00A0 $\exists$ $\forall$ $\subseteq$ 2713x 27FA$\Longleftrightarrow$ $\surd$ $\sqrt[3]{}$ $\oplus$ $\otimes$

roslibpy-docs-zh

1.1.0

2021 04 22

Gramazio Kohler Research

Wu Xin

:

Python ROS Bridge library **roslibpy** Python IronPython ROS WebSockets rosbridge 2.0 publishing subscribing service calls actionlib TF ROS

rospy **ROS** Linux ROS

roslibpy API roslibjs

2020 7 8 **roslibpy** 1.1.0 GitHub

CHAPTER 1

Python ROS Bridge library **roslibpy** Python IronPython ROS WebSockets rosbridge 2.0 publishing subscribing service calls actionlib TF ROS

rospy **ROS** Linux ROS

roslibpy API roslibjs

1.1

- (Topic)
- (Service)
- ROS (get/set/delete)
- ROS API
- Actionlib
- tf2_web_republisher TF Client

Roslibpy Python 2.7 Python 3.x IronPython 2.7

1.2

`pip` **roslibpy**:

```
pip install roslibpy
```

IronPython `pip` :

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

1.3

1.4

- roslibpy
-
- :

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

pyinvoke

- `invoke clean`:
- `invoke check`:
- `invoke docs`:
- `invoke test`:
- `invoke`:

1.5

roslibpy

- semver
  - `patch`: BUG
  - `minor`: Feature
  - `major`:
- `CHANGELOG.rst`
- :

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

- Profit!

1.6 Credits

roslibjs Python roslibjs

CHAPTER 2

roslibpy ROS

: ROS **rosbridge server** **TF2 web republisher** *ROS*

ROS `host` `'localhost'` ROS master *IP*

: `port` `9090` `rosbridge` `9090`

2.1

`roslibpy`:

```
>>> import roslibpy
```

:

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

:

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

(• ω •)y

2.2

Python

ros-hello-world.py

```
import roslibpy

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

:

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

ROS

2.3

run() ROS *roslibpy.Ros* run_forever()

roslibpy.Ros.on_ready()

run_forever() on_ready()

```
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()
```

: run() run_forever()

2.4

rosbridge

- *roslibpy.Ros.close()*: websocket . *roslibpy.Ros.connect()*:
- *roslibpy.Ros.terminate()*:

: twisted/authbahn twisted (reacters)

2.5

rosbridge roslibpy

2.6 Hello World: Topics

ROS Hello World / talker listener ROS

roslibpy

2.6.1 talker

ROS Ctrl+C

```
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

: ROS std_msgs/String Python

2.6.2 listener

Listener

```
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

2.6.3

talker :

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

listener :

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

: Ros master

2.7 Services

ROS

get_loggers

```
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

2.8 Services

ROS

ROS std_srvs/SetBool :

```
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

  :

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

  Ctrl+C

  :

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

  :

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

: **roslibpy** *API*

2.9 Actions

ROS **Actions**

roslibpy action *roslibpy.actionlib.SimpleActionServer* action

action action actionlib_tutorials

2.9.1 Action

```
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

  :

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

action Ctrl+C

2.9.2 Action

action

action

```
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,
                               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

  :

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

action

roslibpy.actionlib.Goal.wait() result

2.10 ROS API

ROS API API Python

2.10.1

ROS

:

```
$ 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
```

2.10.2 Python

Python ROS API

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

- *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()*

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

2.11

roslibpy

2.11.1 Enable debug logging

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

```
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()
```

2.11.2 Check roundtrip message latency

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

```
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 MASTER 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
```

2.11.3 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.

```
import time
import logging

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 to_epoch(stamp):
    stamp_secs = stamp['secs']
    stamp_nsecs = stamp['nsecs']
    return stamp_secs + stamp_nsecs*1e-9

def from_epoch(stamp):
    stamp_secs = int(stamp)
    stamp_nsecs = (stamp - stamp_secs) * 1e9
    return {'secs': stamp_secs, 'nsecs': stamp_nsecs}

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

publisher = roslibpy.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 = roslibpy.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 = dict(frame_id='', seq=seq, stamp=from_epoch(time.time()))
    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.

2.11.4 Publish images

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

```
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()
```

2.11.5 Subscribe to images

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

```
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)

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()
```

pull request issue tracker

CHAPTER 3

API

Robot Web Tools ROS bridge suite WebSockets ROS
ROS bridge protocol JSON publishing, subscribing, service calls, actionlib, TF ROS

3.1 ROS

ROS `rosbridge`

ROS master **rosbridge suite**:

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

:

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

: `roslaunch` `roscore` `roscore`

3.2 ROS

ROS *Ros*

ROS

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

Args: 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.

Args: callback: Callable function to be invoked from the thread. timeout (:obj: 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 the ROS Master once the connection is established.

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

Args: 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.

Args: callback (`callable`): Callable function to be invoked.

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

Args: 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 the ROS Master 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.

Args: message (*Message*): ROS Bridge Message containing the request. timeout (:obj: 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`()
Disconnect from ROS master.

`connect`()
Connect to ROS master.

`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.

Args: 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_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`.

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

Returns: int: An auto-incremented ID.

`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.

Args: 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.

Args: 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.

Args: 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.

Args: 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 the ROS Master once the connection is established.

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

Args: 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.

3.3 ROS

3.3.1

ROS ROS , messages **ROS messages** *Message* *Topics* /

`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.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.

Args: 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 the topic (both for publisher and subscribers) if a reconnection is detected.

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

`is_advertised`
Indicate if the topic is currently advertised or not.

Returns: bool: True if advertised as publisher of this topic, False otherwise.

`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.

Args: 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.

Args: 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.

3.3.2

/ ROS *Services* /

class roslibpy.Service(*ros*, *name*, *service_type*)
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.

Args: 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.

Args:

callback: Callback invoked on every service call. It should accept two parameters: *service_req*
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.

Args: 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**.

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.

3.3.3

ROS *Param*

class roslibpy.Param(*ros*, *name*)
A ROS parameter.

Args: 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.

Args: 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.

Args: 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.

Args: 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.

3.4 Actionlib

ROS **actionlib** ROS Actions

Actions *ActionClient* *Goals* goal Action **status result feedback 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.

Args: 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.

`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.

Args: 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.

Args: 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.

Args: 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.

Args: 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.

Args: 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.

Args: 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.

Args: result (`dict`): Dictionary of key/values to set as the result of the action.

`start(`*action_callback*`)`
Start the action server.

Args: 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.

3.5 TF

ROS TF2 **roslibpy** *TFClient* tf2_web_republisher

`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.

Args: 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 to update the TF republisher's list of 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.

Args: 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.

Args: 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.

CHAPTER 4

4.1

pull request

1. Fork clone
2. `virtualenv conda`
3. :

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

4. :

```
invoke test
```

5.
6. :

```
invoke test
```

7. `AUTHORS.rst`
8. Commit+push GitHub
9. GitHub pull request

pyinvoke

- `invoke clean`:
- `invoke check`:

- `invoke docs`:
- `invoke test`:
- `invoke`:

4.2

/ / docstrings API
reStructuredText Sphinx HTML
:

```
invoke docs
```

4.3 BUG

BUG

-
- ROS
-
- BUG

4.4 Feature

GitHub issue feature

-
-

CHAPTER 5

5.1

- Gramazio Kohler Research @gramaziokohler
- Gonzalo Casas <casas@arch.ethz.ch> @gonzalocasas
- Mathias Ldtke @ipa-mdl

5.2

- Wu Xin @XinArkh

CHAPTER 6

changelog Keep a Changelog (Semantic Versioning)

6.1 Unreleased

Changed

Added

Fixed

6.2 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.

6.3 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

6.4 0.7.1

Fixed

- Fixed blocking service calls for Mac OS

6.5 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`.

6.6 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.

6.7 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.

6.8 0.4.1

Fixed

- Resolve reconnection issues.

6.9 0.4.0

Added

- Add a non-blocking event loop runner

6.10 0.3.0

Changed

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

6.11 0.2.1

Fixed

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

6.12 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`

6.13 0.1.1

Fixed

- Minimal documentation fixes

6.14 0.1.0

Added

- Initial version

CHAPTER 7

roslibpy

Sphinx + reStructuredText + readthedocs

GitHub star

>> **roslibpy** roslibpy <<

>> roslibpy-docs-zh <<

Wu Xin

2019.4.26.

2019.6.8. edited.

2019.7.8. edited.

CHAPTER 8

- genindex
- modindex
- search

r

S

T

U

W