roslibpy

Release 1.5.0

Gramazio Kohler Research

Mar 29, 2023

CONTENTS

Python ROS Bridge library allows to use Python and IronPython to interact with [ROS](), the open-source robotic middleware. It uses WebSockets to connect to [rosbridge 2.0]() and provides publishing, subscribing, service calls, actionlib, TF, and other essential ROS functionality.

Unlike the [rospy]() library, this does not require a local ROS environment, allowing usage from platforms other than Linux.

The API of **roslibpy** is modeled to closely match that of [roslibjs]().

CONTENTS

1.1 roslibpy: ROS Bridge library

Python ROS Bridge library allows to use Python and IronPython to interact with ROS, the open-source robotic middleware. It uses WebSockets to connect to rosbridge 2.0 and provides publishing, subscribing, service calls, actionlib, TF, and other essential ROS functionality.

Unlike the rospy library, this does not require a local ROS environment, allowing usage from platforms other than Linux.

The API of **roslibpy** is modeled to closely match that of roslibjs.

1.1.1 Main features

- Topic publishing and subscribing.

- Service calls (client).

- Service advertisement (server).

- ROS parameter management (get/set/delete).

- ROS API services for getting ROS meta-information.

- Actionlib support for interfacing with preemptable tasks.

- TF Client via the `tf2_web_republisher`.

Roslibpy runs on Python 3.x and IronPython 2.7.

1.1.2 Installation

To install **roslibpy**, simply use `pip`:

```
pip install roslibpy
```

For IronPython, the `pip` command is slightly different:

```
ipy -X:Frames -m pip install --user roslibpy
```

Remember that you will need a working ROS setup including the **rosbridge server** and **TF2 web republisher** accessible within your network.

1.1.3 Documentation

The full documentation, including examples and API reference is available on readthedocs.

1.1.4 Contributing

Make sure you setup your local development environment correctly:

- Clone the roslibpy repository.
- Create a virtual environment.
- Install development dependencies:

```
pip install -r requirements-dev.txt
```

You're ready to start coding!

During development, use pyinvoke tasks on the command prompt to ease recurring operations:

- `invoke clean`: Clean all generated artifacts.
- `invoke check`: Run various code and documentation style checks.
- `invoke docs`: Generate documentation.
- `invoke test`: Run all tests and checks in one swift command.
- `invoke`: Show available tasks.

For more details, check the *Contributor's Guide* available as part of the documentation.

The default branch was recently renamed to *main*. If you've already cloned this repository, you'll need to update your local repository structure with the following lines:

```
git branch -m master main
git fetch origin
git branch -u origin/main main
```

1.1.5 Releasing this project

Ready to release a new version **roslibpy**? Here's how to do it:

- We use semver, i.e. we bump versions as follows:

    - `patch`: bugfixes.

    - `minor`: backwards-compatible features added.

    - `major`: backwards-incompatible changes.

- Update the `CHANGELOG.rst` with all novelty!

- Ready? Release everything in one command:

```
invoke release [patch|minor|major]
```

- Profit!

1.1.6 Credits

This library is based on roslibjs and to a large extent, it is a line-by-line port to Python, changing only where a more idiomatic form makes sense, so a huge part of the credit goes to the roslibjs authors.

1.2 Examples

Getting started with **roslibpy** is simple. The following examples will help you on the first steps using it to connect to a ROS environment. Before you start, make sure you have ROS and *rosbridge* running (see *ROS Setup*).

These examples assume ROS is running on the same computer where you run the examples. If that is not the case, change the `host` argument from `'localhost'` to the *IP Address* of your ROS instance.

1.2.1 First connection

We start importing `roslibpy` as follows:

```
>>> import roslibpy
```

And we initialize the connection with:

```
>>> ros = roslibpy.Ros(host='localhost', port=9090)
>>> ros.run()
```

Easy, right? Let's check the status:

```
>>> ros.is_connected
True
```

Yay! Our first connection to ROS!

1.2.2 Putting it all together

Let's build a full example into a python file. Create a file named `ros-hello-world.py` and paste the following content:

```python
import roslibpy

client = roslibpy.Ros(host='localhost', port=9090)
client.run()
print('Is ROS connected?', client.is_connected)
client.terminate()
```

Now run it from the command prompt typing:

```
$ python ros-hello-world.py
```

The program will run, print once we are connected and terminate the connection.

Controlling the event loop

In the previous examples, we started the ROS connection with a call to `run()`, which starts the event loop in the background. In some cases, we want to handle the main event loop more explicitly in the foreground. *roslibpy. Ros* provides the method `run_forever()` for this purpose.

If we use this method to start the event loop, we need to setup all connection handlers beforehand. We will use the *roslibpy.Ros.on_ready()* method to do this. We will pass a function to it, that will be invoked when the connection is ready.

The following snippet shows the same connection example above but using `run_forever()` and `on_ready`:

```python
from __future__ import print_function
import roslibpy

client = roslibpy.Ros(host='localhost', port=9090)
client.on_ready(lambda: print('Is ROS connected?', client.is_connected))
client.run_forever()
```

Note: The difference between `run()` and `run_forever()` is that the former starts the event processing in a separate thread, while the latter blocks the calling thread.

1.2.3 Disconnecting

Once your task is done, you should disconnect cleanly from `rosbridge`. There are two related methods available for this:

- *roslibpy.Ros.close()*: Disconnect the websocket connection. Once the connection is closed, it is still possible to reconnect by calling *roslibpy.Ros.connect()*: again.

- *roslibpy.Ros.terminate()*: Terminate the main event loop. If the connection is still open, it will first close it.

Note: Terminating the event loop is an irreversible action when using the `twisted/authbahn` loop because `twisted` reactors cannot be restarted. This operation should be reserved to be executed at the very end of your

program.

1.2.4 Reconnecting

If `rosbridge` is not responsive when the connection is started or if an established connection drops uncleanly, `roslibpy` will try to reconnect automatically, and reconnect subscriber and publisher topics as well. Reconnect will be retried with an exponential back-off.

1.2.5 Hello World: Topics

The `Hello world` of ROS is to start two nodes that communicate using topic subscription/publishing. The nodes (a talker and a listener) are extremely simple but they exemplify a distributed system with communication between two processes over the ROS infrastructure.

Writing the talker node

The following example starts a ROS node and begins to publish messages in loop (to terminate, press `ctrl+c`):

```python
import time

import roslibpy

client = roslibpy.Ros(host='localhost', port=9090)
client.run()

talker = roslibpy.Topic(client, '/chatter', 'std_msgs/String')

while client.is_connected:
    talker.publish(roslibpy.Message({'data': 'Hello World!'}))
    print('Sending message...')
    time.sleep(1)

talker.unadvertise()

client.terminate()
```

- `ros-hello-world-talker.py`

Writing the listener node

Now let's move on to the listener side:

```python
from __future__ import print_function
import roslibpy

client = roslibpy.Ros(host='localhost', port=9090)
client.run()

listener = roslibpy.Topic(client, '/chatter', 'std_msgs/String')
listener.subscribe(lambda message: print('Heard talking: ' + message['data']))

try:
```

```
    while True:
        pass
except KeyboardInterrupt:
    client.terminate()
```

- `ros-hello-world-listener.py`

Running the example

Open a command prompt and start the talker:

```
python ros-hello-world-talker.py
```

Now open a second command prompt and start the listener:

```
python ros-hello-world-listener.py
```

Note: It is not relevant where the files are located. They can be in different folders or even in different computers as long as the ROS instance is the same.

1.2.6 Using services

Another way for nodes to communicate between each other is through ROS Services.

Services require the definition of request and response types so the following example shows how to use an existing service called `get_loggers`:

```python
import roslibpy

client = roslibpy.Ros(host='localhost', port=9090)
client.run()

service = roslibpy.Service(client, '/rosout/get_loggers', 'roscpp/GetLoggers')
request = roslibpy.ServiceRequest()

print('Calling service...')
result = service.call(request)
print('Service response: {}'.format(result['loggers']))

client.terminate()
```

- `ros-service-call-logger.py`

1.2.7 Creating services

It is also possible to create new services, as long as the service type definition is present in your ROS environment.

The following example shows how to create a simple service that uses one of the standard service types defined in ROS (`std_srvs/SetBool`):

```python
import roslibpy


def handler(request, response):
    print('Setting speed to {}'.format(request['data']))
    response['success'] = True
    return True


client = roslibpy.Ros(host='localhost', port=9090)

service = roslibpy.Service(client, '/set_ludicrous_speed', 'std_srvs/SetBool')
service.advertise(handler)
print('Service advertised.')

client.run_forever()
client.terminate()
```

- `ros-service.py`

Download it and run it from the command prompt typing:

```
$ python ros-service.py
```

The service will be active while the program is running (to terminate, press `ctrl+c`).

Leave this service running and download and run the following service calling code example to verify the service is working:

- `ros-service-call-set-bool.py`

Download it and run it from the command prompt typing:

```
$ python ros-service-call-set-bool.py
```

Note: Now that you have a grasp of the basics of `roslibpy`, check out more details in the *API Reference*.

1.2.8 Actions

Besides Topics and Services, ROS provides **Actions**, which are intended for long-running tasks, such as navigation, because they are non-blocking and allow the cancellation (preempting) of an action while it is executing.

`roslibpy` supports both consuming actions (i.e. action clients) and also providing actions, through the *roslibpy.actionlib.SimpleActionServer*.

The following examples use the **Fibonacci** action, which is defined in the actionlib_tutorials.

Action servers

Let's start with the definition of the fibonacci server:

```python
import roslibpy
import roslibpy.actionlib

client = roslibpy.Ros(host='localhost', port=9090)
server = roslibpy.actionlib.SimpleActionServer(client, '/fibonacci', 'actionlib_
↪tutorials/FibonacciAction')


def execute(goal):
    print('Received new fibonacci goal: {}'.format(goal['order']))

    seq = [0, 1]

    for i in range(1, goal['order']):
        if server.is_preempt_requested():
            server.set_preempted()
            return

        seq.append(seq[i] + seq[i - 1])
        server.send_feedback({'sequence': seq})

    server.set_succeeded({'sequence': seq})


server.start(execute)
client.run_forever()
```

- ros-action-server.py

Download it and run it from the command prompt typing:

```
$ python ros-action-server.py
```

The action server will be active while the program is running (to terminate, press `ctrl+c`).

Leave this window running if you want to test it with the next example.

Action clients

Now let's see how to write an action client for our newly created server.

The following program shows a simple action client:

```python
from __future__ import print_function
import roslibpy
import roslibpy.actionlib

client = roslibpy.Ros(host='localhost', port=9090)
client.run()

action_client = roslibpy.actionlib.ActionClient(client,
                                                '/fibonacci',
                                                'actionlib_tutorials/FibonacciAction')

goal = roslibpy.actionlib.Goal(action_client,
```

(continues on next page)

```
                                    roslibpy.Message({'order': 8}))

goal.on('feedback', lambda f: print(f['sequence']))
goal.send()
result = goal.wait(10)
action_client.dispose()

print('Result: {}'.format(result['sequence']))
```

- `ros-action-client.py`

Download it and run it from the command prompt typing:

```
$ python ros-action-client.py
```

You will immediately see all the intermediate calculations of our action server, followed by a line indicating the resulting fibonacci sequence.

This example is very simplified and uses the `roslibpy.actionlib.Goal.wait()` function to make the code easier to read as an example. A more robust way to handle results is to hook up to the `result` event with a callback.

1.2.9 Querying ROS API

ROS provides an API to inspect topics, services, nodes and much more. This API can be used programmatically from Python code, and also be invoked from the command line.

Usage from the command-line

The command line mimics closely that of ROS itself.

The following commands are available:

```
$ roslibpy topic list
$ roslibpy topic type /rosout
$ roslibpy topic find std_msgs/Int32
$ roslibpy msg info rosgraph_msgs/Log

$ roslibpy service list
$ roslibpy service type /rosout/get_loggers
$ roslibpy service find roscpp/GetLoggers
$ roslibpy srv info roscpp/GetLoggers

$ roslibpy param list
$ roslibpy param set /foo "[\"1\", 1, 1.0]"
$ roslibpy param get /foo
$ roslibpy param delete /foo
```

Usage from Python code

And conversely, the following methods allow to query the ROS API from Python code.

Topics

- *roslibpy.Ros.get_topics()*
- *roslibpy.Ros.get_topic_type()*
- *roslibpy.Ros.get_topics_for_type()*
- *roslibpy.Ros.get_message_details()*

Services

- *roslibpy.Ros.get_services()*
- *roslibpy.Ros.get_service_type()*
- *roslibpy.Ros.get_services_for_type()*
- *roslibpy.Ros.get_service_request_details()*
- *roslibpy.Ros.get_service_response_details()*

Params

- *roslibpy.Ros.get_params()*
- *roslibpy.Ros.get_param()*
- *roslibpy.Ros.set_param()*
- *roslibpy.Ros.delete_param()*

Time

- *roslibpy.Ros.get_time()*

1.2.10 Advanced examples

The following list is a compilation of more elaborate examples of the usage of `roslibpy`.

We encourage everyone to submit suggestions for new examples, either send a pull request or request it via the issue tracker.

Enable debug logging

This example shows how to enable debugging output using Python `logging` infrastructure.

```python
import logging

import roslibpy

# Configure logging to high verbosity (DEBUG)
fmt = '%(asctime)s %(levelname)8s: %(message)s'
logging.basicConfig(format=fmt, level=logging.DEBUG)
log = logging.getLogger(__name__)

client = roslibpy.Ros(host='127.0.0.1', port=9090)
client.on_ready(lambda: log.info('On ready has been triggered'))

client.run_forever()
```

Check roundtrip message latency

This example shows how to check roundtrip message latency on your system.

```python
import logging
import time

import roslibpy

# Configure logging
fmt = '%(asctime)s %(levelname)8s: %(message)s'
logging.basicConfig(format=fmt, level=logging.INFO)
log = logging.getLogger(__name__)

client = roslibpy.Ros(host='127.0.0.1', port=9090)

def receive_message(msg):
    age = int(time.time() * 1000) - msg['data']
    log.info('Age of message: %6dms', age)

publisher = roslibpy.Topic(client, '/check_latency', 'std_msgs/UInt64')
publisher.advertise()

subscriber = roslibpy.Topic(client, '/check_latency', 'std_msgs/UInt64')
subscriber.subscribe(receive_message)

def publish_message():
    publisher.publish(dict(data=int(time.time() * 1000)))
    client.call_later(.5, publish_message)

client.on_ready(publish_message)
client.run_forever()
```

The output on the console should look similar to the following:

```
$ python 02_check_latency.py
2020-04-09 07:45:49,909     INFO: Connection to ROS ready.
2020-04-09 07:45:50,431     INFO: Age of message:      2ms
```

```
2020-04-09 07:45:50,932       INFO: Age of message:      2ms
2020-04-09 07:45:51,431       INFO: Age of message:      1ms
2020-04-09 07:45:51,932       INFO: Age of message:      2ms
2020-04-09 07:45:52,434       INFO: Age of message:      3ms
2020-04-09 07:45:52,934       INFO: Age of message:      2ms
2020-04-09 07:45:53,435       INFO: Age of message:      3ms
2020-04-09 07:45:53,934       INFO: Age of message:      1ms
2020-04-09 07:45:54,436       INFO: Age of message:      2ms
```

Throttle messages for a slow consumer

This example shows how to throttle messages that are published are a rate faster than what a slow consumer (subscribed) can process. In this example, only the newest messages are preserved, messages that cannot be consumed on time are dropped.

```python
import time
import logging

from roslibpy import Header
from roslibpy import Ros
from roslibpy import Time
from roslibpy import Topic
from roslibpy.core import LOGGER


# Configure logging
fmt = '%(asctime)s %(levelname)8s: %(message)s'
logging.basicConfig(format=fmt, level=logging.INFO)
log = logging.getLogger(__name__)

client = Ros(host='127.0.0.1', port=9090)


def receive_message(msg):
    header = Header(msg['seq'], msg['stamp'], msg['frame_id'])
    age = time.time() - header['stamp'].to_sec()
    fmt = 'Age of message (sequence #%d): %6.3f seconds'
    log.info(fmt, msg['seq'], age)
    # Simulate a very slow consumer
    time.sleep(.5)

publisher = Topic(client, '/slow_consumer', 'std_msgs/Header')
publisher.advertise()

# Queue length needs to be used in combination with throttle rate (in ms)
# This value must be tuned to the expected duration of the slow consumer
# and ideally bigger than the max of it,
# otherwise message will be older than expected (up to a limit)
subscriber = Topic(client, '/slow_consumer', 'std_msgs/Header',
                           queue_length=1, throttle_rate=600)
subscriber.subscribe(receive_message)


seq = 0
def publish_message():
    global seq
    seq += 1
    header = Header(frame_id='', seq=seq, stamp=Time.now())
```

```
    publisher.publish(header)
    client.call_later(.001, publish_message)

client.on_ready(publish_message)
client.run_forever()
```

In the console, you should see gaps in the sequence of messages, because the publisher is producing messages every 0.001 seconds, but we configure a queue of length 1, with a throttling of 600ms to give time to our slow consumer. Without this throttling, the consumer would process increasingly old messages.

Publish images

This example shows how to publish images using the built-in `sensor_msgs/CompressedImage` message type.

```python
import base64
import logging
import time

import roslibpy

# Configure logging
fmt = '%(asctime)s %(levelname)8s: %(message)s'
logging.basicConfig(format=fmt, level=logging.INFO)
log = logging.getLogger(__name__)

client = roslibpy.Ros(host='127.0.0.1', port=9090)

publisher = roslibpy.Topic(client, '/camera/image/compressed', 'sensor_msgs/
↪CompressedImage')
publisher.advertise()

def publish_image():
    with open('robots.jpg', 'rb') as image_file:
        image_bytes = image_file.read()
        encoded = base64.b64encode(image_bytes).decode('ascii')

    publisher.publish(dict(format='jpeg', data=encoded))

client.on_ready(publish_image)
client.run_forever()
```

Subscribe to images

This example shows how to subscribe to a topic of images using the built-in `sensor_msgs/CompressedImage` message type.

```python
import base64
import logging
import time

import roslibpy

# Configure logging
fmt = '%(asctime)s %(levelname)8s: %(message)s'
```

```python
logging.basicConfig(format=fmt, level=logging.INFO)
log = logging.getLogger(__name__)

client = roslibpy.Ros(host='127.0.0.1', port=9090)

def receive_image(msg):
    log.info('Received image seq=%d', msg['header']['seq'])
    base64_bytes = msg['data'].encode('ascii')
    image_bytes = base64.b64decode(base64_bytes)
    with open('received-image-{}.{}'.format(msg['header']['seq'], msg['format']) , 'wb
→') as image_file:
        image_file.write(image_bytes)

subscriber = roslibpy.Topic(client, '/camera/image/compressed', 'sensor_msgs/
→CompressedImage')
subscriber.subscribe(receive_image)

client.run_forever()
```

1.3 API Reference

This library relies on the ROS bridge suite by Robot Web Tools to interact with ROS via WebSockets.

The ROS bridge protocol uses JSON as message transport to allow access to ROS functionality such as publishing, subscribing, service calls, actionlib, TF, etc.

1.3.1 ROS Setup

In order to use this library, your ROS environment needs to be setup to run `rosbridge`.

First install the **rosbridge suite** with the following commands:

```
sudo apt-get install -y ros-kinetic-rosbridge-server
sudo apt-get install -y ros-kinetic-tf2-web-republisher
```

And before starting a connection, make sure you launch all services:

```
roslaunch rosbridge_server rosbridge_websocket.launch
rosrun tf2_web_republisher tf2_web_republisher
```

1.3.2 Connecting to ROS

The connection to ROS is managed by the *Ros* class. Besides connection and disposal, it handles automatic reconnections when needed.

Other classes that need an active connection with ROS receive this instance as an argument to their constructors.

class roslibpy.**Ros**(*host*, *port=None*, *is_secure=False*)
    Connection manager to ROS server.

> **Parameters**

> > • **host** (`str`) – Name or IP address of the ROS bridge host, e.g. `127.0.0.1`.

- **port** (`int`) – ROS bridge port, e.g. `9090`.

- **is_secure** (`bool`) – `True` to use a secure web sockets connection, otherwise `False`.

blocking_call_from_thread(*callback*, *timeout*)
  Call the given function from a thread, and wait for the result synchronously for as long as the timeout will allow.

  Parameters

  - **callback** – Callable function to be invoked from the thread.

  - **timeout** (`int`) – Number of seconds to wait for the response before raising an exception.

  Returns The results from the callback, or a timeout exception.

call_async_service(*message*, *callback*, *errback*)
  Send a service request to ROS once the connection is established.

  If a connection to ROS is already available, the request is sent immediately.

  Parameters

  - **message** ([*Message*](#)) – ROS Bridge Message containing the request.

  - **callback** – Callback invoked on successful execution.

  - **errback** – Callback invoked on error.

call_in_thread(*callback*)
  Call the given function in a thread.

  The threading implementation is deferred to the factory.

  Parameters **callback** (`callable`) – Callable function to be invoked.

call_later(*delay*, *callback*)
  Call the given function after a certain period of time has passed.

  Parameters

  - **delay** (`int`) – Number of seconds to wait before invoking the callback.

  - **callback** (`callable`) – Callable function to be invoked when ROS connection is ready.

call_sync_service(*message*, *timeout*)
  Send a blocking service request to ROS once the connection is established, waiting for the result to be return.

  If a connection to ROS is already available, the request is sent immediately.

  Parameters

  - **message** ([*Message*](#)) – ROS Bridge Message containing the request.

  - **timeout** (`int`) – Number of seconds to wait for the response before raising an exception.

  Returns Either returns the service request results or raises a timeout exception.

close(*timeout=10*)
  Disconnect from ROS.

connect()
  Connect to ROS.

delete_param(*name*, *callback=None*, *errback=None*)
  Delete parameter from the ROS Parameter Server.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

---

emit(*event_name*, **args*)
    Trigger a named event.

get_action_servers(*callback*, *errback=None*)
    Retrieve list of action servers in ROS.

get_message_details(*message_type*, *callback=None*, *errback=None*)
    Retrieve details of a message type in ROS.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

> **Returns** Message type details if blocking, otherwise `None`.

get_node_details(*node*, *callback=None*, *errback=None*)
    Retrieve list subscribed topics, publishing topics and services of a specific node name.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

---

get_nodes(*callback=None*, *errback=None*)
    Retrieve list of active node names in ROS.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

---

get_param(*name*, *callback=None*, *errback=None*)
    Get the value of a parameter from the ROS Parameter Server.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

> **Returns** Parameter value if blocking, otherwise `None`.

get_params(*callback=None*, *errback=None*)
    Retrieve list of param names from the ROS Parameter Server.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

> **Returns** *list* – List of parameters if blocking, otherwise `None`.

get_service_request_callback(*message*)
    Get the callback which, when called, sends the service request.

> **Parameters** **message** ([*Message*](#)) – ROS Bridge Message containing the request.

> **Returns** A callable which makes the service request.

---

get_service_request_details(*type*, *callback=None*, *errback=None*)
  Retrieve details of a ROS Service Request.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

>   **Returns** Service Request details if blocking, otherwise `None`.

get_service_response_details(*type*, *callback=None*, *errback=None*)
  Retrieve details of a ROS Service Response.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

>   **Returns** Service Response details if blocking, otherwise `None`.

get_service_type(*service_name*, *callback=None*, *errback=None*)
  Retrieve the type of a service in ROS.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

>   **Returns** *str* – Service type if blocking, otherwise `None`.

get_services(*callback=None*, *errback=None*)
  Retrieve list of active service names in ROS.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

>   **Returns** *list* – List of services if blocking, otherwise `None`.

get_services_for_type(*service_type*, *callback=None*, *errback=None*)
  Retrieve list of services in ROS matching the specified type.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

>   **Returns** *list* – List of services matching the specified type if blocking, otherwise `None`.

get_time(*callback=None*, *errback=None*)
  Retrieve the current ROS time.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

>   **Returns** *[Time](#)* – An instance of ROS Time.

get_topic_type(*topic*, *callback=None*, *errback=None*)
    Retrieve the type of a topic in ROS.

> **Note:** To make this a blocking call, pass `None` to the `callback` parameter .

> > **Returns** *str* – Topic type if blocking, otherwise `None`.

get_topics(*callback=None*, *errback=None*)
    Retrieve list of topics in ROS.

> **Note:** To make this a blocking call, pass `None` to the `callback` parameter .

> > **Returns** *list* – List of topics if blocking, otherwise `None`.

get_topics_for_type(*topic_type*, *callback=None*, *errback=None*)
    Retrieve list of topics in ROS matching the specified type.

> **Note:** To make this a blocking call, pass `None` to the `callback` parameter .

> > **Returns** *list* – List of topics matching the specified type if blocking, otherwise `None`.

property id_counter
    Generate an auto-incremental ID starting from 1.

> **Returns** *int* – An auto-incremented ID.

property is_connected
    Indicate if the ROS connection is open or not.

> **Returns** *bool* – True if connected to ROS, False otherwise.

off(*event_name*, *callback=None*)
    Remove a callback from an arbitrary named event.

> **Parameters**
>
> - **event_name** (`str`) – Name of the event from which to unsubscribe.
> - **callback** – Callable function. If `None`, all callbacks of the event will be removed.

on(*event_name*, *callback*)
    Add a callback to an arbitrary named event.

> **Parameters**
>
> - **event_name** (`str`) – Name of the event to which to subscribe.
> - **callback** – Callable function to be executed when the event is triggered.

on_ready(*callback*, *run_in_thread=True*)
    Add a callback to be executed when the connection is established.

    If a connection to ROS is already available, the callback is executed immediately.

> **Parameters**
>
> - **callback** – Callable function to be invoked when ROS connection is ready.

> • **run_in_thread** (`bool`) – True to run the callback in a separate thread, False otherwise.

run(*timeout=10*)

Kick-starts a non-blocking event loop.

> **Parameters** **timeout** – Timeout to wait until connection is ready.

run_forever()

Kick-starts a blocking loop to wait for events.

Depending on the implementations, and the client applications, running this might be required or not.

send_on_ready(*message*)

Send message to ROS once the connection is established.

If a connection to ROS is already available, the message is sent immediately.

> **Parameters** **message** (*[Message](#)*) – ROS Bridge Message to send.

set_param(*name*, *value*, *callback=None*, *errback=None*)

Set the value of a parameter from the ROS Parameter Server.

Note: To make this a blocking call, pass `None` to the `callback` parameter .

---

terminate()

Signals the termination of the main event loop.

roslibpy.**set_rosapi_timeout**(*timeout*)

Set the default number of seconds to wait for a response before raising an exception.

> **Parameters** **timeout** (`int`) – Duration in seconds.

1.3.3 Main ROS concepts

Topics

ROS is a communication infrastructure. In ROS, different **nodes** communicate with each other through messages. **ROS messages** are represented by the *[Message](#)* class and are passed around via *[Topics](#)* using a **publish/subscribe** model.

class roslibpy.**Message**(*values=None*)

Message objects used for publishing and subscribing to/from topics.

A message is fundamentally a dictionary and behaves as one.

class roslibpy.**Header**(*seq=None*, *stamp=None*, *frame_id=None*)

Represents a message header of the ROS type std_msgs/Header.

class roslibpy.**Topic**(*ros*, *name*, *message_type*, *compression=None*, *latch=False*, *throttle_rate=0*, *queue_size=100*, *queue_length=0*, *reconnect_on_close=True*)

Publish and/or subscribe to a topic in ROS.

> **Parameters**
>
> • **ros** (*[Ros](#)*) – Instance of the ROS connection.
>
> • **name** (`str`) – Topic name, e.g. `/cmd_vel`.
>
> • **message_type** (`str`) – Message type, e.g. `std_msgs/String`.
>
> • **compression** (`str`) – Type of compression to use, e.g. *png*. Defaults to *None*.

- **throttle_rate** (`int`) – Rate (in ms between messages) at which to throttle the topics.

- **queue_size** (`int`) – Queue size created at bridge side for re-publishing webtopics.

- **latch** (`bool`) – True to latch the topic when publishing, False otherwise.

- **queue_length** (`int`) – Queue length at bridge side used when subscribing.

- **reconnect_on_close** (`bool`) – Reconnect topic (both publisher & subscriber) on reconnection.

advertise()
> Register as a publisher for the topic.

property is_advertised
> Indicate if the topic is currently advertised or not.

> > **Returns** *bool* – True if advertised as publisher of this topic, False otherwise.

property is_subscribed
> Indicate if the topic is currently subscribed or not.

> > **Returns** *bool* – True if subscribed to this topic, False otherwise.

publish(*message*)
> Publish a message to the topic.

> > **Parameters** **message** ([`Message`](#)) – ROS Bridge Message to publish.

subscribe(*callback*)
> Register a subscription to the topic.

> Every time a message is published for the given topic, the callback will be called with the message object.

> > **Parameters** **callback** – Function to be called when messages of this topic are published.

unadvertise()
> Unregister as a publisher for the topic.

unsubscribe()
> Unregister from a subscribed the topic.

Services

Besides the publish/subscribe model used with topics, ROS offers a request/response model via [`Services`](#).

class roslibpy.**Service**(*ros*, *name*, *service_type*, *reconnect_on_close=True*)
> Client/server of ROS services.

> This class can be used both to consume other ROS services as a client, or to provide ROS services as a server.

> > **Parameters**

> > - **ros** ([`Ros`](#)) – Instance of the ROS connection.

> > - **name** (`str`) – Service name, e.g. `/add_two_ints`.

> > - **service_type** (`str`) – Service type, e.g. `rospy_tutorials/AddTwoInts`.

advertise(*callback*)
> Start advertising the service.

> This turns the instance from a client into a server. The callback will be invoked with every request that is made to the service.

> If the service is already advertised, this call does nothing.

> **Parameters callback** – Callback invoked on every service call. It should accept two parameters: *service_request* and *service_response*. It should return *True* if executed correctly, otherwise *False*.

call(*request*, *callback=None*, *errback=None*, *timeout=None*)
> Start a service call.

Note: The service can be used either as blocking or non-blocking. If the `callback` parameter is `None`, then the call will block until receiving a response. Otherwise, the service response will be returned in the callback.

> **Parameters**
>
> * **request** ([`ServiceRequest`](#)) – Service request.
>
> * **callback** – Callback invoked on successful execution.
>
> * **errback** – Callback invoked on error.
>
> * **timeout** – Timeout for the operation, in seconds. Only used if blocking.
>
> **Returns** *object* – Service response if used as a blocking call, otherwise `None`.

property is_advertised
> Service servers are registered as advertised on ROS.
>
> This class can be used to be a service client or a server.
>
> > **Returns** *bool* – True if this is a server, False otherwise.

unadvertise()
> Unregister as a service server.

class roslibpy.**ServiceRequest**(*values=None*)
> Request for a service call.

class roslibpy.**ServiceResponse**(*values=None*)
> Response returned from a service call.

Parameter server

ROS provides a parameter server to share data among different nodes. This service can be accessed via the [*Param*](#) class.

class roslibpy.**Param**(*ros*, *name*)
> A ROS parameter.
>
> > **Parameters**
> >
> > * **ros** ([*Ros*](#)) – Instance of the ROS connection.
> >
> > * **name** (str) – Parameter name, e.g. `max_vel_x`.

delete(*callback=None*, *errback=None*, *timeout=None*)
> Delete the parameter.

Note: This method can be used either as blocking or non-blocking. If the `callback` parameter is `None`, the call will block until completion.

> **Parameters**
>
> - **callback** – Callable function to be invoked when the operation is completed.
> - **errback** – Callback invoked on error.
> - **timeout** – Timeout for the operation, in seconds. Only used if blocking.

get(*callback=None*, *errback=None*, *timeout=None*)
    Fetch the current value of the parameter.

> ---
> **Note:** This method can be used either as blocking or non-blocking. If the `callback` parameter is `None`, the call will block and return the parameter value. Otherwise, the parameter value will be passed on to the callback.
> ---

> **Parameters**
>
> - **callback** – Callable function to be invoked when the operation is completed.
> - **errback** – Callback invoked on error.
> - **timeout** – Timeout for the operation, in seconds. Only used if blocking.
>
> **Returns** *object* – Parameter value if used as a blocking call, otherwise `None`.

set(*value*, *callback=None*, *errback=None*, *timeout=None*)
    Set a new value to the parameter.

> ---
> **Note:** This method can be used either as blocking or non-blocking. If the `callback` parameter is `None`, the call will block until completion.
> ---

> **Parameters**
>
> - **callback** – Callable function to be invoked when the operation is completed.
> - **errback** – Callback invoked on error.
> - **timeout** – Timeout for the operation, in seconds. Only used if blocking.

Time

To represent time, there is the concept of ROS time primitive type, which consists of two integers: seconds since epoch and nanoseconds since seconds.

class roslibpy.**Time**(*secs*, *nsecs*)
    Represents ROS time with two integers: seconds since epoch and nanoseconds since seconds.

static from_sec(*float_secs*)
    Create new Time instance from a float seconds representation (e.g. `time.time()`).

is_zero()
    Return `True` if zero (secs and nsecs) otherwise `False`.

static now()
    Create new Time instance from the current system time (not ROS time).

property nsecs
    Nanoseconds since seconds.

property secs
>   Seconds since epoch.

to_nsec()
>   Return time as nanoseconds from epoch.

to_sec()
>   Return time as float seconds representation (same as `time.time()`).

1.3.4 Actionlib

Another way to interact with ROS is through the **actionlib** stack. Actions in ROS allow to execute preemptable tasks, i.e. tasks that can be interrupted by the client.

Actions are used via the `ActionClient` to which `Goals` can be added. Each goal emits events that can be listened to in order to react to the updates from the action server. There are four events emmitted: **status**, **result**, **feedback**, and **timeout**.

class `roslibpy.actionlib.Goal`(*action_client*, *goal_message*)
>   Goal for an action server.
>
>   After an event has been added to an action client, it will emit different events to indicate its progress:
>
>   - `status`: fires to notify clients on the current state of the goal.
>
>   - `feedback`: fires to send clients periodic auxiliary information of the goal.
>
>   - `result`: fires to send clients the result upon completion of the goal.
>
>   - `timeout`: fires when the goal did not complete in the specified timeout window.
>
>   >   **Parameters**
>   >
>   >   - **action_client** (`ActionClient`) – Instance of the action client associated with the goal.
>   >
>   >   - **goal_message** (`Message`) – Goal for the action server.

cancel()
>   Cancel the current goal.

property is_finished
>   Indicate if the goal is finished or not.
>
>   >   **Returns** *bool* – True if finished, False otherwise.

send(*result_callback=None*, *timeout=None*)
>   Send goal to the action server.
>
>   >   **Parameters**
>   >
>   >   - **timeout** (`int`) – Timeout for the goal's result expressed in seconds.
>   >
>   >   - **callback** (`callable`) – Function to be called when a result is received.
>   >
>   >   - **It is a shorthand for hooking on the ``result`` event.**

wait(*timeout=None*)
>   Block until the result is available.
>
>   If `timeout` is `None`, it will wait indefinitely.
>
>   >   **Parameters** **timeout** (`int`) – Timeout to wait for the result expressed in seconds.
>
>   >   **Returns** Result of the goal.

class roslibpy.actionlib.**ActionClient**(*ros*, *server_name*, *action_name*, *timeout=None*, *omit_feedback=False*, *omit_status=False*, *omit_result=False*)

    Client to use ROS actions.

    Parameters

- **ros** (*Ros*) – Instance of the ROS connection.
- **server_name** (str) – Action server name, e.g. /fibonacci.
- **action_name** (str) – Action message name, e.g. actionlib_tutorials/ FibonacciAction.
- **timeout** (int) – **Deprecated.** Connection timeout, expressed in seconds.

    add_goal(*goal*)

        Add a goal to this action client.

        Parameters goal (*Goal*) – Goal to add.

    cancel()

        Cancel all goals associated with this action client.

    dispose()

        Unsubscribe and unadvertise all topics associated with this action client.

class roslibpy.actionlib.**SimpleActionServer**(*ros*, *server_name*, *action_name*)

    Implementation of the simple action server.

    The server emits the following events:

- goal: fires when a new goal has been received by the server.
- cancel: fires when the client has requested the cancellation of the action.

    Parameters

- **ros** (*Ros*) – Instance of the ROS connection.
- **server_name** (str) – Action server name, e.g. /fibonacci.
- **action_name** (str) – Action message name, e.g. actionlib_tutorials/ FibonacciAction.

    is_preempt_requested()

        Indicate whether the client has requested preemption of the current goal.

    send_feedback(*feedback*)

        Send feedback.

        Parameters feedback (dict) – Dictionary of key/values of the feedback message.

    set_preempted()

        Set the current action to preempted (cancelled).

    set_succeeded(*result*)

        Set the current action state to succeeded.

        Parameters result (dict) – Dictionary of key/values to set as the result of the action.

    start(*action_callback*)

        Start the action server.

        Parameters

- **action_callback** – Callable function to be invoked when a new goal is received.

- **It takes one paramter containing the goal message.**

class roslibpy.actionlib.**GoalStatus**
Valid goal statuses.

1.3.5 TF

ROS provides a very powerful transform library called TF2, which lets the user keep track of multiple coordinate frames over time.

The **roslibpy** library offers access to it through the tf2_web_republisher via the *TFClient* class.

class roslibpy.tf.**TFClient**(*ros*, *fixed_frame='/base_link'*, *angular_threshold=2.0*, *translation_threshold=0.01*, *rate=10.0*, *update_delay=50*, *topic_timeout=2000.0*, *server_name='/tf2_web_republisher'*, *repub_service_name='/republish_tfs'*)
A TF Client that listens to TFs from tf2_web_republisher.

> **Parameters**
>
> - **ros** (*Ros*) – Instance of the ROS connection.
>
> - **fixed_frame** (str) – Fixed frame, e.g. /base_link.
>
> - **angular_threshold** (float) – Angular threshold for the TF republisher.
>
> - **translation_threshold** (float) – Translation threshold for the TF republisher.
>
> - **rate** (float) – Rate for the TF republisher.
>
> - **update_delay** (int) – Time expressed in milliseconds to wait after a new subscription before update TFs.
>
> - **topic_timeout** (int) – Timeout parameter for the TF republisher expressed in milliseconds.
>
> - **repub_service_name** (str) – Name of the republish tfs service, e.g. /republish_tfs.

dispose()
Unsubscribe and unadvertise all topics associated with this instance.

subscribe(*frame_id*, *callback*)
Subscribe to the given TF frame.

> **Parameters**
>
> - **frame_id** (str) – TF frame identifier to subscribe to.
>
> - **callback** (callable) – A callable functions receiving one parameter with *transform* data.

unsubscribe(*frame_id*, *callback*)
Unsubscribe from the given TF frame.

> **Parameters**
>
> - **frame_id** (str) – TF frame identifier to unsubscribe from.
>
> - **callback** (callable) – The callback function to remove.

update_goal()
Send a new service request to the tf2_web_republisher based on the current list of TFs.

1.4 Contributor's Guide

Contributions are always welcome and greatly appreciated!

1.4.1 Code contributions

We love pull requests from everyone! Here's a quick guide to improve the code:

1. Fork the repository and clone the fork.

2. Create a virtual environment using your tool of choice (e.g. `virtualenv`, `conda`, etc).

3. Install development dependencies:

```
pip install -r requirements-dev.txt
```

4. Run the docker container:

```
docker run -d -p 9090:9090 --name roslibpy_integration_tests gramaziokohler/
↪integration-tests-bridge /bin/bash -c "roslaunch /integration-tests.launch"
```

5. Make sure all tests pass:

```
invoke test
```

6. Start making your changes to the **main** branch (or branch off of it).

7. Make sure all tests still pass:

```
invoke test
```

8. Stop your docker container:

```
docker stop roslibpy_integration_tests
```

9. Add yourself to `AUTHORS.rst`.

10. Commit your changes and push your branch to GitHub.

11. Create a pull request through the GitHub website.

During development, use pyinvoke tasks on the command prompt to ease recurring operations:

- `invoke clean`: Clean all generated artifacts.
- `invoke check`: Run various code and documentation style checks.
- `invoke docs`: Generate documentation.
- `invoke test`: Run all tests and checks in one swift command.
- `invoke`: Show available tasks.

1.4.2 Documentation improvements

We could always use more documentation, whether as part of the introduction/examples/usage documentation or API documentation in docstrings.

Documentation is written in reStructuredText and use Sphinx to generate the HTML output.

Once you made the documentation changes locally, run the documentation generation:

```
invoke docs
```

1.4.3 Bug reports

When reporting a bug please include:

- Operating system name and version.
- ROS version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

1.4.4 Feature requests and feedback

The best way to send feedback is to file an issue on Github. If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.

1.5 Authors

- Gramazio Kohler Research @gramaziokohler
- Gonzalo Casas <casas@arch.ethz.ch> @gonzalocasas
- Mathias Lüdtke @ipa-mdl
- Beverly Lytle @beverlylytle
- Alexis Jeandeau @jeandeaual
- Hiroyuki Obinata @obi-t4
- Pedro Pereira @MisterOwlPT
- Domenic Rodriguez @DomenicP

1.6 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

1.6.1 Unreleased

Added

Changed

Fixed

Deprecated

Removed

1.6.2 1.5.0

Added

- Added a wait event to close on IronPython to ensure the close request is sent before returning.

Changed

Fixed

- Fixed reconnection behavior on IronPython which would trigger reconnects even after a manual disconnect.

Deprecated

Removed

1.6.3 1.4.2

Added

Changed

- Added `tls` to the twisted requirement (#111).

Fixed

Deprecated

Removed

1.6.4 1.4.1

Added

Changed

Fixed

- Fixed bug with action client/server and now they work as expected.
- Fixed Python 2.7 incompatibilities introduced in 1.4.0.

Deprecated

Removed

1.6.5 1.4.0

Added

Changed

- Switched to `black` for python code formatting.

- Fix incompatible settings between `black` and `flake8`.

- Updated Github Actions workflows to remove python 3.6 builds.

- Replaced occurrences of `raise Exception` with more specific `Exception` subclasses.

Fixed

Deprecated

Removed

1.6.6 1.3.0

Added

- Added function to set the default timeout value.

- Added ROS host and port parameters to the command-line interface.

Fixed

- Fixed #87 where a goal could be marked as terminal on result alone rather than both result and status.

- Ensure input of `Time` is always two integers.

1.6.7 1.2.1

Added

Changed

Fixed

- Fixed blocking issues on the Twisted/Autobahn-based implementation of websockets.

Deprecated

Removed

1.6.8 1.2.0

Changed

- Changed behavior: Advertising services automatically reconnect when websockets is reconnected.

- References to ROS master change to ROS.

Added

- Added `Header` and `Time` data types.

- Added ROS API method to retrieve current ROS time: `ros.get_time`.

1.6.9 1.1.0

Added

- Added `set_initial_delay`, `set_max_delay` and `set_max_retries` to `RosBridgeClientFactory` to control reconnection parameters.

- Added `closing` event to `Ros` class that gets triggered right before closing the connection.

1.6.10 1.0.0

Changed

- Changed behavior: Topics automatically reconnect when websockets is reconnected.

Added

- Added blocking behavior to more ROS API methods: `ros.get_nodes` and `ros.get_node_details`.

- Added reconnection support to IronPython implementation of websockets.

- Added automatic topic reconnection support for both subscribers and publishers.

Fixed

- Fixed reconnection issues on the Twisted/Autobahn-based implementation of websockets.

1.6.11 0.7.1

Fixed

- Fixed blocking service calls for Mac OS.

1.6.12 0.7.0

Changed

- The non-blocking event loop runner `run()` now defaults to 10 seconds timeout before raising an exception.

Added

- Added blocking behavior to ROS API methods, e.g. `ros.get_topics`.

- Added command-line mode to ROS API, e.g. `roslibpy topic list`.

- Added blocking behavior to the `Param` class.

- Added parameter manipulation methods to `Ros` class: `get_param`, `set_param`, `delete_param`.

1.6.13 0.6.0

Changed

- For consistency, `timeout` parameter of `Goal.send()` is now expressed in **seconds**, instead of milliseconds.

Deprecated

- The `timeout` parameter of `ActionClient()` is ignored in favor of blocking until the connection is established.

Fixed

- Raise exceptions when timeouts expire on ROS connection or service calls.

Added

- Support for calling a function in a thread from the Ros client.
- Added implementation of a Simple Action Server.

1.6.14 0.5.0

Changed

- The non-blocking event loop runner now waits for the connection to be established in order to minimize the need for `on_ready` handlers.

Added

- Support blocking and non-blocking service calls.

Fixed

- Fixed an internal unsubscribing issue.

1.6.15 0.4.1

Fixed

- Resolve reconnection issues.

1.6.16 0.4.0

Added

- Add a non-blocking event loop runner.

1.6.17 0.3.0

Changed

- Unsubscribing from a listener no longer requires the original callback to be passed.

1.6.18 0.2.1

Fixed

- Fix JSON serialization error on TF Client (on Python 3.x).

1.6.19 0.2.0

Added

- Add support for IronPython 2.7.

Changed

- Handler `on_ready` now defaults to run the callback in thread.

Deprecated

- Rename `run_event_loop` to the more fitting `run_forever`.

1.6.20 0.1.1

Fixed

- Minimal documentation fixes.

1.6.21 0.1.0

Added

- Initial version.

1.7 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

r