign custom callback queues and service them separately. This can be done in one of two granularities:

1. Per `subscribe()`, `advertise()`, `advertiseService()`, etc.
2. Per `NodeHandle`

(1) is possible using the advanced versions of those calls that take a `*Options` structure. See the API docs (http://www.ros.org/doc/api/roscpp/html/classros_1_1NodeHandle.html) for those calls for more information.

(2) is the more common way:

Toggle line numbers

```
1 ros::NodeHandle nh;
2 nh.setCallbackQueue(&my_callback_queue);
```

This makes all subscription, service, timer, etc. callbacks go through `my_callback_queue` instead of roscpp (/roscpp)'s default queue. This means `ros::spin()` and `ros::spinOnce()` will **not** call these callbacks. Instead, you must service that queue separately. You can do so manually using the `ros::CallbackQueue::callAvailable()` and `ros::CallbackQueue::callOne()` methods:

Toggle line numbers

```
1 my_callback_queue.callAvailable(ros::WallDuration());
2 // alternatively, .callOne(ros::WallDuration()) to only call a single cal
lback instead of all available
3
```

The various `*Spinner` objects can also take a pointer to a callback queue to use rather than the default one:

Toggle line numbers

```
1 ros::AsyncSpinner spinner(0, &my_callback_queue);
2 spinner.start();
```

or

Toggle line numbers

```
1 ros::MultiThreadedSpinner spinner(0);
2 spinner.spin(&my_callback_queue);
```

4.1 Uses

Separating out callbacks into different queues can be useful for a number of reasons. Some examples include:

1. Long-running services. Assigning a service its own callback queue that gets serviced in a separate thread means that service is guaranteed not to block other callbacks.
2. Threading specific computationally expensive callbacks. Similar to the long-running service case, this allows you to thread specific callbacks while keeping the simplicity of single-threaded callbacks for the rest your application.

Wiki: roscpp/Overview/Callbacks and Spinning (last edited 2017-07-29 18:28:14 by MichaelGoerner (/MichaelGoerner))

Brought to you by:  Open Source Robotics Foundation

(http://www.osrfoundation.org)