rosonic

None

None

None

Table of contents

1. Motivation

Following are some problems outlined with ROS and `rospy` that motivates the use of `rosonic`. However, no `rosonic` code will be found here.

1.1 ROS Nodes

When creating a node in ROS, it will typically look something like any of the examples below.

```
#! /usr/bin/env python import rospy ## Publisher pattern ## def main(): # initialization pub = rospy.Publisher(...) rate = rospy.Rate(...) # main loop whi
```

Neither of these are very complicated. However, this is mainly since they follow a clear design pattern. When developing, especially in research- oriented projects,following design patterns are often second priority and complexity tend to get out of hand. This is especially true when you start using other ROS concepts more and more.

For new users there is also the overhead of learning what a ROS node is, how it behaves and how they relate to other ROS concepts. It may be easy to explain that ROS starts a separate process that runs your python script as a standalone program, the script declares itself to be ROS node with `rospy.init_node`, opens communication channels for any of your topics, and then it executes your logic. However, that is probably unclear for any developers/researcher that gets thrown into ROS for the first time.

As a minor example, how should you design a subscribe-publish node? To clarify, you node should subscribe to some topic, augment the incoming data then, on a new topic, publish it. Should you initialize in `main`? How does `callback` then have your `pub`? Maybe you declare `pub` globally, but that's a Python anti-pattern! Of course you can solve this neatly and in many ways, but there is nothing in `rospy` that motivates you to follow best-practice and/or a good design patter. Below is one solution where `pub` is sent as an argument to `callback` but you can imagine how that can get out of hand with more publisher and subscribers. The initialization is moved to a dedicated function and, like previously, the node ends on `rospy.spin()`.

```
#! /usr/bin/env python import rospy def callback(msg, pub): pub.publish(...) def init(): pub = rospy.Publisher(...) rospy.Subscriber(..., callback, pub) i
```

1.2 ROS Parameters

Enough about design patterns! I don't care about writing nice code, why should I care about `rosonic`?

Now, let me tell you! Surely the most important problems to solve are those that cause mild inconvenience. `rosonic` will help you once again before you can say "ROS parameters"! Let's look at a typical scenario...

```
#! /usr/bin/env python import rospy use_filter = True def main(): global use_filter # initialize has_image2 = rospy.has_param('~image2') assert has_image2
```

There are multiple problems with passing topic names through parameters, a very typical use case. Firstly, the distinction between `'~image1'` and `'image1'` is not very clear (a problem not solved by `rosonic`). Many times `'~image1'` may have the default value `'image1'` as well! Secondly, asserting that required parameters, such as `'~image2'`, exists is clunky. Thirdly, parameter values must be passed to different functions/scopes or declared global (same problem as `pub` before).