wrapyfi

Release 0.4.44

Fares Abawi

May 05, 2024

OVERVIEW:

WRAPYFI

Wrapyfi is multi-middleware Python package supporting ROS, ROS 2, YARP and ZeroMQ.

Wrapyfi is a middleware communication wrapper for transmitting data across nodes, without altering the operation pipeline of your Python scripts. Wrapyfi introduces a number of helper functions to make middleware integration possible without the need to learn an entire framework, just to parallelize your processes on multiple machines. Wrapyfi supports YARP, ROS, ROS 2, and ZeroMQ.

To integrate Wrapyfi with your scripts, add the decorators describing the transmitting and listening method parameters.

CHAPTER

ONE

ATTRIBUTION

Please refer to the following paper when citing Wrapyfi in academic work:

```
@inproceedings{abawi2024wrapyfi,
  title = {Wrapyfi: A Python Wrapper for Integrating Robots, Sensors, and Applications␣
↪across Multiple Middleware},
  author = {Abawi, Fares and Allgeuer, Philipp and Fu, Di and Wermter, Stefan},
  booktitle = {Proceedings of the ACM/IEEE International Conference on Human-Robot␣
↪Interaction (HRI '24)},
  year = {2024},
  organization = {ACM},
  isbn = {79-8-4007-0322-5},
  doi = {10.1145/3610977.3637471},
  url = {https://github.com/fabawi/wrapyfi}
}
```

CHAPTER

TWO

GETTING STARTED

Before using Wrapyfi, YARP, ROS, or ZeroMQ must be installed.

- Follow the YARP installation guide. Note that the iCub package is not needed for Wrapyfi to work and does not have to be installed if you do not intend to use the iCub robot.
- For installing ROS, follow the ROS installation guide [Ubuntu][Windows]. We recommend installing ROS on Conda using the RoboStack environment. Additionally, the Wrapyfi ROS interfaces must be built to support messages needed for audio transmission
- For installing ROS 2, follow the ROS 2 installation guide [Ubuntu][Windows]. We recommend installing ROS 2 on Conda using the RoboStack environment. Additionally, the Wrapyfi ROS 2 interfaces must be built to support messages and services needed for audio transmission and the REQ/REP pattern
- ZeroMQ can be installed using pip: `pip install pyzmq`. The xpub-xsub pattern followed in our ZeroMQ implementation requires a proxy broker. A broker is spawned by default as a daemon process. To avoid automatic spawning, pass the argument `start_proxy_broker=False` to the method register decorator. A standalone broker can be found here

2.1 Compatibility

- Operating System
  - Ubuntu >= 18.04 (Not tested with earlier versions of Ubuntu or other Linux distributions)
  - Windows >= 10 [*beta support*]:
    - Multiprocessing is disabled. ZeroMQ brokers spawn as threads only
    - Not tested with YARP and ROS 2
    - ROS only tested within mamba/micromamba environment installed using RoboStack
    - ROS and ROS 2 interfaces not tested
    - Installation instructions across Wrapyfi guides and tutorials are not guaranteed to be compatible with Windows 11
  - MacOS 10.14 Mojave
- Python >= 3.6
- OpenCV >= 4.2
- NumPy >= 1.19
- YARP >= v3.3.2
- ROS Noetic Ninjemys

- ROS 2 Humble Hawksbill | Galactic Geochelone | Foxy Fitzroy
- PyZMQ 16.0, 17.1 and 19.0

2.2 Installation

You can install Wrapyfi with **pip** or from source.

2.2.1 Pip

To install all the necessary components for the majority of common uses of Wrapyfi (e.g., NativeObject, Image, Audio, etc.) using **pip**, this process installs both Wrapyfi and its dependencies, like NumPy and OpenCV (`opencv-contrib-python`, `opencv-headless`, and `opencv-python` are supported), that are essential for various workloads, along with ZeroMQ being the default middleware. This option is the best for users running Wrapyfi out of the box in a newly created environment (without any middleware installed beforehand), installing `numpy`, `opencv-contrib-python`, and `pyzmq`:

```
pip install wrapyfi[all]
```

Note that most plugins require additional dependencies and should be installed separately.

or when installing Wrapyfi on a *server* (headless) including `numpy`, `opencv-python-headless`, and `pyzmq`:

```
pip install wrapyfi[headless]
```

Other middleware such as ROS are environment-specific and require dependencies that cannot be installed using pip. Wrapyfi **could** and should be used within such environments with minimal requirements to avoid conflicts with existing NumPy and OpenCV packages:

```
pip install wrapyfi
```

2.2.2 Source (Pip)

Clone this repository:

```
git clone --recursive https://github.com/fabawi/wrapyfi.git
cd wrapyfi
```

You can choose to install minimal dependencies including `numpy`, `opencv-contrib-python`, and `pyzmq`, for running a basic Wrapyfi script:

```
pip install .[all]
```

or when installing Wrapyfi on a *server* (headless) including `numpy`, `opencv-python-headless`, and `pyzmq`:

```
pip install .[headless]
```

or install Wrapyfi *without* NumPy, OpenCV, and ZeroMQ:

```
pip install .
```

2.2.3 Docker

Wrapyfi Docker images can be pulled/installed directly from the modularml/wrapyfi repository on the Docker Hub. Dockerfiles for all supported environments can be built as well by following the Wrapyfi Docker instructions.

2.3 Usage

Wrapyfi supports two patterns of communication:

- **Publisher-Subscriber** (PUB/SUB): A publisher sends data to a subscriber accepting arguments and executing methods on the publisher's end. e.g., with YARP

```
# Just your usual Python class


class HelloWorld(object):




    def send_message(self):
        msg = input("Type your message: ")
        obj = {"message": msg}
        return obj,


hello_world = HelloWorld()




while True:
    my_message, = hello_world.send_message()
    print(my_message)
```

```
from wrapyfi.connect.wrapper import MiddlewareCommunicator


class HelloWorld(MiddlewareCommunicator):
    @MiddlewareCommunicator.register("NativeObject", "yarp",
                                     "HelloWorld",
                                     "/hello/my_message",
                                     carrier="", should_wait=True)
    def send_message(self):
        msg = input("Type your message: ")
        obj = {"message": msg}
        return obj,


hello_world = HelloWorld()
```

(continues on next page)

(continued from previous page)

```
LISTEN = True
mode = "listen" if LISTEN else "publish"
hello_world.activate_communication(hello_world.send_message, mode=mode)

while True:
    my_message, = hello_world.send_message()
    print(my_message)
```

Run `yarpserver` from the command line. Now execute the Python script above (with Wrapyfi) twice setting `LISTEN = False` and `LISTEN = True`. You can now type with the publisher's command line and preview the message within the listener's

- **Request-Reply** (REQ/REP): A requester sends a request to a responder, which responds to the request in a synchronous manner. e.g., with ROS

```
# Just your usual Python class


class HelloWorld(object):




    def send_message(self, a, b):
        msg = input("Type your message: ")
        obj = {"message": msg,
               "a": a, "b": b, "sum": a + b}
        return obj,


hello_world = HelloWorld()




while True:
    my_message, = hello_world.send_message(a=1,
                                           b=2)
    print(my_message)
```

```
from wrapyfi.connect.wrapper import MiddlewareCommunicator


class HelloWorld(MiddlewareCommunicator):
    @MiddlewareCommunicator.register("NativeObject", "ros",
                                     "HelloWorld",
                                     "/hello/my_message",
                                     carrier="", should_wait=True)
    def send_message(self, a, b):
        msg = input("Type your message: ")
        obj = {"message": msg,
```

(continues on next page)

(continued from previous page)

```
                "a": a, "b": b, "sum": a + b}
        return obj,


hello_world = HelloWorld()

LISTEN = True
mode = "request" if LISTEN else "reply"
hello_world.activate_communication(hello_world.send_message, mode=mode)

while True:
    my_message, = hello_world.send_message(a=1 if LISTEN else None,
                                           b=2 if LISTEN else None)
    print(my_message)
```

Run `roscore` from the command line. Now execute the Python script above (with Wrapyfi) twice setting `LISTEN = False` and `LISTEN = True`. You can now type within the server's command line and preview the message within the client's. Note that the server's command line will not show the message until the client's command line has been used to send a request. The arguments are passed from the client to the server and the server's response is passed back to the client.

For more examples of usage, refer to the user guide. Run scripts in the examples directory for trying out Wrapyfi.

CHAPTER

THREE

SUPPORTED FORMATS

3.1 Middleware

- **YARP**
- **ROS**
- **ROS 2**
- **ZeroMQ** [*beta feature*]:
  - `should_wait` trigger introduced with event monitoring
  - Event monitoring currently cannot be disabled

3.2 Serializers

- **JSON**
- **msgpack**
- **protobuf**

3.3 Data Structures

Supported Objects by the `NativeObject` type include:

- **NumPy Array | Generic**
- **PyTorch Tensor**
- **TensorFlow 2 Tensor**
- **JAX Tensor**
- **Trax Array**
- **MXNet Tensor**
- **PaddlePaddle Tensor**
- **pandas DataFrame | Series**
- **Pillow Image**
- **PyArrow Array**

- **CuPy Array**
- **Xarray DataArray | Dataset**
- **Dask Array | DataFrame**
- **Zarr Array | Group**
- **Pint Quantity**
- **Gmpy 2 MPZ**
- **MLX Tensor**

3.4 Image

Supported Objects by the `Image` type include:

- **NumPy Array** [*supports many libraries including scikit-image, imageio, Open CV, imutils, matplotlib.image, and Mahotas*]

3.5 Sound

Supported Objects by the `AudioChunk` type include:

- Tuple(**NumPy Array**, int) [*supports the sounddevice format*]

CHAPTER

FOUR

USER GUIDE

To Wrapyfi your code:

```
from wrapyfi.connect.wrapper import MiddlewareCommunicator

class TheClass(MiddlewareCommunicator)
        ...

        @MiddlewareCommunicator.register(...)
        @MiddlewareCommunicator.register(...)
        def encapsulated_method(...):
            ...
            return encapsulated_a, encapsulated_b

        def encapsulating_method(...)
            ...
            encapsulated_a, encapsulated_b = self.encapsulated_method(...)
            ...
            return result,


the_class = TheClass()
the_class.activate_communication(the_class.encapsulated_method, mode="publish")
while True:
    the_class.encapsulating_method(...)
```

The primary component for facilitating communication is the `MiddlewareCommunicator`. To register the methods for a given class, the class should inherit the `MiddlewareCommunicator`. Any method decorated with `@MiddlewareCommunicator.register(<Data structure type>, <Communicator>, <Class name>, <Topic name>)` is automatically registered by Wrapyfi.

The `<Data structure type>` is the publisher/listener type for a given method's return. The supported data types are listed *here* section.

The `<Communicator>` defines the communication medium e.g.: `yarp`, `ros2`, `ros`, or `zeromq`. The default communicator is `zeromq` but can be replaced by setting the environment variables `WRAPYFI_DEFAULT_COMMUNICATOR` or `WRAPYFI_DEFAULT_MWARE` (`WRAPYFI_DEFAULT_MWARE` overrides `WRAPYFI_DEFAULT_COMMUNICATOR` when both are provided) to the middleware of choice e.g.:

```
        export WRAPYFI_DEFAULT_COMMUNICATOR=yarp
```

The `<Class name>` serves no purpose in the current Wrapyfi version, but has been left for future support of module-level decoration, where the methods don't belong to a class, and must therefore have a unique identifier for declaration

in the *configuration files*.

The <Topic name> is the name used for the connected topic and is dependent on the middleware platform. The listener and publisher receive the same topic name.

The `@MiddlewareCommunicator.register` decorator is defined for each of the method's returns in the same order. As shown in the example above, the first decorator defines the properties of `encapsulated_a`'s publisher and listener, whereas the second decorator belongs to `encapsulated_b`. A decorated method must always return a tuple which can easily be enforced by adding a `comma` after the return in case a single variable is returned. Lists are also supported for single returns e.g.:

```
        @MiddlewareCommunicator.register([...], {...}], [...], {...}], [...])
        @MiddlewareCommunicator.register(...)
        def encapsulated_method(...):
            ...
            encapsulated_a = [[...], [...], [...]]
            ...
            return encapsulated_a, encapsulated_b
```

Warning: Methods with a single return should be followed by a comma e.g., return encapsulated a, . This explicitly casts the return as a tuple to avoid confusion with list returns as single return element/s

Each of the list's returns is encapsulated with its own publisher and listener, with the named arguments transmitted as a single dictionary within the list. Notice that `encapsulated_a` returns a list of length 3, therefore, the first decorator contains 3 list configurations as well. This is useful especially when transmitting multiple images or audio chunks over YARP, ROS, and ROS 2. Note that by using a single `NativeObject` as a <Data structure type>, the same can be achieved. However, the implementation of the `NativeObject` for most middleware serializes the objects as strings before transmission. The `NativeObject` may result in a greater overhead and should only be used when multiple nesting depths are required or the objects within a list are not within the *supported data structure types*.

4.1 Argument Passing

The `$` symbol is used in Wrapyfi to specify that a decorator should update its arguments according to the arguments of the decorated method. This can be useful when the decorator needs to modify its behavior during runtime. For instance:

```
...
        @MiddlewareCommunicator.register('NativeObject',
           '$0', 'ExampleCls', '/example/example_arg_pass',
           carrier='tcp', should_wait='$blocking')
          def example_arg_pass(self, mware, msg='', blocking=True):
```

Setting the decorator's keyword argument `should_wait='$blocking'` expects the decorated method to receive a boolean `blocking` argument, altering the encapsulating decorator's behavior when the encapsulated method is called. Setting the decorator's second argument to `$0` acquires the value of `mware` (the first argument passed to `example_arg_pass`) and sets it as the middleware for that method. These arguments take effect on the first invocation of a method. Changing arguments after the first invocation results in no change in behavior, unless a `MiddlewareCommunicator` inheriting class for a given method is *closed*.

4.2 Closing and Deleting Classes

Currently, closing a connection requires closing all connections established by every method within that class.

> **Warning:** Selectively deactivating method connections is not supported

To close and delete a `MiddlewareCommunicator` inheriting class means that the middleware connection will be disconnected gracefully. The class references will be removed from all registries, the communication ports will be freed, and the instance will be destroyed. To close a class instance:

```
# assuming an existing instance-> example_instance = ExampleCls()
example_instance.close()
del example_instance
```

CHAPTER

FIVE

COMMUNICATION PATTERNS

Wrapyfi supports the publisher-subscriber *(PUB/SUB)* pattern as well as the request-reply *(REQ/REP)* pattern. The PUB/SUB pattern assumes message arguments are passed from the publisher-calling script to the publishing method. The publisher executes the method and the subscriber (listener) merely triggers the method call, awaits the publisher to execute the method, and returns the publisher's method returns. The REQ/REP pattern on the other hand assumes arguments from the client (requester) are sent to the server (responder or replier). Once the server receives the request, it passes the arguments to its own method, executes it, and replies to the client back with its method returns.

Warning: in REQ/REP, the requester transmits all arguments passed to the method as a dictionary encoded as a string. This is not ideal for predefined services, where the service expects a certain object/message type. A better approach would include the option to pass a single item of a certain value and type

5.1 Publishers and Listeners/Subscribers (PUB/SUB)

The publishers and listeners of the same message type should have identical constructor signatures. The current Wrapyfi version supports 4 universal message types for all middleware. The extended types such as `ROSMessage` and `ROS2Message` are exclusive to the provided middleware.

5.1.1 YARP:

Note: YARP publishers remain persistent. To disable persistence, pass the argument `persistent=False` to the `@MiddlewareCommunicator.register` decorator.

All messages are transmitted using the `yarp` Python bindings

- **Image**: Transmits and receives a `cv2` or `numpy` image using either `yarp.BufferedPortImageRgb` or `yarp.BufferedPortImageFloat`. When JPG conversion is specified, it uses a `yarp.BufferedPortBottle` message carrying a JPEG encoded string instead
- **AudioChunk**: Transmits and receives a `numpy` audio chunk with the sound properties using `yarp.Port` transporting `yarp.Sound`
- **NativeObject**: Transmits and receives a `json` string supporting all native Python objects, `numpy` arrays and *other formats* using `yarp.BufferedPortBottle`
- **Properties**: Transmits properties

5.1.2 ROS:

Warning: ROS requires a custom message to handle audio. This message must be compiled first before using Wrapyfi with ROS Audio. Refer to these instructions for compiling Wrapyfi ROS services and messages.

All messages are transmitted using the `rospy` Python bindings as topic messages

- **Image**: Transmits and receives a `cv2` or `numpy` image using `sensor_messages.msg.Image`. When JPG conversion is specified, uses the `sensor_messages.msg.CompressedImage` message instead
- **AudioChunk**: Transmits and receives a `numpy` audio chunk using `wrapyfi_ros_interfaces.msg.ROSAudioMessage`
- **NativeObject**: Transmits and receives a `json` string supporting all native Python objects, `numpy` arrays, and *other formats* using `std_msgs.msg.String`
- **Properties**: Transmits and receives parameters to/from the parameter server using the methods `rospy.set_param` and `rospy.get_param` respectively
- **ROSMessage**: Transmits and receives a single ROS message per return decorator. Note that currently, only common ROS interface messages are supported and detected automatically. This means that messages defined in common interfaces such as std_msgs, geometry_msgs, and sensor_msgs can be directly returned by the method do not need to be converted to native types

5.1.3 ROS 2:

Warning: ROS 2 requires a custom message to handle audio. This message must be compiled first before using Wrapyfi with ROS 2 Audio. Refer to these instructions for compiling Wrapyfi ROS 2 services and messages.

All messages are transmitted using the `rclpy` Python bindings as topic messages

- **Image**: Transmits and receives a `cv2` or `numpy` image using `sensor_messages.msg.Image`. When JPG conversion is specified, uses the `sensor_messages.msg.CompressedImage` message instead
- **AudioChunk**: Transmits and receives a `numpy` audio chunk using `wrapyfi_ros2_interfaces.msg.ROS2AudioMessage`
- **NativeObject**: Transmits and receives a `json` string supporting all native Python objects, `numpy` arrays, and *other formats* using `std_msgs.msg.String`
- **Properties**: Transmits properties
- **ROS2Message**: Transmits and receives a single ROS 2 message per return decorator

5.1.4 ZeroMQ:

Note: ZeroMQ exchanges in REQ/REP rely on a broker with a dedicated socket. By default, Wrapyfi will not spawn a new connection to the socket when multiple threads are created. For multi-threaded applications, this leads to race conditions. We avoid that by detecting whether a new instance of the socket is available in the thread's local storage. This multi-threading-friendly mode is enabled by passing `multi_threaded=True` to the `@MiddlewareCommunicator.register` decorator. This is only recommended when registering methods that are going to be multi-threaded.

All messages are transmitted using the `zmq` Python bindings. Transmission follows the proxied XPUB/XSUB pattern

- **Image**: Transmits and receives a `cv2` or `numpy` image wrapped in the `NativeObject` construct. Note that all `Image` types are transmitted as multipart messages, where the first element is the topic name and the second element is the header (e.g., timestamp), and the third element is the image itself
- **AudioChunk**: Transmits and receives a `numpy` audio chunk wrapped in the `NativeObject` construct
- **NativeObject**: Transmits and receives a `json` string supporting all native Python objects, `numpy` arrays and *other formats* using `zmq context.socket(zmq.PUB).send_multipart` for publishing and `zmq context.socket(zmq.SUB).receive_multipart` for receiving messages. The `zmq.PUB` socket is wrapped in a `zmq.proxy` to allow multiple subscribers to the same publisher. Note that all `NativeObject` types are transmitted as multipart messages, where the first element is the topic name and the second element is the message itself (Except for `Image`)
- **Properties**: Transmits properties

5.2 Servers and Clients (REQ/REP)

The servers and clients of the same message type should have identical constructor signatures. The current Wrapyfi version supports 3 universal message types for all middleware. The extended types such as `ROSMessage` and `ROS2Message` are exclusive to the provided middleware.

5.2.1 YARP:

All messages are transmitted using the `yarp` Python bindings for RPC communication. The requester encodes its arguments as a `json` string supporting all native Python objects, `numpy` arrays, and *other formats* using `yarp.Bottle`. The requester formats its arguments as *([args], {kwargs})*

- **Image**: Transmits and receives a `cv2` or `numpy` image encoded as a `json` string using `yarp.Bottle`. *JPG conversion is currently not supported*
- **AudioChunk**: Transmits and receives a `numpy` audio chunk encoded as a `json` string using `yarp.Bottle`
- **NativeObject**: Transmits and receives a `json` string supporting all native Python objects, `numpy` arrays, and *other formats* using `yarp.Bottle`

5.2.2 ROS:

Warning: ROS requires a custom service to handle audio. This service must be compiled first before using Wrapyfi with ROS Audio. Refer to these instructions for compiling Wrapyfi ROS services and messages.

All messages are transmitted using the `rospy` Python bindings as services. The requester encodes its arguments as a `json` string supporting all native Python objects, `numpy` arrays, and *other formats* using `std_msgs.msg.String`. The requester formats its arguments as *([args], {kwargs})*

- **Image**: Transmits and receives a `cv2` or `numpy` image using `sensor_messages.msg.Image` *JPG conversion is currently not supported*
- **AudioChunk**: Transmits and receives a `numpy` audio chunk using `wrapyfi_ros_interfaces.msg.ROSAudioMessage`
- **NativeObject**: Transmits and receives a `json` string supporting all native Python objects, `numpy` arrays, and *other formats* using `std_msgs.msg.String`

5.2.3 ROS 2:

Warning: ROS 2 requires custom services to handle arbitrary messages. These services must be compiled first before using Wrapyfi in this mode. Refer to these instructions for compiling Wrapyfi ROS 2 services.

All messages are transmitted using the rclpy Python bindings as services. The requester encodes its arguments as a `json` string supporting all native Python objects, `numpy` arrays, and *other formats* using `std_msgs.msg.String`. The requester formats its arguments as *([args], {kwargs})*

- **Image**: Transmits and receives a `cv2` or `numpy` image using `sensor_messages.msg.Image`
- **AudioChunk**: Transmits and receives a `numpy` audio chunk using `wrapyfi_ros2_interfaces.msg.ROS2AudioMessage`
- **NativeObject**: Transmits and receives a `json` string supporting all native Python objects, `numpy` arrays, and *other formats* using `std_msgs.msg.String`

5.2.4 ZeroMQ:

All messages are transmitted using the `zmq` Python bindings. Transmission follows the proxied XREP/XREQ pattern The requester encodes its arguments as a `json` string supporting all native Python objects, `numpy` arrays, and *other formats* using `zmq context.socket(zmq.REQ).send_multipart`. The requester formats its arguments as *([args], {kwargs})*

- **Image**: Transmits and receives a `cv2` or `numpy` image wrapped in the `NativeObject` construct
- **AudioChunk**: Transmits and receives a `numpy` audio chunk wrapped in the `NativeObject` construct
- **NativeObject**: Transmits and receives a `json` string supporting all native Python objects, `numpy` arrays, and *other formats* using `zmq context.socket(zmq.REP)` for replying and `zmq context.socket(zmq.REQ)` for receiving messages

5.3 Publisher- and Listener-specific Arguments

Warning: Differences are expected between the returns of publishers and listeners, sometimes due to compression methods (e.g., setting `jpg=True` when transmitting an **Image** compresses the image but the encoding remains the same), intentional setting of different devices for different tensors (refer to *device mapping for tensors*), and differences in library versions between receiving and transmitting plugins (refer to *plugins*).

To direct arguments specifically toward the publisher or subscriber without exposing one or the other to the same argument values, the corresponding arguments can be added to the dictionary `listener_kwargs` to control the listener only, or `publisher_kwargs` to control the publisher only. Both dictionaries can be passed directly to the Wrapyfi decorator. Since the transmitting and receiving arguments should generally be the same regardless of the communication pattern, `publisher_kwargs` and `listener_kwargs` also apply to the servers and clients respectively.

CHAPTER

SIX

COMMUNICATION SCHEMES

We introduce three communication schemes: **Mirroring**, **Channeling**, and **Forwarding**. These schemes are communication forms that can be useful in different scenarios.

6.1 Mirroring

For the REQ/REP pattern, mirroring is a communication scheme that allows a client to send arguments to a server, and receive the method returns back from the server. As for the PUB/SUB pattern, mirroring allows a publisher to send the returns of a method to a subscriber based on the publisher's method arguments. Following both patterns, the returns of a method are mirrored on the receiver and the sender side. This is useful when the pipeline for each receiver is identical, but we would like to delegate the processing to different publishers when processing requires more resources than a single publisher can provide.

6.1.1 Mirroring Example

In the mirroring_example.py, the module transmits a user input message from the publisher to a listener (PUB/SUB pattern), and displays the message along with other native objects on the listener and publisher side. Similarly, we transmit a user input message from the server to a client (REQ/REP pattern), when the client requests the message from the server. The example can be run from the examples/communication_schemes/ directory.

6.2 Forwarding

Forwarding is a communication scheme that allows a server or publisher to forward the method arguments to another server or publisher (acting as a client or listener), and in return, forwards the received messages to another client or listener. This is useful when the server or publisher is not able to communicate with the client or listener directly due to limited middleware support on the client or listener side. The middle server or publisher can then act as a bridge between the two, and forward the messages between them, effectively chaining the communication. The chain can be extended and is not limited to two servers or publishers.

6.2.1 Forwarding Example

In the forwarding_example.py, the module constantly publishes a string from `chain_A` to a listener on `chain_A`. The `chain_A` listener then forwards the message by publishing to `chain_B`. The string is then forwarded to a third instances which listens exclusively to `chain_B`, without needing to support the middleware used by `chain_A`. The example can be run from the examples/communication_schemes/ directory.

6.3 Channeling

Channeling differs from mirroring, in that there are multiple returns from a method. Disabling one or more of these returns is possible, allowing the server or publisher to transmit the message to multiple channels, each with a different topic, and potentially, a different middleware. This is useful for transmitting messages using the same method, but to different receivers based on what they choose to receive. Not all clients or subscribers require all the messages from a method, and can therefore selectively filter out what is needed and operate on that partial return.

6.3.1 Channeling Example

In the channeling_example.py, the module constantly publishes three data types (**NativeObject**, **Image**, and **AudioChunk**) over one or more middlware. The listeners can then choose to receive one or more of these data types, depending on the middleware they support. When `--mware_...` for one of the channels is not provided, it automatically disables the topic for that channel/s and returns a `None` type value. The example can be run from the examples/communication_schemes/ directory.

CHAPTER

SEVEN

CONFIGURATION

The `MiddlewareCommunicator`'s child class method modes can be independently set to:

- **publish**: Run the method and publish the results using the middleware's transmission protocol
- **listen**: Skip the method and wait for the publisher with the same port name to transmit a message, eventually returning the received message
- **reply**: Run the method and publish the results using the middleware's transmission protocol. Arguments are received from the requester
- **request**: Send a request to the replier in the form of arguments passed to the method. Skip the method and wait for the replier with the same port name to transmit a message, eventually returning the received message
- **none**(default): Run the method as usual without triggering publish, listen, request or reply. *hint*: Setting the mode to `None` (or `null` within a yaml configuration file) has the same effect
- **disable**: Disables the method and returns None for all its returns. Caution should be taken when disabling a method since it could break subsequent calls

These properties can be set by calling:

`activate_communication(<Method name>, mode=<Mode>)`

where <`Method name`> is the method's name (`string` name of method by definition) and <`Mode`> is the transmission mode (`"publish"`, `"listen"`, `"reply"`, `request`, `"none"` | `None`, `"disable"`) depending on the communication pattern . The `activate_communication` method can be called multiple times. <`Method name`> could also be a class instance method, by calling:

`activate_communication(<MiddlwareCommunicator instance>.method_of_class_instance, mode=<Mode>)`

for each decorated method within the class. This however requires modifying your scripts for each machine or process running on Wrapyfi. To overcome this limitation, use the `ConfigManager` e.g.:

```
from wrapyfi.config.manager import ConfigManager
ConfigManager(<Configuration file path *.yml>)
```

The `ConfigManager` is a singleton that must be called once before the initialization of any `MiddlewareCommunicator`. Initializing it multiple times has no effect. This limitation was created by design to avoid loading the configuration file multiple times.

The <`Configuration file path *.yml`>'s configuration file has a very simple format e.g.:

```
TheClass:
  encapsulated_method: "publish"
```

where `TheClass` is the class name, `encapsulated_method` is the method's name, and `publish` is the transmission mode. This is useful when running the same script on multiple machines, where one is set to publish and the other listens. Multiple instances of the same class' method can have different modes, which can be set independently using the configuration file. This can be achieved by providing the mode as a list:

```
TheClass:
  encapsulated_method:
        "publish"
        null
        "listen"
        "listen"
        "disable"
        null
```

where the list element index corresponds to the instance index. When providing a list, the number of list elements should correspond to the number of instances. If the number of instances exceeds the list length, the script exits and raises an error.

CHAPTER

EIGHT

ENVIRONMENT VARIABLES

Wrapyfi reserves specific environment variable names for the functionality of its internal components:

- `WRAPYFI_PLUGINS_PATH`: Path/s to *plugin* extension directories
- `WRAPYFI_DEFAULT_COMMUNICATOR` or `WRAPYFI_DEFAULT_MWARE` (`WRAPYFI_DEFAULT_MWARE` overrides `WRAPYFI_DEFAULT_COMMUNICATOR` when both are provided): Name of default when non is provided as the second argument to the Wrapyfi decorator.

ZeroMQ requires socket configurations that can be passed as arguments to the respective middleware constructor (through the Wrapyfi decorator) or using environment variables. Note that these configurations are needed both by the proxy and the message publisher and listener. The downside to such an approach is that all messages share the same configs. Since the proxy broker spawns once on first trigger (if enabled) as well as a singleton subscriber monitoring instance, using environment variables is the recommended approach to avoid unintended behavior. This can be achieved by setting:

- `WRAPYFI_ZEROMQ_SOCKET_IP`: IP address of the socket. Defaults to "127.0.0.1"
- `WRAPYFI_ZEROMQ_SOCKET_PUB_PORT`: The publishing socket port. Defaults to 5555
- `WRAPYFI_ZEROMQ_SOCKET_SUB_PORT`: The sub-socket port (listening port for the broker). Defaults to 5556
- `WRAPYFI_ZEROMQ_PUBSUB_MONITOR_TOPIC`: The topic name for the pub-sub monitor. Defaults to "ZEROMQ/CONNECTIONS"
- `WRAPYFI_ZEROMQ_PUBSUB_MONITOR_LISTENER_SPAWN`: Either spawn the pub-sub monitor listener as a "process" or "thread". Defaults to "process"
- `WRAPYFI_ZEROMQ_START_PROXY_BROKER`: Spawn a new broker proxy without running the standalone proxy broker. Defaults to "True"
- `WRAPYFI_ZEROMQ_PROXY_BROKER_SPAWN`: Either spawn broker as a "process" or "thread". Defaults to "process")
- `WRAPYFI_ZEROMQ_PARAM_POLL_INTERVAL`: Polling interval in milliseconds for the parameter server. Defaults to 1 (**currently not supported**)
- `WRAPYFI_ZEROMQ_PARAM_REQREP_PORT`: The parameter server request-reply port. Defaults to 5659 (**currently not supported**)
- `WRAPYFI_ZEROMQ_PARAM_PUB_PORT`: The parameter server pub-socket port. Defaults to 5655 (**currently not supported**)
- `WRAPYFI_ZEROMQ_PARAM_SUB_PORT`: The parameter server sub-socket port. Defaults to 5656 (**currently not supported**)

ROS and ROS 2 queue sizes can be set by:

- `WRAPYFI_ROS_QUEUE_SIZE`: Size of the queue buffer. Defaults to 5

- `WRAPYFI_ROS2_QUEUE_SIZE`: Size of the queue buffer. Defaults to 5

CHAPTER

NINE

PLUGINS

The **NativeObject** message type supports structures beyond native python objects. Wrapyfi already supports a number of non-native objects including numpy arrays and tensors. Wrapyfi can be extended to support objects by using the plugin API. All currently supported plugins by Wrapyfi can be found in the *plugins directory*. Plugins can be added by:

- Creating a derived class that inherits from the base class `wrapyfi.utils.Plugin`
- Overriding the `encode` method for converting the object to a `json` serializable string. Deserializing the string is performed within the overridden `decode` method
- Specifying custom object properties by defining keyword arguments for the class constructor. These properties can be passed directly to the Wrapyfi decorator
- Decorating the class with `@PluginRegistrar.register` and appending the plugin to the list of supported objects
- Appending the script path where the class is defined to the `WRAPYFI_PLUGINS_PATH` environment variable
- Ensure that the plugin resides within a directory named `plugins` nested inside the `WRAPYFI_PLUGINS_PATH` and that the directory contains an `__init__.py` file

9.1 Plugin Example

An example for adding a plugin for a custom Astropy object is provided in the astropy_example.py example. In the example, we append the example's directory to the `WRAPYFI_PLUGINS_PATH` environment variable and import the plugin. The plugin (astropy_tables.py) in the plugins directory is then used to encode and decode the custom object (from within the `examples/encoders/` directory):

```
# create the publisher with default middleware (changed with --mware). The plugin is␣
↪automatically loaded
python3 astropy_example.py --mode publish
# create the listener with default middleware (changed with --mware). The plugin is␣
↪automatically loaded
python3 astropy_example.py --mode listen
```

from the two terminal outputs, the same object should be printed after typing a random message and pressing enter:

```
Method result: [{'message': 'hello world', 'astropy_table': <Table length=3>
  name     flux
 bytes8  float64
-------- -------
source 1     1.2
```

(continues on next page)

(continued from previous page)

```
source 2     2.2
source 3     3.1, 'list': [1, 2, 3]}, 'string', 0.4344, {'other': (1, 2, 3, 4.32)}]
```

Warning: Due to differences in versions, the decoding may result in inconsitent outcomes, which must be handled for all versions e.g., MXNet plugin differences are handled in the existing plugin.

9.2 Data Structure Types

Other than native python objects, the following objects are supported:

- `numpy.ndarray` and `numpy.generic`
- `pandas.DataFrame` and `pandas.Series` with pandas v1 (*NumPy* only) and v2 (*PyArrow* and *NumPy* supported)
- `torch.Tensor`
- `tensorflow.Tensor` and `tensorflow.EagerTensor`
- `mxnet.nd.NDArray`
- `jax.numpy.DeviceArray`
- `trax.ArrayImpl` -> `jaxlib.xla_extension.ArrayImpl`
- `paddle.Tensor`
- `PIL.Image`
- `pyarrow.StructArray`
- `xarray.DataArray` and `xarray.Dataset`
- `cupy.ndarray`
- `dask.array.Array` and `dask.dataframe.DataFrame`
- `zarr.core.Array` and `zarr.core.Group`
- `pint.Quantity`

9.3 Device Mapping for Tensors

To map tensor listener decoders to specific devices (CPUs/GPUs), add an argument to tensor data structures with direct GPU/TPU mapping to support re-mapping on mirrored nodes e.g.,

```
@PluginRegistrar.register
class MXNetTensor(Plugin):
    def __init__(self, load_mxnet_device=None, map_mxnet_devices=None, **kwargs):
```

where `map_mxnet_devices` should be `{'default': mxnet.gpu(0)}` when `load_mxnet_device=mxnet.gpu(0)` and `map_mxnet_devices=None`. For instance, when `load_mxnet_device=mxnet.gpu(0)` or `load_mxnet_device="cuda:0"`, `map_mxnet_devices` can be set manually as a dictionary representing the source device as key and the target device as value for non-default device maps.

Suppose we have the following Wrapyfied method:

```
    @MiddlewareCommunicator.register("NativeObject", args.mware, "Notify", "/notify/test_
↪native_exchange",
                                     carrier="tcp", should_wait=True, load_mxnet_
↪device=mxnet.cpu(0),
                                     map_mxnet_devices={"cuda:0": "cuda:1",
                                                        mxnet.gpu(1): "cuda:0",
                                                        "cuda:3": "cpu:0",
                                                        mxnet.gpu(2):  mxnet.gpu(0)})
    def exchange_object(self):
        msg = input("Type your message: ")
        ret = {"message": msg,
               "mx_ones": mxnet.nd.ones((2, 4)),
               "mxnet_zeros_cuda1": mxnet.nd.zeros((2, 3), ctx=mxnet.gpu(1)),
               "mxnet_zeros_cuda0": mxnet.nd.zeros((2, 3), ctx=mxnet.gpu(0)),
               "mxnet_zeros_cuda2": mxnet.nd.zeros((2, 3), ctx=mxnet.gpu(2)),
               "mxnet_zeros_cuda3": mxnet.nd.zeros((2, 3), ctx=mxnet.gpu(3))}
        return ret,
```

then the source and target gpus 1 & 0 would be flipped, gpu 3 would be placed on cpu 0, and gpu 2 would be placed on gpu 0. Defining `mxnet.gpu(1): mxnet.gpu(0)` and `cuda:1`: `cuda:2` in the same mapping should raise an error since the same device is mapped to two different targets.

The plugins supporting remapping are:

- `mxnet.nd.NDArray`
- `torch.Tensor`
- `paddle.Tensor`
- `cupy.ndarray` **ONLY SUPPORTS CUDA DEVICES**

9.4 Serialization

Warning: When encoding dictionaries, `json` supports string keys only and converts any instances of `int` keys to string, causing a difference between the publisher and subscriber returns. It is best to avoid using `int` keys, otherwise handle the difference on the receiving end.

Wrapyfi currently supports JSON as the only serializer. This introduces a number of limitations (beyond serializing native python objects only by default), including:

- dictionary keys cannot be integers. Integers are automatically converted to strings
- Tuples are converted to lists. Sets are not serializable. Tuples and sets are encoded as strings and restored on listening, which resolves this limitation but adds to the encoding overhead. This conversion is supported in Wrapyfi

CHAPTER

TEN

COMPILING ROS 2 INTERFACES

WARNING: These instructions are located in https://github.com/modular-ml/wrapyfi_ros2_interfaces

To run the Wrapyfi ROS 2 services and transmit audio messages, you need to compile the ROS 2 interfaces. ROS 2 must already be installed on your system, with all its build dependencies. You can find the installation instructions here or install using Robostack.

10.1 Prerequisites

- ROS 2 Galactic/Humble
- Python 3.6

10.2 Compiling

1. Copy the `wrapyfi_ros2_interfaces` folder to your ROS 2 workspace (assumed to be `~/ros2_ws`).

```
# from the current directory
cd ../
cp -r wrapyfi_ros2_interfaces ~/ros2_ws/src
```

2. Compile the ROS 2 interfaces:

```
cd ~/ros2_ws
colcon build --packages-select wrapyfi_ros2_interfaces
```

   Note: If the wrong version of Python is used, the compilation will fail. Make sure that the correct version of cmake is used by modifying the `cmake_minimum_required` version in the `~/ros2_ws/src/wrapyfi_ros2_interfaces/CMakeLists.txt` file:

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.5)
# ...
```

   Replacing VERSION 3.5 with the correct version of cmake.

3. Source the ROS 2 workspace:

```
source ~/ros2_ws/install/setup.bash
```

4. Verify that the ROS 2 Native object service interface is compiled:

```
ros2 interface show wrapyfi_ros2_interfaces/srv/ROS2NativeObjectService
```

Which should output:

```
string request
---
string response
```

5. Verify that the ROS 2 Image service interface is compiled:

```
ros2 interface show wrapyfi_ros2_interfaces/srv/ROS2ImageService
```

Which should output:

```
string request
---
sensor_msgs/Image response
    std_msgs/Header header
        builtin_interfaces/Time stamp
            int32 sec
            uint32 nanosec
        string frame_id
                                # Header frame_id should be optical frame of camera
                                # origin of frame should be optical center of␣
↪cameara
                                # +x should point to the right in the image
                                # +y should point down in the image
                                # +z should point into to plane of the image
                                # If the frame_id here and the frame_id of the␣
↪CameraInfo
                                # message associated with the image conflict
                                # the behavior is undefined
    uint32 height
    uint32 width
    string encoding
                           # taken from the list of strings in include/sensor_msgs/
↪image_encodings.hpp
    uint8 is_bigendian
    uint32 step
    uint8[] data
```

6. Verify that the ROS 2 Audio service interface is compiled:

```
ros2 interface show wrapyfi_ros2_interfaces/srv/ROS2AudioService
```

Which should output:

```
string request
---
wrapyfi_ros2_interfaces/ROS2AudioMessage response
   std_msgs/Header header
       builtin_interfaces/Time stamp
           int32 sec
           uint32 nanosec
```

(continues on next page)

(continued from previous page)

```
    string frame_id
uint32 chunk_size
uint8 channels
uint32 sample_rate
string encoding
uint8 is_bigendian
uint32 bitrate
string coding_format
uint32 step
uint8[] data
```

Run your Wrapyfi enabled script from the same terminal. Now you can use the REQ/REP pattern (server/client) in Wrapyfi [example], and transmit ROS 2 audio messages [example].

CHAPTER

ELEVEN

COMPILING ROS INTERFACES

WARNING: These instructions are located in https://github.com/modular-ml/wrapyfi_ros_interfaces

To transmit ROS audio messages with Wrapyfi, you need to compile the ROS interfaces. ROS must already be installed on your system, with all its build dependencies. You can find the installation instructions here or install using Robostack.

11.1 Prerequisites

- ROS Noetic
- Python 3.6

11.2 Compiling

1. Copy the `wrapyfi_ros_interfaces` folder to your ROS workspace (assumed to be `~/ros_ws`).

```
# from the current directory
cd ../
cp -r wrapyfi_ros_interfaces ~/ros_ws/src
```

2. Compile the ROS interfaces:

```
cd ~/ros_ws
catkin_make
```

   Note: If the wrong version of Python is used, the compilation will fail. Make sure that the correct version of cmake is used by modifying the `cmake_minimum_required` version in the `~/ros_ws/src/wrapyfi_ros_interfaces/CMakeLists.txt` file:

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.0.2)
# ...
```

   Replacing VERSION 3.0.2 with the correct version of cmake.

3. Source the ROS workspace:

```
source ~/ros_ws/devel/setup.bash
```

4. Verify that the ROS Audio message interface is compiled:

```
rosmsg show ROSAudioMessage
```

Which should output:

```
[wrapyfi_ros_interfaces/ROSAudioMessage]:
std_msgs/Header header
 uint32 seq
 time stamp
 string frame_id
uint32 chunk_size
uint8 channels
uint32 sample_rate
string encoding
uint8 is_bigendian
uint32 bitrate
string coding_format
uint32 step
uint8[] data
```

5. Verify that the ROS Audio service interface is compiled:

```
rossrv show ROSAudioService
```

Which should output:

```
[wrapyfi_ros_interfaces/ROSAudioService]:
string request
---
wrapyfi_ros_interfaces/ROSAudioMessage response
 std_msgs/Header header
   uint32 seq
   time stamp
   string frame_id
 uint32 chunk_size
 uint8 channels
 uint32 sample_rate
 string encoding
 uint8 is_bigendian
 uint32 bitrate
 string coding_format
 uint32 step
 uint8[] data
```

Run your Wrapyfi enabled script from the same terminal. Now you can transmit ROS audio messages in PUB/SUB [example] and REQ/REP [example].

CHAPTER

TWELVE

YARP + ICUB SETUP AND INSTRUCTIONS

To run the iCub simulator, the YARP framework as well as the iCub simulator need to be installed. Before cloning YARP and iCub, checkout the compatible versions. For this guide, we install YARP v3.3.2 and iCub (icub-main) v1.16.0.

12.1 Installing YARP

Tip: For installing YARP on **Windows** follow these instructions or install it within a conda or conda-like (mamba/micromamba) environment

Several instructional tutorials are available for installing YARP and its python bindings. The installation process described here targets Ubuntu 20.04 and Python 3.8. Download the YARP source and install following the official installation documentation (do not use the precompiled binaries). Set the environment variable `YARP_ROOT` to the YARP source location:

```
export YARP_ROOT=<YOUR YARP LOCATION>
```

Once YARP has been successfully installed, we proceed by installing the python bindings

1. Go to the YARP directory `cd $YARP_ROOT/bindings`
2. Create a build directory `mkdir build`
3. Compile the Python bindings (Assuming Python3.8 is installed):

```
cmake -DYARP_COMPILE_BINDINGS:BOOL=ON -DCREATE_PYTHON:BOOL=ON -DYARP_USE_PYTHON_
↪VERSION=3.8 -DPYTHON_INCLUDE_DIR=/usr/include/python3.8 -DPYTHON_LIBRARY=/usr/lib/
↪x86_64-linux-gnu/libpython3.8.so -DCMAKE_INSTALL_PYTHONDIR=lib/python3.8/dist-
↪packages -DPYTHON_EXECUTABLE=/usr/bin/python3.8 ..
```

   NOTE: Specify the following env. vars if Python3 libraries cannot be found (make sure that the locations exist):

```
export PYTHON_INCLUDE_DIR=/usr/include/python3.8
export PYTHON_LIBRARY=/usr/lib/x86_64-linux-gnu/libpython3.8.so
```

4. Install the Python bindings

```
make
sudo make install
```

5. Per official instructions, the `PYTHONPATH` and `LD_LIBRARY_PATH` should be set to the following (assuming you are in the build directory):

```
export PYTHONPATH=$PWD:$PYTHONPATH
setenv LD_LIBRARY_PATH $PWD:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH
```

however, setting `PYTHONPATH` to the directory where the `.so` file exists only seemed to work:

```
export PYTHONPATH=$YARP_ROOT/bindings/build/lib/python:$PYTHONPATH
```

You can add the `export PYTHON...` command to your `~/.bashrc` to avoid setting it again for a new session.

12.2 Installing iCub

Install from the source by following the official installation documentation.

12.3 Running the Interface (in Simulation)

1. Start the YARP server from the console

   ```
   yarpserver
   ```

2. Start the iCub simulator in another console

   ```
   iCub_SIM
   ```

3. To issue instructions for controlling the iCub's head from this Python-based interface, simply start the following script from this repository's root directory:

   ```
   python3 examples/robots/icub_head.py \
   --simulation --get_cam_feed --control_head \
   --set_head_eye_coordinates \
   --head_eye_coordinates_port "/control_interface/head_eye_coordinates" \
   --control_expressions --set_facial_expressions \
   --facial_expressions_port "/emotion_interface/facial_expression"
   ```

   You can also manually control the joints by using the YARP's built-in motor controller GUI

   ```
   yarpmotorgui
   ```

   This interface uses YARP for controlling the iCub, however, communicating control from other devices is limited to YARP. The default communcation (set to yarp) middleware can be overridden through environment variables set before running the `interface.py` file e.g.:

   ```
   export ICUB_DEFAULT_MWARE=ros
   ```

   or passing `--mware` argument to the `interface.py` file e.g.:

   ```
   python3 yaw/robots/icub_head/interface.py --mware ros ...
   ```

12.4 Facial Expressions in Simulation

The facial expressions do not run out-of-the-box. For compatibility with the `icub_head.py`, the following scripts need to be executed after starting the `iCub_SIM`:

1. Start the simulator face expression interface:

   `simFaceExpressions`

2. Start the emotion interface:

   `emotionInterface --name /icubSim/face/emotions --context faceExpressions --from emotions.ini`

3. Connect the emotion interface ports:

```
yarp connect /face/eyelids /icubSim/face/eyelids
yarp connect /face/image/out /icubSim/texture/face
yarp connect /icubSim/face/emotions/out /icubSim/face/raw/in
```

12.5 Running YARP on multiple machines

1. Run the YARP server on the host machine:

   `yarpserver --ip <YOUR NETWORK IP> --socket 10000`

2. Detect the YARP server on the client machine(s):

```
yarp conf <YOUR NETWORK IP> 10000
# alternatively
# yarp detect --write
```

CHAPTER

THIRTEEN

BASIC EXAMPLES

13.1 Hello World

This example shows how to use the MiddlewareCommunicator to send and receive messages. It can be used to test the functionality of the middleware using the PUB/SUB pattern and the REQ/REP pattern. The example can be run on a single machine or on multiple machines. In this example (as with all other examples), the communication middleware is selected using the `--mware` argument. The default is ZeroMQ, but YARP, ROS, and ROS 2 are also supported.

CHAPTER

FOURTEEN

COMMUNICATION SCHEMES

14.1 Mirroring

This script demonstrates the capability to mirror messages using the MiddlewareCommunicator within the Wrapyfi library. The communication follows the PUB/SUB and REQ/REP patterns, allowing message publishing, listening, requesting, and replying functionalities between processes or machines.

14.2 Forwarding

This script demonstrates message forwarding using the MiddlewareCommunicator within the Wrapyfi library. The communication follows chained forwarding through two methods, enabling PUB/SUB pattern that allows message publishing and listening functionalities between processes or machines.

14.3 Channeling

This script demonstrates message channeling through three different middleware (A, B, and C) using the MiddlewareCommunicator within the Wrapyfi library. It allows message publishing and listening functionalities between processes or machines.

CHAPTER

FIFTEEN

COMMUNICATION PATTERNS

15.1 REQ/REP

This script demonstrates the capability to request and reply to messages using the MiddlewareCommunicator within the Wrapyfi library. The communication follows the REQ/REP pattern, allowing message requesting and replying functionalities between processes or machines.

CHAPTER

SIXTEEN

CUSTOM MESSAGES

16.1 ROS Message

This script demonstrates the capability to transmit ROS messages, specifically `geometry_msgs/Pose` and `std_msgs/String`, using the MiddlewareCommunicator within the Wrapyfi library. The communication follows the PUB/SUB pattern allowing message publishing and listening functionalities between processes or machines.

16.2 ROS Parameter

This script demonstrates the capability to transmit ROS properties, specifically using the Properties message, using the MiddlewareCommunicator within the Wrapyfi library. The communication follows the PUB/SUB pattern allowing property publishing and listening functionalities between processes or machines.

16.3 ROS2 Message

This script demonstrates the capability to transmit ROS 2 messages, specifically `geometry_msgs/Pose` and `std_msgs/String`, using the MiddlewareCommunicator within the Wrapyfi library. The communication follows the PUB/SUB pattern allowing message publishing and listening functionalities between processes or machines.

CHAPTER

SEVENTEEN

PLUGINS (ENCODERS)

17.1 Astropy

A message publisher and listener for native Python objects and Astropy Tables (external plugin).

17.2 Pint

A message publisher and listener for native Python objects and Pint Quantities.

17.3 Pillow

A message publisher and listener for PIL (Pillow) images.

17.4 DASK

A message publisher and listener for native Python objects and Dask Arrays/Dataframes.

17.5 NumPy and pandas

A message publisher and listener for native Python objects, NumPy Arrays, and pandas Series/Dataframes.

17.6 PyArrow

A message publisher and listener for native Python objects and PyArrow arrays.

17.7 xarray

A message publisher and listener for native Python objects and xarray DataArrays.

17.8 CuPy

A message publisher and listener for native Python objects and CuPy arrays.

17.9 Zarr

A message publisher and listener for native Python objects and Zarr arrays/groups.

17.10 JAX

A message publisher and listener for native Python objects and JAX tensors.

17.11 Trax

A message publisher and listener for native Python objects and Trax arrays.

17.12 MXNet

A message publisher and listener for native Python objects and MXNet tensors.

17.13 PaddlePaddle

A message publisher and listener for native Python objects and PaddlePaddle tensors.

17.14 PyTorch

A message publisher and listener for native Python objects and PyTorch tensors.

17.15 TensorFlow

A message publisher and listener for native Python objects and TensorFlow tensors.

CHAPTER

EIGHTEEN

ROBOTS

18.1 iCub Head

This script demonstrates the capability to control the iCub robot's head and view its camera feed using the MiddlewareCommunicator within the Wrapyfi library. The communication follows the PUB/SUB pattern, allowing for message publishing and listening functionalities between processes or machines.

CHAPTER

NINETEEN

SENSORS

19.1 Camera and microphone

This script demonstrates the capability to transmit audio and video streams using the MiddlewareCommunicator within the Wrapyfi library. The communication follows the PUB/SUB pattern allowing message publishing and listening functionalities between processes or machines.

CHAPTER

TWENTY

ROBOTICS

20.1 Multiple Robot Control using the Mirroring and Forwarding Schemes

This tutorial demonstrates how to use the Wrapyfi framework to run a facial expression recognition (FER) model on multiple robots. The model recognizes 8 facial expressions which are propagated to the Pepper and iCub robots. The expression categories are displayed by changing the Pepper robot's eye and shoulder LED colors—or *robotic facial expressions*—by changing the iCub robot's eyebrow and mouth LED patterns.

20.2 Switching between Sensors using the Mirroring and Channeling Schemes

This tutorial demonstrates how to use the Wrapyfi framework to run a head pose estimation model and/or acquire head orientation from inertial measurement unit (IMU) readings to mirror the movements of an actor on the iCub robot in a near-real-time setting. Under the model-controlled condition, the iCub robot's movements are actuated by a vision-based head pose estimation model. Under the IMU-controlled condition, the orientation readings arrived instead from an IMU attached to a wearable eye tracker.

CHAPTER

TWENTYONE

NEURAL NETWORKS

21.1 Horizontal and Vertical Layer Sharding

This tutorial demonstrates how to use the Wrapyfi to shard a neural network across multiple machines. We provide two examples: (1) horizontal sharding of the facial expression recognition model, and (2) vertical sharding of the Llama LLM model.

CHAPTER

TWENTYTWO

LATENCY: PLUGIN ENCODING AND DECODING

We measure the transmission latency over multiple trials to assess the effectiveness of Wrapyfi in supporting different frameworks and libraries. The results shown in ***Figure 1*** do not reflect the performances of the middleware or libraries themselves but rather those of our serialization and deserialization mechanisms within the given environment. The evaluation was conducted in publishing and subscribing modes on the same machine with an Intel Core i9-11900 running at 2.5 GHz, with 64 GB RAM and an NVIDIA GeForce RTX 3080 Ti GPU with 12 GB VRAM.

We observe NumPy array transmission to result in the lowest latency compared to other data types. This is due to the fact that most plugin encoders are implemented using NumPy arrays. Variances in performance appear most significant with the ROS middleware, which also results in the highest latency on average. The ROS Python bindings serialize messages natively, resulting in additional overhead. GPU tensor mapping to memory shows insignificant delay compared to memory-mapped counterparts in the case of MXNet, PyTorch, and PaddlePaddle. pandas data frames are transmitted with the highest latency owing to their large memory

Warning: Tests conducted using pandas version 1 with a NumPy backend

Compared to NumPy, pandas provides a wealth of tools for statistical analysis and data filtration, making it the better option when data management is prioritized. In terms of optimality, NumPy uses C arrays compared to pandas native Python objects, giving NumPy a significant boost in encoding and decoding performance.

22.1 Running the Benchmarks

The benchmarks are executed using the benchmarking_native_object.py script. The script is executed once as a **publisher** and once as a **listener**, running simultaneously . The logs for serialization and deserialization are saved in the `results` directory (within the working directory). With the benchmarking script, we can specify the plugins, the middleware to use, and the shape of the array/tensor to transmit. The script also allows us to specify the number of trials to conduct and the publishing rate. The script can be executed as follows:

```
python benchmarking_native_object.py --listen --plugins pillow numpy --mwares ros yarp --
↪width 200 --height 200 --trials 2000
```

```
python benchmarking_native_object.py --publish --plugins pillow numpy --mwares ros yarp -
↪-width 200 --height 200 --trials 2000
```

Warning: ROS and ROS 2 cannot run within the same environment. Therefore, the benchmarks must be executed in separate environments.

The stored files can then be plotted using the jupyter notebook. Make sure the loaded log files from the `results` directory are changed according to the latest benchmark runs.

CHAPTER

TWENTYTHREE

WRAPYFI

23.1 wrapyfi package

23.1.1 Subpackages

wrapyfi.clients package

Submodules

wrapyfi.clients.ros module

class wrapyfi.clients.ros.**ROSClient**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *ros_kwargs: dict | None = None*, ***kwargs*)

Bases: *Client*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *ros_kwargs: dict | None = None*, ***kwargs*)

Initialize the client.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for rep/req pattern. Default is 'tcp'
- **ros_kwargs** – dict: Additional kwargs for the ROS middleware
- **kwargs** – dict: Additional kwargs for the client

close()

Close the client.

class wrapyfi.clients.ros.**ROSNativeObjectClient**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *persistent: bool = True*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROSClient*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *persistent: bool = True*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

The NativeObject client using the ROS String message assuming the data is serialized as a JSON string. Deserializes the data (including plugins) using the decoder and parses it to a Python object.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for rep/req pattern. Default is 'tcp'
- **persistent** – bool: Whether to keep the service connection alive across multiple service calls. Default is True
- **serializer_kwargs** – dict: Additional kwargs for the serializer
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the client's connection to the ROS service.

request(**args*, ***kwargs*) → Any

Send a request to the ROS service.

Parameters

- **args** – tuple: Positional arguments to send in the request
- **kwargs** – dict: Keyword arguments to send in the request

Returns

Any: The response from the ROS service

class wrapyfi.clients.ros.**ROSImageClient**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *width: int = -1*, *height: int = -1*, *persistent: bool = True*, *rgb: bool = True*, *fp: bool = False*, *serializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROSClient*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *width: int = -1*, *height: int = -1*, *persistent: bool = True*, *rgb: bool = True*, *fp: bool = False*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The Image client using the ROS Image message parsed to a numpy array.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for rep/req pattern. Default is 'tcp'
- **width** – int: The width of the image. Default is -1
- **height** – int: The height of the image. Default is -1
- **rgb** – bool: Whether the image is RGB. Default is True
- **fp** – bool: Whether to utilize floating-point precision. Default is False
- **persistent** – bool: Whether to keep the service connection alive across multiple service calls. Default is True
- **serializer_kwargs** – dict: Additional kwargs for the serializer

establish()

Establish the client's connection to the ROS service.

request(**args*, ***kwargs*)

Send a request to the ROS service.

Parameters

- **args** – tuple: Positional arguments to send in the request
- **kwargs** – dict: Keyword arguments to send in the request

Returns

np.array: The received image from the ROS service

class wrapyfi.clients.ros.**ROSAudioChunkClient**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *persistent: bool = True*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *serializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROSClient*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *persistent: bool = True*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The AudioChunk client using the ROS Audio message parsed to a numpy array.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for rep/req pattern. Default is 'tcp'
- **channels** – int: Number of audio channels. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: The size of audio chunks. Default is -1
- **persistent** – bool: Whether to keep the service connection alive across multiple service calls. Default is True
- **serializer_kwargs** – dict: Additional kwargs for the serializer

establish()

Establish the client's connection to the ROS service.

request(**args*, ***kwargs*)

Send a request to the ROS service

Parameters

- **args** – tuple: Positional arguments to send in the request
- **kwargs** – dict: Keyword arguments to send in the request

Returns

Tuple[np.array, int]: The received audio chunk and rate from the ROS service

wrapyfi.clients.ros2 module

class wrapyfi.clients.ros2.**ROS2Client**(**args: Any*, ***kwargs: Any*)

Bases: *Client*, Node

__init__(*name: str*, *in_topic: str*, *ros2_kwargs: dict | None = None*, ***kwargs*)

Initialize the client.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **ros2_kwargs** – dict: Additional kwargs for the ROS 2 middleware
- **kwargs** – dict: Additional kwargs for the client

close()

Close the client.

class wrapyfi.clients.ros2.**ROS2NativeObjectClient**(**args: Any*, ***kwargs: Any*)

Bases: *ROS2Client*

__init__(*name: str*, *in_topic: str*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

The NativeObject listener using the ROS 2 String message assuming the data is serialized as a JSON string. Deserializes the data (including plugins) using the decoder and parses it to a Python object.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **serializer_kwargs** – dict: Additional kwargs for the serializer
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the client's connection to the ROS 2 service.

request(**args*, ***kwargs*)

Send a request to the ROS 2 service.

Parameters

- **args** – tuple: Positional arguments to send in the request
- **kwargs** – dict: Keyword arguments to send in the request

Returns

Any: The response from the ROS 2 service

class wrapyfi.clients.ros2.**ROS2ImageClient**(**args: Any*, ***kwargs: Any*)

Bases: *ROS2Client*

__init__(*name: str*, *in_topic: str*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The Image client using the ROS 2 Image message parsed to a numpy array.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **width** – int: The width of the image. Default is -1 (use the width of the received image)
- **height** – int: The height of the image. Default is -1 (use the height of the received image)
- **rgb** – bool: Whether the image is RGB. Default is True
- **fp** – bool: Whether to utilize floating-point precision. Default is False
- **jpg** – bool: True if the image should be decompressed from JPG. Default is False
- **serializer_kwargs** – dict: Additional kwargs for the serializer

establish()

Establish the client's connection to the ROS 2 service.

request(**args*, ***kwargs*)

Send a request to the ROS 2 service.

Parameters

- **args** – tuple: Positional arguments to send in the request
- **kwargs** – dict: Keyword arguments to send in the request

Returns

Any: The response from the ROS 2 service

class wrapyfi.clients.ros2.**ROS2AudioChunkClient**(**args: Any*, ***kwargs: Any*)

Bases: *ROS2Client*

__init__(*name: str*, *in_topic: str*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The AudioChunk client using the ROS 2 Audio message parsed to a numpy array.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **channels** – int: Number of channels in the audio. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: Number of samples in the audio chunk. Default is -1 (use the chunk size of the received audio)
- **serializer_kwargs** – dict: Additional kwargs for the serializer

establish()

Establish the client's connection to the ROS 2 service.

request(**args*, ***kwargs*)

Send a request to the ROS 2 service.

Parameters

- **args** – tuple: Positional arguments to send in the request
- **kwargs** – dict: Keyword arguments to send in the request

Returns

Any: The response from the ROS 2 service

wrapyfi.clients.yarp module

class wrapyfi.clients.yarp.**YarpClient**(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *persistent: bool = True*, *yarp_kwargs: dict | None = None*, ***kwargs*)

Bases: *Client*

__init__(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *persistent: bool = True*, *yarp_kwargs: dict | None = None*, ***kwargs*)

Initialize the client.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **persistent** – bool: Whether to keep the service connection alive across multiple service calls. Default is True
- **yarp_kwargs** – dict: Additional kwargs for the Yarp middleware
- **kwargs** – dict: Additional kwargs for the client

close()

Close the client.

class wrapyfi.clients.yarp.**YarpNativeObjectClient**(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *persistent: bool = True*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *YarpClient*

__init__(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *persistent: bool = True*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

The NativeObject listener using the YARP Bottle construct assuming the data is serialized as a JSON string. Deserializes the data (including plugins) using the decoder and parses it to a Python object.

Parameters

- **name** – str: Name of the client
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **serializer_kwargs** – dict: Additional kwargs for the serializer
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer
- **persistent** – bool: Whether to keep the service connection alive across multiple service calls. Default is True
- **serializer_kwargs** – dict: Additional kwargs for the serializer
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer
- **kwargs** – dict: Additional kwargs for the client

establish()

Establish the client's connection to the YARP service.

request(**args*, ***kwargs*)

Send a request to the YARP service.

Parameters

- **args** – tuple: Positional arguments to send in the request
- **kwargs** – dict: Keyword arguments to send in the request

Returns

Any: The response from the YARP service

class wrapyfi.clients.yarp.**YarpImageClient**(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *persistent: bool = True*, *serializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *YarpNativeObjectClient*

__init__(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *persistent: bool = True*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The Image client using the YARP Bottle construct parsed to a numpy array.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **width** – int: The width of the image. Default is -1
- **height** – int: The height of the image. Default is -1
- **rgb** – bool: Whether the image is RGB. Default is True
- **fp** – bool: Whether to utilize floating-point precision. Default is False
- **persistent** – bool: Whether to keep the service connection alive across multiple service calls. Default is True
- **serializer_kwargs** – dict: Additional kwargs for the serializer

class wrapyfi.clients.yarp.**YarpAudioChunkClient**(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *persistent: bool = True*, *serializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *YarpNativeObjectClient*

__init__(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *persistent: bool = True*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The AudioChunk client using the YARP Bottle construct parsed to a numpy array.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'

- **channels** – int: Number of audio channels. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: The size of audio chunks. Default is -1
- **persistent** – bool: Whether to keep the service connection alive across multiple service calls. Default is True
- **serializer_kwargs** – dict: Additional kwargs for the serializer

wrapyfi.clients.zeromq module

class wrapyfi.clients.zeromq.**ZeroMQClient**(*name*, *in_topic*, *carrier='tcp'*, *socket_ip: str = '127.0.0.1'*, *socket_rep_port: int = 5558*, *zeromq_kwargs: dict | None = None*, ***kwargs*)

Bases: *Client*

__init__(*name*, *in_topic*, *carrier='tcp'*, *socket_ip: str = '127.0.0.1'*, *socket_rep_port: int = 5558*, *zeromq_kwargs: dict | None = None*, ***kwargs*)

Initialize the client.

Parameters

- **name** – str: Name of the client
- **out_topic** – str: Topics are not supported for the REQ/REP pattern in ZeroMQ. Any given topic is ignored
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **zeromq_kwargs** – dict: Additional kwargs for the ZeroMQ middleware
- **kwargs** – dict: Additional kwargs for the client

close()

Close the subscriber.

class wrapyfi.clients.zeromq.**ZeroMQNativeObjectClient**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ZeroMQClient*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific client for handling native Python objects, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Topics are not supported for the REQ/REP pattern in ZeroMQ. Any given topic is ignored
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for REQ/REP pattern. Default is 'tcp'
- **serializer_kwargs** – dict: Additional kwargs for the serializer

- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish(***kwargs*)

Establish the connection to the server.

request(**args*, ***kwargs*)

Serialize the provided Python objects to JSON strings, send a request to the server, and await a reply. The method uses the configured JSON encoder for serialization before sending the resultant string to the server.

Parameters

- **args** – tuple: Arguments to be sent to the server
- **kwargs** – dict: Keyword arguments to be sent to the server

Returns

Any: The Python object received from the server, deserialized using the configured JSON decoder hook

class wrapyfi.clients.zeromq.**ZeroMQImageClient**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, *serializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ZeroMQNativeObjectClient*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The Image client using the ZeroMQ message construct parsed to a numpy array.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Topics are not supported for the REQ/REP pattern in ZeroMQ. Any given topic is ignored
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **width** – int: Width of the image. Default is -1 (use the width of the received image)
- **height** – int: Height of the image. Default is -1 (use the height of the received image)
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be compressed to JPG before sending. Default is False
- **serializer_kwargs** – dict: Additional kwargs for the serializer

class wrapyfi.clients.zeromq.**ZeroMQAudioChunkClient**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *serializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ZeroMQNativeObjectClient*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The AudioChunk client using the ZeroMQ message construct parsed to a numpy array.

Parameters

- **name** – str: Name of the client
- **in_topic** – str: Topics are not supported for the REQ/REP pattern in ZeroMQ. Any given topic is ignored
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **width** – int: Width of the image. Default is -1 (use the width of the received image)
- **height** – int: Height of the image. Default is -1 (use the height of the received image)
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be compressed to JPG before sending. Default is False
- **serializer_kwargs** – dict: Additional kwargs for the serializer

Module contents

class wrapyfi.clients.**FallbackClient**(*name: str*, *in_topic: str*, *carrier: str = ''*, *missing_middleware_object: str = ''*, ***kwargs*)

Bases: *Client*

__init__(*name: str*, *in_topic: str*, *carrier: str = ''*, *missing_middleware_object: str = ''*, ***kwargs*)

Initialize the client.

Parameters

- **name** – str: The name of the client
- **in_topic** – str: The topic to listen to
- **carrier** – str: The middleware carrier to use

establish(*repeats: int = -1*, ***kwargs*)

Establish the client.

request(**args*, ***kwargs*)

Send a request to the server.

close()

Close the connection.

wrapyfi.config package

Submodules

wrapyfi.config.manager module

class wrapyfi.config.manager.**ConfigManager**(**args*, ***kwargs*)

Bases: object

The configuration manager is a singleton which is invoked once throughout the runtime.

__init__(*config: dict | str | None*, ***kwargs*)

Initializing the ConfigManager. The configuration can be provided as a yaml file name or as a dictionary.

Parameters

config – Union[dict, str]: Loads a yaml configuration file when "str" type provided. Directly set when "dict" type provided

Module contents

wrapyfi.connect package

Submodules

wrapyfi.connect.clients module

class wrapyfi.connect.clients.**Clients**

Bases: `object`

A class that holds all clients and their corresponding middleware communicators.

registry = {'AudioChunk:ros': <class 'wrapyfi.clients.ros.ROSAudioChunkClient'>, 'AudioChunk:ros2': <class 'wrapyfi.clients.ros2.ROS2AudioChunkClient'>, 'AudioChunk:yarp': <class 'wrapyfi.clients.yarp.YarpAudioChunkClient'>, 'AudioChunk:zeromq': <class 'wrapyfi.clients.zeromq.ZeroMQAudioChunkClient'>, 'Image:ros': <class 'wrapyfi.clients.ros.ROSImageClient'>, 'Image:ros2': <class 'wrapyfi.clients.ros2.ROS2ImageClient'>, 'Image:yarp': <class 'wrapyfi.clients.yarp.YarpImageClient'>, 'Image:zeromq': <class 'wrapyfi.clients.zeromq.ZeroMQImageClient'>, 'MMO:fallback': <class 'wrapyfi.clients.FallbackClient'>, 'NativeObject:ros': <class 'wrapyfi.clients.ros.ROSNativeObjectClient'>, 'NativeObject:ros2': <class 'wrapyfi.clients.ros2.ROS2NativeObjectClient'>, 'NativeObject:yarp': <class 'wrapyfi.clients.yarp.YarpNativeObjectClient'>, 'NativeObject:zeromq': <class 'wrapyfi.clients.zeromq.ZeroMQNativeObjectClient'>}

mwares = {'fallback', 'ros', 'ros2', 'yarp', 'zeromq'}

classmethod register(*data_type: str*, *communicator: str*)

Register a client with the given data type and middleware communicator.

Parameters

- **data_type** – str: The data type to register the client for e.g., "NativeObject", "Image", "AudioChunk", etc.
- **communicator** – str: The middleware communicator to register the client for e.g., "ros", "ros2", "yarp", "zeromq", etc.

static scan()

Scan for clients and add them to the registry.

class wrapyfi.connect.clients.**Client**(*name: str*, *in_topic: str*, *carrier: str = ''*, ***kwargs*)

Bases: `object`

A base class for clients.

__init__(*name: str*, *in_topic: str*, *carrier: str = ''*, ***kwargs*)

Initialize the client.

Parameters

- **name** – str: The name of the client
- **in_topic** – str: The topic to listen to
- **carrier** – str: The middleware carrier to use

establish()

Establish the client.

request(**args*, ***kwargs*)

Send a request to the server.

close()

Close the connection.

wrapyfi.connect.listeners module

class wrapyfi.connect.listeners.**ListenerWatchDog**(**args*, ***kwargs*)

Bases: object

A watchdog that scans for listeners and removes them from the ring if they are not established.

__init__(*repeats: int = 10*, *inner_repeats: int = 10*)

Initialize the ListenerWatchDog.

Parameters

repeats – int: The number of times to repeat the scan

param inner_repeats: int: The number of times to repeat the scan for each listener

add_listener(*listener*)

Add a listener to the ring.

Parameters

listener – Listener: The listener to add

remove_listener(*listener*)

Remove a listener from the ring.

Parameters

listener – Listener: The listener to remove

scan()

Scan for listeners and remove them from the ring if they are established.

class wrapyfi.connect.listeners.**Listeners**

Bases: object

A class that holds all listeners and their corresponding middleware communicators.

registry = {}

mwares = {}

classmethod register(*data_type: str*, *communicator: str*)

Register a listener with the given data type and middleware communicator.

Parameters

- **data_type** – str: The data type to register the listener for e.g., "NativeObject", "Image", "AudioChunk", etc.
- **communicator** – str: The middleware communicator to register the listener for e.g., "ros", "ros2", "yarp", "zeromq", etc.

static scan()

Scan for listeners and add them to the registry.

class wrapyfi.connect.listeners.**Listener**(*name: str*, *in_topic: str*, *carrier: str = ''*, *should_wait: bool = True*, ***kwargs*)

Bases: object

A base class for listeners.

__init__(*name: str*, *in_topic: str*, *carrier: str = ''*, *should_wait: bool = True*, ***kwargs*)

Initialize the Listener.

Parameters

- **name** – str: The name of the listener
- **in_topic** – str: The topic to listen to
- **carrier** – str: The middleware carrier to use
- **should_wait** – bool: Whether to wait for the listener to be established or not

check_establishment(*established: bool*)

Check if the listener is established or not.

Parameters
established – bool: Whether the listener is established or not

Returns
bool: Whether the listener is established or not

establish(*repeats: int = -1*, ***kwargs*)

Establish the listener.

listen()

Listen for incoming data.

close()

Close the connection.

wrapyfi.connect.publishers module

class wrapyfi.connect.publishers.**PublisherWatchDog**(**args*, ***kwargs*)

Bases: object

A watchdog that scans for publishers and removes them from the ring if they are not established.

__init__(*repeats: int = 10*, *inner_repeats: int = 10*)

Initialize the PublisherWatchDog.

Parameters
repeats – int: The number of times to repeat the scan

param inner_repeats: int: The number of times to repeat the scan for each publisher

add_publisher(*publisher*)

Add a publisher to the ring.

Parameters
publisher – Publisher: The publisher to add

remove_publisher(*publisher*)

Remove a publisher from the ring.

Parameters
publisher – Publisher: The publisher to remove

scan()

Scan for publishers and remove them from the ring if they are established.

class wrapyfi.connect.publishers.**Publishers**

Bases: object

A class that holds all publishers and their corresponding middleware communicators.

registry = {}

mwares = {}

classmethod register(*data_type: str*, *communicator: str*)

Register a publisher for a given data type and middleware communicator.

Parameters

- **data_type** – str: The data type to register the publisher for e.g., "NativeObject", "Image", "AudioChunk", etc.
- **communicator** – str: The middleware communicator to register the publisher for e.g., "ros", "ros2", "yarp", "zeromq", etc.

Returns
Callable[..., Any]: A decorator function that registers the decorated class as a publisher for the given data type and middleware communicator

static scan()

Scan for publishers and add them to the registry.

class wrapyfi.connect.publishers.**Publisher**(*name: str*, *out_topic: str*, *carrier: str = ''*, *should_wait: bool = True*, ***kwargs*)

Bases: object

A base class for all publishers.

__init__(*name: str*, *out_topic: str*, *carrier: str = ''*, *should_wait: bool = True*, ***kwargs*)

Initialize the Publisher.

Parameters

- **name** – str: The name of the publisher

- **out_topic** – str: The name of the output topic
- **carrier** – str: The name of the carrier to use
- **should_wait** – bool: Whether to wait for the publisher to be established or not

check_establishment(*established: bool*)

Check if the publisher is established and remove it from the ring if it is.

Parameters
established – bool: Whether the publisher is established or not

Returns
bool: Whether the publisher is established or not

establish(*repeats: int = -1*, ***kwargs*)

Establish the publisher.

publish(*obj*)

Publish an object.

close()

Close the connection.

wrapyfi.connect.servers module

class wrapyfi.connect.servers.**Servers**

Bases: object

A class that holds all servers and their corresponding middleware communicators.

registry = {}

mwares = {}

classmethod register(*data_type: str*, *communicator: str*)

Register a server with the given data type and middleware communicator.

Parameters

- **data_type** – str: The data type to register the server for e.g., "NativeObject", "Image", "AudioChunk", etc.
- **communicator** – str: The middleware communicator to register the server for e.g., "ros", "ros2", "yarp", "zeromq", etc.

Returns
Callable: A decorator that registers the server with the given data type and middleware communicator

static scan()

Scan for servers and add them to the registry.

class wrapyfi.connect.servers.**Server**(*name: str*, *out_topic: str*, *carrier: str = ''*, *out_topic_connect: str | None = None*, ***kwargs*)

Bases: object

A base class for servers.

__init__(*name: str*, *out_topic: str*, *carrier: str = ''*, *out_topic_connect: str | None = None*, ***kwargs*)

Initialize the server.

Parameters

- **name** – str: The name of the server
- **out_topic** – str: The topic to publish to
- **carrier** – str: The middleware carrier to use
- **out_topic_connect** – str: The topic to connect to (this is deprecated and will be removed in the future since its usage is limited to YARP)

establish()

Establish the server.

await_request(**args*, ***kwargs*)

Await a request from a client.

reply(*obj*)

Reply to a client request.

close()

Close the connection.

wrapyfi.connect.wrapper module

class wrapyfi.connect.wrapper.**MiddlewareCommunicator**

Bases: object

__init__()

Initialises the middleware communicator.

classmethod **register**(*data_type: str | List[Any]*, *middleware: str = 'zeromq'*, **args*, ***kwargs*)

Registers a function to the middleware communicator, defining its communication message type and associated middleware. Note that the function returned is a wrapper that can alter the behavior of the registered function based on the communication mode set in the middleware communicator registry.

Parameters

- **data_type** – Union[str, List[Any]]: Specifies the communication message type, either as a string or a list containing specifications of the data type and middleware
- **middleware** – str: Specifies the middleware to be used for the communication, defaults to DEFAULT_COMMUNICATOR
- **args** – tuple: Variable positional arguments to be passed to the function
- **kwargs** – dict: Additional keyword arguments to be passed to the function

Returns

Callable[..., Any]: A wrapper function that triggers specific communication modes (e.g., publish, listen, reply, request) based on the registered function and middleware settings

Raises

NotImplementedError: If *data_type* is a dictionary or an unsupported type

activate_communication(*func: str | Callable[[...], Any]*, *mode: str | List[str]*)

Activates the communication mode for a registered function in the middleware communicator. The mode determines how the function will interact with the middleware communicator upon invocation, e.g., whether it should publish its return values, listen for values, etc. The communication mode can be set for all instances of the function or individually per instance.

Parameters

- **func** – Union[str, Callable[..., Any]]: The function or the name of the function for which the communication mode should be activated
- **mode** – Union[str, List[str]]: Specifies the communication mode to be activated, either as a single mode (string) applicable to all instances, or as a list of modes (strings) per instance

Raises

IndexError: If mode is a list and the number of elements does not match the number of instances

Raises

ValueError: If the instance address cannot be found in the registry

close()

Closes this middleware communicator instance.

static get_communicators()

Returns the available middleware communicators.

Returns

Set[str]: The available middleware communicators (excluding the fallback communicator)

classmethod close_all_instances()

Closes all instances of the middleware communicator.

classmethod close_instance(*instance_addr: str | None = None*)

Closes a specific instance of the middleware communicator. Note that the instance address is the hexadecimal representation of the instance's id. If no instance address is provided, all instances will be closed.

Parameters

instance_addr – str: The unique identifier of the instance to be closed, defaults to None

Raises

ValueError: If the instance address cannot be found in the registry

Module contents

wrapyfi.listeners package

Submodules

wrapyfi.listeners.ros module

class wrapyfi.listeners.ros.**ROSListener**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *ros_kwargs: dict | None = None*, ***kwargs*)

Bases: *Listener*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *ros_kwargs: dict | None = None*, ***kwargs*)

Initialize the subscriber.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5
- **ros_kwargs** – dict: Additional kwargs for the ROS middleware
- **kwargs** – dict: Additional kwargs for the subscriber

close()

Close the subscriber

class wrapyfi.listeners.ros.**ROSNativeObjectListener**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROSListener*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

The NativeObject listener using the ROS String message assuming the data is serialized as a JSON string. Deserializes the data (including plugins) using the decoder and parses it to a Python object.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the subscriber.

listen() → Any

Listen for a message.

Returns

Any: The received message as a native python object

class wrapyfi.listeners.ros.**ROSImageListener**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

Bases: *ROSListener*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

The Image listener using the ROS Image message parsed to a numpy array.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5
- **width** – int: Width of the image. Default is -1 (use the width of the received image)
- **height** – int: Height of the image. Default is -1 (use the height of the received image)
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be decompressed from JPG. Default is False

establish()

Establish the subscriber.

listen()

Listen for a message.

Returns

np.ndarray: The received message as a numpy array formatted as a cv2 image np.ndarray[img_height, img_width, channels]

class wrapyfi.listeners.ros.**ROSAudioChunkListener**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

Bases: *ROSListener*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

The AudioChunk listener using the ROS Image message parsed to a numpy array.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'

- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5
- **channels** – int: Number of channels in the audio. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: Number of samples in the audio chunk. Default is -1 (use the chunk size of the received audio)

establish()

Establish the subscriber.

listen()

Listen for a message.

Returns

Tuple[np.ndarray, int]: The received message as a numpy array formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

class wrapyfi.listeners.ros.**ROSPropertiesListener**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, ***kwargs*)

Bases: *ROSListener*

Gets rospy parameters. Behaves differently from other data types by directly acquiring ROS parameters. Note that the listener is not guaranteed to receive the updated signal, since the listener can trigger before property is set. The property decorated method returns accept native python objects (excluding None), but care should be taken when using dictionaries, since they are analogous with node namespaces: http://wiki.ros.org/rospy/Overview/Parameter%20Server

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, ***kwargs*)

The PropertiesListener using the ROS Parameter Server.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for a parameter to be set. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5

await_connection(*in_topic: int | None = None*, *repeats: int | None = None*)

Wait for a parameter to be set.

Parameters

- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **repeats** – int: Number of times to check for the parameter. None for infinite. Default is None

establish(*repeats: int | None = None*, ***kwargs*)

Establish the subscriber.

Parameters

repeats – int: Number of times to check for the parameter. None for infinite. Default is None

listen()

Listen for a message.

Returns

Any: The received message as a native python object

class wrapyfi.listeners.ros.**ROSMessageListener**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, ***kwargs*)

Bases: *ROSListener*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, ***kwargs*)

The ROSMessageListener using the ROS message type inferred from the message type. Supports standard ROS msgs.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for a message to be published. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5

establish()

Establish the subscriber.

listen()

Listen for a message.

Returns

rospy.msg: The received message as a ROS message object

wrapyfi.listeners.ros2 module

class wrapyfi.listeners.ros2.**ROS2Listener**(*name: str*, *in_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *ros2_kwargs: dict | None = None*, ***kwargs*)

Bases: *Listener*, Node

__init__(*name: str*, *in_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *ros2_kwargs: dict | None = None*, ***kwargs*)

Initialize the subscriber.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5
- **ros2_kwargs** – dict: Additional kwargs for the ROS 2 middleware
- **kwargs** – dict: Additional kwargs for the subscriber

close()

Close the subscriber.

class wrapyfi.listeners.ros2.**ROS2NativeObjectListener**(*name: str*, *in_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROS2Listener*

__init__(*name: str*, *in_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

The NativeObject listener using the ROS 2 String message assuming the data is serialized as a JSON string. Deserializes the data (including plugins) using the decoder and parses it to a native object.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the subscriber.

listen()

Listen for a message.

Returns

Any: The received message as a native python object

class wrapyfi.listeners.ros2.**ROS2ImageListener**(*name: str*, *in_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

Bases: *ROS2Listener*

__init__(*name: str*, *in_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

The Image listener using the ROS 2 Image message parsed to a numpy array.

Parameters

- **name** – str: Name of the subscriber

- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5
- **width** – int: Width of the image. Default is -1 (use the width of the received image)
- **height** – int: Height of the image. Default is -1 (use the height of the received image)
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be decompressed from JPG. Default is False

establish()

Establish the subscriber

listen()

Listen for a message.

Returns

np.ndarray: The received message as a numpy array formatted as a cv2 image np.ndarray[img_height, img_width, channels]

class wrapyfi.listeners.ros2.**ROS2AudioChunkListener**(*name: str*, *in_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

Bases: *ROS2Listener*

__init__(*name: str*, *in_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

The AudioChunk listener using the ROS 2 Audio message parsed to a numpy array.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5
- **channels** – int: Number of channels in the audio. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: Number of samples in the audio chunk. Default is -1 (use the chunk size of the received audio)

establish()

Establish the subscriber.

listen()

Listen for a message.

Returns

Tuple[np.ndarray, int]: The received message as a numpy array formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

class wrapyfi.listeners.ros2.**ROS2PropertiesListener**(*name*, *in_topic*, ***kwargs*)

Bases: *ROS2Listener*

__init__(*name*, *in_topic*, ***kwargs*)

Initialize the subscriber.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5
- **ros2_kwargs** – dict: Additional kwargs for the ROS 2 middleware
- **kwargs** – dict: Additional kwargs for the subscriber

class wrapyfi.listeners.ros2.**ROS2MessageListener**(*name: str*, *in_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, ***kwargs*)

Bases: *ROS2Listener*

__init__(*name: str*, *in_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, ***kwargs*)

The ROS2MessageListener using the ROS 2 message type inferred from the message type. Supports standard ROS 2 msgs.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5

get_topic_type(*topic_name*)

Get the type of a specific topic.

Parameters

topic_name – str: Name of the topic to get its type

Returns

str or None: The topic type as a string, or None if the topic does not exist

establish()

Establish the subscriber.

listen()

Listen for a message.

Returns

ROS2Message: The received message as a ROS 2 message object

wrapyfi.listeners.yarp module

class wrapyfi.listeners.yarp.**YarpListener**(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *yarp_kwargs: dict | None = None*, ***kwargs*)

Bases: *Listener*

__init__(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *yarp_kwargs: dict | None = None*, ***kwargs*)

Initialize the subscriber.

Parameters

- **name** – str: Name of the publisher
- **in_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **persistent** – bool: Whether the subscriber port should remain connected after closure. Default is True
- **yarp_kwargs** – dict: Additional kwargs for the Yarp middleware
- **kwargs** – dict: Additional kwargs for the subscriber

await_connection(*in_topic: str | None = None*, *repeats: int | None = None*)

Wait for the publisher to connect to the subscriber.

Parameters

- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **repeats** – int: Number of times to check for the parameter. None for infinite. Default is None

Returns

bool: True if connection established, False otherwise

read_port(*port*)

Read the port.

Parameters

port – yarp.Port: Port to read from

Returns

yarp.Value: Value read from the port

close()

Close the subscriber

class wrapyfi.listeners.yarp.**YarpNativeObjectListener**(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *YarpListener*

__init__(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

The NativeObject listener using the BufferedPortBottle string construct assuming the data is serialized as a JSON string. Deserializes the data (including plugins) using the decoder and parses it to a Python object.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **persistent** – bool: Whether the subscriber port should remain connected after closure. Default is True
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection to the publisher.

Parameters

repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns

bool: True if connection established, False otherwise

listen()

Listen for a message.

Returns

Any: The received message as a native python object

class wrapyfi.listeners.yarp.**YarpImageListener**(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

Bases: *YarpListener*

__init__(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

The Image listener using the BufferedPortImage construct parsed to a numpy array.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **persistent** – bool: Whether the subscriber port should remain connected after closure. Default is True
- **width** – int: Width of the image. Default is -1 (use the width of the received image)
- **height** – int: Height of the image. Default is -1 (use the height of the received image)

- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be decompressed from JPG. Default is False

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection to the publisher.

Parameters
repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

listen()

Listen for a message.

Returns
np.ndarray: The received message as a numpy array formatted as a cv2 image np.ndarray[img_height, img_width, channels]

class wrapyfi.listeners.yarp.**YarpAudioChunkListener**(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

Bases: *YarpListener*

__init__(*name: str*, *in_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

The AudioChunk listener using the Sound construct parsed as a numpy array.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **persistent** – bool: Whether the subscriber port should remain connected after closure. Default is True
- **channels** – int: Number of channels in the audio. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: Number of samples in the audio chunk. Default is -1 (use the chunk size of the received audio)

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection to the publisher.

Parameters
repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

`listen()`

Listen for a message.

Returns

Tuple[np.ndarray, int]: The received message as a numpy array formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

`class wrapyfi.listeners.yarp.YarpPropertiesListener(name, in_topic, **kwargs)`

Bases: *YarpListener*

`__init__(name, in_topic, **kwargs)`

Initialize the subscriber.

Parameters

- **name** – str: Name of the publisher
- **in_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **persistent** – bool: Whether the subscriber port should remain connected after closure. Default is True
- **yarp_kwargs** – dict: Additional kwargs for the Yarp middleware
- **kwargs** – dict: Additional kwargs for the subscriber

wrapyfi.listeners.zeromq module

`class wrapyfi.listeners.zeromq.ZeroMQListener(name: str, in_topic: str, carrier: str = 'tcp', should_wait: bool = True, socket_ip: str = '127.0.0.1', socket_pub_port: int = 5555, pubsub_monitor_topic: str = 'ZEROMQ/CONNECTIONS', pubsub_monitor_listener_spawn: str | None = 'process', zeromq_kwargs: dict | None = None, **kwargs)`

Bases: *Listener*

`__init__(name: str, in_topic: str, carrier: str = 'tcp', should_wait: bool = True, socket_ip: str = '127.0.0.1', socket_pub_port: int = 5555, pubsub_monitor_topic: str = 'ZEROMQ/CONNECTIONS', pubsub_monitor_listener_spawn: str | None = 'process', zeromq_kwargs: dict | None = None, **kwargs)`

Initialize the subscriber.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **socket_ip** – str: IP address of the socket. Default is '127.0.0.1

- **socket_pub_port** – int: Port of the socket for publishing. Note that the subscriber listens directly to this port which is proxied . Default is 5555
- **pubsub_monitor_topic** – str: Topic to monitor the connections. Default is 'ZEROMQ/CONNECTIONS'
- **pubsub_monitor_listener_spawn** – str: Whether to spawn the PUB/SUB monitor listener as a process or thread. Default is 'process'
- **zeromq_kwargs** – dict: Additional kwargs for the ZeroMQ middleware
- **kwargs** – dict: Additional kwargs for the subscriber

await_connection(*socket=None*, *in_topic: str | None = None*, *repeats: int | None = None*)

Wait for the connection to be established.

Parameters

- **socket** – zmq.Socket: Socket to await connection to
- **in_topic** – str: Name of the input topic
- **repeats** – int: Number of repeats to await connection. None for infinite. Default is None

Returns

bool: True if connection established, False otherwise

read_socket(*socket*)

Read the socket.

Parameters

socket – zmq.Socket: Socket to read from

Returns

bytes: Data read from the socket

Returns

yarp.Value: Value read from the port

close()

Close the subscriber.

class wrapyfi.listeners.zeromq.**ZeroMQNativeObjectListener**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ZeroMQListener*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

The NativeObject listener using the ZeroMQ message construct assuming the data is serialized as a JSON string. Deserializes the data (including plugins) using the decoder and parses it to a native object.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True

- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection to the publisher.

Parameters
repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

listen()

Listen for a message.

Returns
Any: The received message as a native python object

class wrapyfi.listeners.zeromq.**ZeroMQImageListener**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

Bases: *ZeroMQNativeObjectListener*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

The Image listener using the ZeroMQ message construct parsed to a numpy array.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **width** – int: Width of the image. Default is -1 (use the width of the received image)
- **height** – int: Height of the image. Default is -1 (use the height of the received image)
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be decompressed from JPG. Default is False

listen()

Listen for a message.

Returns
np.ndarray: The received message as a numpy array formatted as a cv2 image np.ndarray[img_height, img_width, channels]

class wrapyfi.listeners.zeromq.**ZeroMQAudioChunkListener**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

Bases: *ZeroMQImageListener*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

The AudioChunk listener using the ZeroMQ message construct parsed to a numpy array.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **queue_size** – int: Size of the queue for the subscriber. Default is 5
- **channels** – int: Number of channels in the audio. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: Number of samples in the audio chunk. Default is -1 (use the chunk size of the received audio)

listen()

Listen for a message.

Returns

Tuple[np.ndarray, int]: The received message as a numpy array formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

class wrapyfi.listeners.zeromq.**ZeroMQPropertiesListener**(*name*, *in_topic*, ***kwargs*)

Bases: *ZeroMQListener*

__init__(*name*, *in_topic*, ***kwargs*)

Initialize the subscriber.

Parameters

- **name** – str: Name of the subscriber
- **in_topic** – str: Name of the input topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether the subscriber should wait for the publisher to transmit a message. Default is True
- **socket_ip** – str: IP address of the socket. Default is '127.0.0.1
- **socket_pub_port** – int: Port of the socket for publishing. Note that the subscriber listens directly to this port which is proxied . Default is 5555
- **pubsub_monitor_topic** – str: Topic to monitor the connections. Default is 'ZEROMQ/CONNECTIONS'
- **pubsub_monitor_listener_spawn** – str: Whether to spawn the PUB/SUB monitor listener as a process or thread. Default is 'process'
- **zeromq_kwargs** – dict: Additional kwargs for the ZeroMQ middleware
- **kwargs** – dict: Additional kwargs for the subscriber

Module contents

class wrapyfi.listeners.**FallbackListener**(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *missing_middleware_object: str = ''*, ***kwargs*)

Bases: *Listener*

__init__(*name: str*, *in_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *missing_middleware_object: str = ''*, ***kwargs*)

Initialize the Listener.

Parameters

- **name** – str: The name of the listener
- **in_topic** – str: The topic to listen to
- **carrier** – str: The middleware carrier to use
- **should_wait** – bool: Whether to wait for the listener to be established or not

establish(*repeats: int = -1*, ***kwargs*)

Establish the listener.

listen()

Listen for incoming data.

close()

Close the connection.

wrapyfi.middlewares package

Submodules

wrapyfi.middlewares.ros module

class wrapyfi.middlewares.ros.**ROSMiddleware**(**args*, ***kwargs*)

Bases: object

ROS middleware wrapper. This class is a singleton, so it can be instantiated only once. The `activate` method should be called to initialize the middleware. The `deinit` method should be called to deinitialize the middleware and destroy all connections. The `activate` and `deinit` methods are automatically called when the class is instantiated and when the program exits, respectively.

static activate(***kwargs*)

Activate the ROS middleware. This method should be called to initialize the middleware.

Parameters

kwargs – dict: Keyword arguments to be passed to the ROS initialization function

__init__(*node_name: str = 'wrapyfi'*, *anonymous: bool = True*, *disable_signals: bool = True*, **args*, ***kwargs*)

Initialize the ROS middleware. This method is automatically called when the class is instantiated.

Parameters

- **node_name** – str: The name of the ROS node
- **anonymous** – bool: Whether the ROS node should be anonymous

- **disable_signals** – bool: Whether the ROS node should disable signals
- **args** – list: Positional arguments to be passed to the ROS initialization function
- **kwargs** – dict: Keyword arguments to be passed to the ROS initialization function

static deinit()

Deinitialize the ROS middleware. This method is automatically called when the program exits.

class wrapyfi.middlewares.ros.ROSNativeObjectService

Bases: object

class wrapyfi.middlewares.ros.ROSImageService

Bases: object

wrapyfi.middlewares.ros2 module

class wrapyfi.middlewares.ros2.ROS2Middleware(**args*, ***kwargs*)

Bases: object

ROS 2 middleware wrapper. This class is a singleton, so it can be instantiated only once. The `activate` method should be called to initialize the middleware. The `deinit` method should be called to deinitialize the middleware and destroy all connections. The `activate` and `deinit` methods are automatically called when the class is instantiated and when the program exits, respectively.

static activate(***kwargs*)

Activate the ROS 2 middleware. This method should be called to initialize the middleware.

Parameters

kwargs – dict: Keyword arguments to be passed to the ROS 2 initialization function

__init__(**args*, ***kwargs*)

Initialize the ROS 2 middleware. This method is automatically called when the class is instantiated.

Parameters

- **args** – list: Positional arguments to be passed to the ROS 2 initialization function
- **kwargs** – dict: Keyword arguments to be passed to the ROS 2 initialization function

static deinit()

Deinitialize the ROS 2 middleware. This method is automatically called when the program exits.

wrapyfi.middlewares.yarp module

class wrapyfi.middlewares.yarp.YarpMiddleware(**args*, ***kwargs*)

Bases: object

YARP middleware wrapper. This class is a singleton, so it can be instantiated only once. The `activate` method should be called to initialize the middleware. The `deinit` method should be called to deinitialize the middleware and destroy all connections. The `activate` and `deinit` methods are automatically called when the class is instantiated and when the program exits, respectively.

static activate(***kwargs*)

Activate the YARP middleware. This method should be called to initialize the middleware.

Parameters

kwargs – dict: Keyword arguments to be passed to the YARP initialization function

__init__(**args*, ***kwargs*)

Initialize the YARP middleware. This method is automatically called when the class is instantiated.

static deinit()

Deinitialize the YARP middleware. This method is automatically called when the program exits.

wrapyfi.middlewares.zeromq module

class wrapyfi.middlewares.zeromq.**ZeroMQMiddlewarePubSub**(**args*, ***kwargs*)

Bases: object

ZeroMQ PUB/SUB middleware wrapper. This class is a singleton, so it can be instantiated only once. The `activate` method should be called to initialize the middleware. The `deinit` method should be called to deinitialize the middleware and destroy all connections. The `activate` and `deinit` methods are automatically called when the class is instantiated and when the program exits, respectively.

class ZeroMQSharedMonitorData(*use_multiprocessing: bool = False*)

Bases: object

Shared data class for the ZeroMQ PUB/SUB monitor. This class is used to share data between the main process and the monitor listener process/thread.

__init__(*use_multiprocessing: bool = False*)

Initialize the shared data class.

Parameters

use_multiprocessing – bool: Whether to use multiprocessing or threading

add_topic(*topic: str*)

Add a topic to the shared data class.

Parameters

topic – str: The topic to add

remove_topic(*topic: str*)

Remove a topic from the shared data class.

Parameters

topic – str: The topic to remove

get_topics()

Get the list of topics in the shared data class.

Returns

list: The list of topics

update_connection(*topic: str*, *data: dict*)

Update the connection data for a topic, e.g. the number of subscribers.

Parameters

- **topic** – str: The topic to update
- **data** – dict: The connection data

remove_connection(*topic: str*)

Remove the connection data for a topic.

Parameters

topic – str: The topic to remove

get_connections()

Get the connection data for all topics.

Returns
dict: The connection data for all topics

`is_connected`(*topic: str*)

Check whether a topic is connected.

Parameters
`topic` – str: The topic to check

`static activate`(***kwargs*)

Activate the ZeroMQ PUB/SUB middleware. This method should be called to initialize the middleware.

Parameters
`kwargs` – dict: Keyword arguments to be passed to the ZeroMQ initialization function

`__init__`(*zeromq_proxy_kwargs: dict | None = None*, *zeromq_post_kwargs: dict | None = None*, ***kwargs*)

Initialize the ZeroMQ PUB/SUB middleware. This method is automatically called when the class is instantiated.

Parameters

- `zeromq_proxy_kwargs` – Optional[dict]: Keyword arguments to be passed to the ZeroMQ proxy initialization function
- `zeromq_post_kwargs` – Optional[dict]: Keyword arguments to be passed to the ZeroMQ initialization function (these are ZeroMQ options)
- `kwargs` – dict: Keyword arguments to be passed to the ZeroMQ initialization function

`static proxy_thread`(*socket_pub_address: str = 'tcp://127.0.0.1:5555'*, *socket_sub_address: str = 'tcp://127.0.0.1:5556'*, *inproc_address: str = 'inproc://monitor'*)

Proxy thread for the ZeroMQ PUB/SUB proxy.

Parameters

- `socket_pub_address` – str: The address of the PUB socket
- `socket_sub_address` – str: The address of the SUB socket
- `inproc_address` – str: The address of the inproc socket (connections within the same process, for exchanging subscription data between the proxy and the monitor)

`static subscription_monitor_thread`(*inproc_address: str = 'inproc://monitor'*, *socket_sub_address: str = 'tcp://127.0.0.1:5556'*, *pubsub_monitor_topic: str = 'ZEROMQ/CONNECTIONS'*, *verbose: bool = False*)

Subscription monitor thread for the ZeroMQ PUB/SUB proxy.

Parameters

- `inproc_address` – str: The address of the inproc socket (connections within the same process, for exchanging subscription data between the proxy and the monitor)
- `socket_sub_address` – str: The address of the SUB socket
- `pubsub_monitor_topic` – str: The topic to use for publishing subscription data
- `verbose` – bool: Whether to print debug messages

`static deinit`()

class wrapyfi.middlewares.zeromq.**ZeroMQMiddlewareReqRep**(**args*, ***kwargs*)

Bases: object

ZeroMQ REQ/REP middleware wrapper. This class is a singleton, so it can be instantiated only once. The activate method should be called to initialize the middleware. The deinit method should be called to deinitialize the middleware and destroy all connections. The activate and deinit methods are automatically called when the class is instantiated and when the program exits, respectively.

static activate(***kwargs*)

Activate the ZeroMQ REQ/REP middleware. This method should be called to initialize the middleware.

Parameters

kwargs – dict: Keyword arguments to be passed to the ZeroMQ initialization function

__init__(*zeromq_proxy_kwargs: dict | None = None*, *zeromq_post_kwargs: dict | None = None*, **args*, ***kwargs*)

Initialize the ZeroMQ REQ/REP middleware. This method is automatically called when the class is instantiated.

Parameters

- **zeromq_proxy_kwargs** – Optional[dict]: Keyword arguments to be passed to the ZeroMQ proxy initialization function
- **zeromq_post_kwargs** – Optional[dict]: Keyword arguments to be passed to the ZeroMQ initialization function (these are ZeroMQ options)
- **args** – list: Positional arguments to be passed to the ZeroMQ initialization function
- **kwargs** – dict: Keyword arguments to be passed to the ZeroMQ initialization function

static deinit()

class wrapyfi.middlewares.zeromq.**ZeroMQMiddlewareParamServer**(**args*, ***kwargs*)

Bases: object

ZeroMQ parameter server middleware wrapper. This class is a singleton, so it can be instantiated only once. The activate method should be called to initialize the middleware. The deinit method should be called to deinitialize the middleware and destroy all connections. The activate and deinit methods are automatically called when the class is instantiated and when the program exits, respectively.

Note: This parameter server is experimental and not fully tested.

static activate(***kwargs*)

__init__(*zeromq_proxy_kwargs: dict | None = None*, *zeromq_post_kwargs: Optional = None*, **args*, ***kwargs*)

Initialize the ZeroMQ parameter server middleware. This method is automatically called when the class is instantiated.

Parameters

- **zeromq_proxy_kwargs** – Optional[dict]: Keyword arguments to be passed to the ZeroMQ proxy initialization function
- **zeromq_post_kwargs** – Optional[dict]: Keyword arguments to be passed to the ZeroMQ initialization function (these are ZeroMQ options)
- **kwargs** – dict: Keyword arguments to be passed to the ZeroMQ initialization function

static publish_params(*param_server*, *params: dict*, *cached_params: dict*, *root_topics: set*, *update_trigger: bool*)

Publish parameters to the parameter server.

Parameters

- **param_server** – zmq.Socket: The parameter server socket
- **params** – dict: The parameters to be published
- **cached_params** – dict: The cached parameters. This is used to check whether the parameters have changed
- **root_topics** – set: The root topics. This is used to check whether there are any active subscribers
- **update_trigger** – bool: Whether to trigger an update of the parameters

Returns

Tuple[bool, dict]: Whether to trigger an update of the parameters and the cached parameters

static deinit()

Deinitialize the ZeroMQ parameter server.

Module contents

wrapyfi.plugins package

Submodules

wrapyfi.plugins.cupy_array module

Encoder and Decoder for CuPy Array Data via Wrapyfi.

This script provides mechanisms to encode and decode CuPy array data using Wrapyfi. It leverages base64 encoding to convert binary data into ASCII strings.

The script contains a class, CuPyArray, registered as a plugin to manage the conversion of CuPy array data (if available) between its original and encoded forms. CuPy only supports GPU devices, specifically NVIDIA CUDA devices. Installing CUDA Toolkit (https://developer.nvidia.com/cuda-toolkit) is required to use CuPy.

Requirements: - Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions) - CuPy: A GPU-accelerated library for numerical computations with a NumPy-compatible interface (refer to https://docs.cupy.dev/en/stable/install.html for installation instructions) Note: If CuPy is not available, HAVE_CUPY will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:

`pip install cupy-cuda12x` # Basic installation of CuPy. Replace 12x with your CUDA version e.g., cupy-cuda11x

wrapyfi.plugins.cupy_array.**cupy_device_to_str**(*device*)

Convert a CuPy device to a string representation.

Parameters

device – Union[cupy.cuda.Device, int]: The CuPy device

Returns

str: A string representing the CuPy device

wrapyfi.plugins.cupy_array.**cupy_str_to_device**(*device_str*)

Convert a string to a CuPy device.

Parameters
device_str – str: A string representing a CuPy device

Returns
cupy.cuda.Device: A CuPy device

class wrapyfi.plugins.cupy_array.**CuPyArray**(*load_cupy_device=None*, *map_cupy_devices=None*, ***kwargs*)

Bases: *Plugin*

__init__(*load_cupy_device=None*, *map_cupy_devices=None*, ***kwargs*)

Initialize the CuPy plugin.

Parameters

- **load_cupy_device** – cupy.cuda.Device or str: Default CuPy device to load tensors onto
- **map_cupy_devices** – dict: [Optional] A dictionary mapping encoded device strings to decoding devices

encode(*obj*, **args*, ***kwargs*)

Encode CuPy array into a base64 ASCII string.

Parameters
obj – cupy.ndarray: The CuPy array to encode

Returns
Tuple[bool, dict]

decode(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a base64 ASCII string back into CuPy array.

Parameters
obj_full – tuple: A tuple containing the encoded data string and device string

Returns
Tuple[bool, cupy.ndarray]

wrapyfi.plugins.dask_data module

Encoder and Decoder for Dask Arrays/Dataframes Data via Wrapyfi.

This script provides mechanisms to encode and decode Dask data using Wrapyfi. It utilizes base64 encoding to convert binary data into ASCII strings.

The script contains a class, *DaskData*, registered as a plugin to manage the conversion of Dask data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- Dask: A flexible library for parallel computing in Python (refer to https://docs.dask.org/en/latest/install.html for installation instructions)

- **pandas: A data structures library for data analysis, time series, and statistics (refer to https://pandas.pydata.org/pandas-docs/stable/getting_started/install.html for installation instructions)**

  Note: If Dask or pandas is not available, HAVE_DASK will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:

`pip install pandas dask[complete]` # Basic installation of Dask and pandas

class wrapyfi.plugins.dask_data.**DaskData**(***kwargs*)

Bases: *Plugin*

__init__(***kwargs*)

Initialize the DaskData plugin.

encode(*obj*, **args*, ***kwargs*)

Encode Dask data into a base64 ASCII string.

Parameters

- **obj** – Union[dd.DataFrame, dd.Series, da.Array]: The Dask data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, encoded data string, and object type

decode(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a base64 ASCII string back into Dask data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string and object type
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, Union[dd.DataFrame, dd.Series, da.Array]]: A tuple containing: - bool: Always True, indicating that the decoding was successful - Union[dd.DataFrame, da.Array]: The decoded Dask data

wrapyfi.plugins.jax_tensor module

Encoder and Decoder for JAX Tensor Data via Wrapyfi.

This script provides mechanisms to encode and decode JAX tensor data using Wrapyfi. It utilizes base64 encoding to convert binary data into ASCII strings.

The script contains a class, *JAXTensor*, registered as a plugin to manage the conversion of JAX tensor data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- **JAX: An open-source high-performance machine learning library (refer to https://github.com/google/jax for installation instructions)**
  Note: If JAX is not available, HAVE_JAX will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:
`pip install jax jaxlib` # Basic installation of JAX

`class wrapyfi.plugins.jax_tensor.JAXTensor(**kwargs)`

Bases: *Plugin*

`__init__(**kwargs)`

Initialize the JAXTensor plugin.

`encode(obj, *args, **kwargs)`

Encode JAX tensor data into a base64 ASCII string.

Parameters

- **obj** – jax.numpy.DeviceArray: The JAX tensor data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name and encoded data string

`decode(obj_type, obj_full, *args, **kwargs)`

Decode a base64 ASCII string back into JAX tensor data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, jax.numpy.DeviceArray]: A tuple containing: - bool: Always True, indicating that the decoding was successful - jax.numpy.DeviceArray: The decoded JAX tensor data

wrapyfi.plugins.mxnet_tensor module

Encoder and Decoder for MXNet Tensor Data via Wrapyfi.

This script provides mechanisms to encode and decode MXNet tensor data using Wrapyfi. It leverages base64 encoding to convert binary data into ASCII strings.

The script contains a class, *MXNetTensor*, registered as a plugin to manage the conversion of MXNet tensor data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- **MXNet: A deep learning framework designed for both efficiency and flexibility (refer to https://mxnet.apache.org/get_started for installation instructions)**
  Note: If MXNet is not available, HAVE_MXNET will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:
`pip install mxnet` # Basic installation of MXNet

`wrapyfi.plugins.mxnet_tensor.`**`mxnet_device_to_str`**`(`*`device`*`)`

Convert an MXNet device to a string representation.

Parameters
`device` – Union[str, mxnet.Context, dict]: Various possible types representing an MXNet device as mxnet.Context or str. Also accepts a dictionary mapping encoded device strings to decoding devices

Returns
Union[str, dict]: A string or dictionary representing the MXNet device

`wrapyfi.plugins.mxnet_tensor.`**`mxnet_str_to_device`**`(`*`device`*`)`

Convert a string to an MXNet device.

Parameters
`device` – str: A string representing an MXNet device

Returns
mxnet.Context: An MXNet context representing the device

`class` `wrapyfi.plugins.mxnet_tensor.`**`MXNetTensor`**`(`*`load_mxnet_device=None`*`, `*`map_mxnet_devices=None`*`, `*`**kwargs`*`)`

Bases: *Plugin*

`__init__``(`*`load_mxnet_device=None`*`, `*`map_mxnet_devices=None`*`, `*`**kwargs`*`)`

Initialize the MXNetTensor plugin.

Parameters

- **`load_mxnet_device`** – Union[mxnet.Context, str]: Default MXNet device to load tensors onto
- **`map_mxnet_devices`** – dict: [Optional] A dictionary mapping encoded device strings to decoding devices

`encode``(`*`obj`*`, `*`*args`*`, `*`**kwargs`*`)`

Encode MXNet tensor data into a base64 ASCII string.

Parameters

- **`obj`** – mxnet.nd.NDArray: The MXNet tensor data to encode
- **`args`** – tuple: Additional arguments (not used)
- **`kwargs`** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, encoded data string, and device string.

decode(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a base64 ASCII string back into MXNet tensor data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string and device string
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, mxnet.nd.NDArray]: A tuple containing: - bool: Always True, indicating that the decoding was successful - mxnet.nd.NDArray: The decoded MXNet tensor data

wrapyfi.plugins.paddle_tensor module

Encoder and Decoder for PaddlePaddle Tensor Data via Wrapyfi.

This script provides mechanisms to encode and decode PaddlePaddle tensor data using Wrapyfi. It utilizes base64 encoding to convert binary data into ASCII strings.

The script contains a class, *PaddleTensor*, registered as a plugin to manage the conversion of PaddlePaddle tensor data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- **PaddlePaddle: An open-source deep learning platform developed by Baidu (refer to https://www.paddlepaddle.org.cn/en/install/quick for installation instructions)**
  Note: If PaddlePaddle is not available, HAVE_PADDLE will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:

```
pip install paddlepaddle # Basic installation of PaddlePaddle
```

wrapyfi.plugins.paddle_tensor.**paddle_device_to_str**(*device*)

Convert a PaddlePaddle device to a string representation.

Parameters

device – Union[str, paddle.fluid.libpaddle.Place, dict]: Various possible types representing a PaddlePaddle device

Returns

Union[str, dict]: A string or dictionary representing the PaddlePaddle device

wrapyfi.plugins.paddle_tensor.**paddle_str_to_device**(*device*)

Convert a string to a PaddlePaddle device.

Parameters

device – str: A string representing a PaddlePaddle device

Returns

paddle.fluid.libpaddle.Place: A PaddlePaddle place representing the device

class wrapyfi.plugins.paddle_tensor.**PaddleTensor**(*load_paddle_device=None*, *map_paddle_devices=None*, ***kwargs*)

Bases: *Plugin*

__init__(*load_paddle_device=None*, *map_paddle_devices=None*, ***kwargs*)

Initialize the PaddleTensor plugin.

Parameters

- **load_paddle_device** – Union[paddle.fluid.libpaddle.Place, str]: Default PaddlePaddle device to load tensors onto
- **map_paddle_devices** – dict: A dictionary mapping encoded device strings to decoding devices

encode(*obj*, **args*, ***kwargs*)

Encode PaddlePaddle tensor data into a base64 ASCII string.

Parameters

- **obj** – paddle.Tensor: The PaddlePaddle tensor data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, encoded data string, and device string

decode(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a base64 ASCII string back into PaddlePaddle tensor data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string and device string
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, paddle.Tensor]: A tuple containing: - bool: Always True, indicating that the decoding was successful - paddle.Tensor: The decoded PaddlePaddle tensor data

wrapyfi.plugins.pandas_data module

Encoder and Decoder for pandas Series/Dataframes Data via Wrapyfi.

This script provides mechanisms to encode and decode pandas data using Wrapyfi. It utilizes base64 encoding to convert binary data into ASCII strings for "pandas<2.0". It utilizes base64 encoding and pickle serialization for "pandas>=2.0".

The script contains a class, *PandasData*, registered as a plugin to manage the conversion of pandas data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- **pandas: A data structures library for data analysis, time series, and statistics (refer to https://pandas.pydata.org/pandas-docs/stable/getting_started/install.html for installation instructions)**
  Note: If pandas is not available, HAVE_PANDAS will be set to False and the plugin will be registered with no types. For pandas>=2.0, the object is pickled (slower but identical on decoding). For pandas<2.0, the object is encoded using base64 (faster but potentially non-identical on decoding).

You can install the necessary packages using pip:
`pip install pandas` # Basic installation of pandas

class wrapyfi.plugins.pandas_data.**PandasData**(***kwargs*)

Bases: *Plugin*

__init__(***kwargs*)

Initialize the PandasData plugin.

encode(*obj*, **args*, ***kwargs*)

Encode pandas data using pickle and base64.

Parameters

- **obj** – Union[pandas.DataFrame, pandas.Series]: The pandas data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, pickled data string, and any buffer data

decode(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a pickled and base64 encoded string back into pandas data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the pickled data string and any buffer data
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, Union[pandas.DataFrame, pandas.Series]]: A tuple containing: - bool: Always True, indicating that the decoding was successful - Union[pandas.DataFrame, pandas.Series]: The decoded pandas data

class wrapyfi.plugins.pandas_data.**PandasLegacyData**(***kwargs*)

Bases: *Plugin*

__init__(***kwargs*)

Initialize the PandasLegacyData plugin.

encode(*obj*, **args*, ***kwargs*)

Encode pandas data into a base64 ASCII string.

Parameters

- **obj** – Union[pandas.DataFrame, pandas.Series]: The pandas data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, encoded data string, and object type

decode(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a base64 ASCII string back into pandas data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string and object type
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, Union[pandas.DataFrame, pandas.Series]]: A tuple containing: - bool: Always True, indicating that the decoding was successful - Union[pandas.DataFrame, pandas.Series]: The decoded pandas data

wrapyfi.plugins.pillow_image module

Encoder and Decoder for PIL Image Data via Wrapyfi.

This script provides mechanisms to encode and decode PIL Image data using Wrapyfi. It utilizes base64 encoding to convert binary data into ASCII strings, and also handles non-ASCII encodings for certain image formats.

The script contains a class, *PILImage*, registered as a plugin to manage the conversion of PIL Image data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- **PIL (Pillow): A Python Imaging Library that adds image processing capabilities to your Python interpreter (refer to https://pillow.readthedocs.io/en/stable/installation.html for installation instructions)**
  Note: If PIL is not available, HAVE_PIL will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:
`pip install Pillow` # Basic installation of Pillow (PIL Fork)

class wrapyfi.plugins.pillow_image.**PILImage**(***kwargs*)

Bases: *Plugin*

`__init__`(***kwargs*)

Initialize the PILImage plugin.

`encode`(*obj*, **args*, ***kwargs*)

Encode PIL Image data into a base64 ASCII string.

Parameters

- **obj** – Image.Image: The PIL Image data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name and encoded data string, with optional image size and mode for raw data

`decode`(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a base64 ASCII string back into PIL Image data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string and optionally image size and mode for raw data
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, Image.Image]: A tuple containing: - bool: Always True, indicating that the decoding was successful - Image.Image: The decoded PIL Image data

wrapyfi.plugins.pint_quantities module

Encoder and Decoder for Pint Quantity Data via Wrapyfi.

This script provides mechanisms to encode and decode Pint Quantity Data using Wrapyfi. It utilizes base64 encoding and JSON serialization.

The script contains a class, *PintData*, registered as a plugin to manage the conversion of Pint Quantity data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- **Pint: A package to define, operate and manipulate physical quantities (refer to https://pint.readthedocs.io/en/stable/ for installation instructions)**
  Note: If Pint is not available, HAVE_PINT will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:

```
pip install pint # Basic installation of Pint
```

class wrapyfi.plugins.pint_quantities.**PintData**(***kwargs*)

Bases: *Plugin*

__init__(***kwargs*)

Initialize the PintData plugin.

encode(*obj*, **args*, ***kwargs*)

Encode Pint Quantity data into a base64 ASCII string.

Parameters

- **obj** – pint.Quantity: The Pint Quantity data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, encoded data string, and object type

decode(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a base64 ASCII string back into Pint Quantity data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string and object type
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, pint.Quantity]: A tuple containing: - bool: Indicating that the decoding was successful - pint.Quantity: The decoded Pint Quantity data

wrapyfi.plugins.pyarrow_array module

Encoder and Decoder for PyArrow StructArray Data via Wrapyfi.

This script provides mechanisms to encode and decode PyArrow StructArray Data using Wrapyfi. It utilizes base64 encoding and pickle serialization.

The script contains a class, *PyArrowArray*, registered as a plugin to manage the conversion of PyArrow StructArray data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- **PyArrow: A cross-language development platform for in-memory data (refer to https://arrow.apache.org/install/ for installation instructions)**
  Note: If PyArrow is not available, HAVE_PYARROW will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:

```
pip install pyarrow # Basic installation of PyArrow
```

class wrapyfi.plugins.pyarrow_array.**PyArrowArray**(***kwargs*)

Bases: *Plugin*

__init__(***kwargs*)

Initialize the PyArrowArray plugin.

encode(*obj*, **args*, ***kwargs*)

Encode PyArrow StructArray data using pickle and base64.

Parameters

- **obj** – pa.StructArray: The PyArrow StructArray data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, pickled data string, and any buffer data

decode(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a pickled and base64 encoded string back into PyArrow StructArray data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the pickled data string and any buffer data
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, pa.StructArray]: A tuple containing: - bool: Always True, indicating that the decoding was successful - pa.StructArray: The decoded PyArrow StructArray data

wrapyfi.plugins.pytorch_tensor module

Encoder and Decoder for PyTorch Tensor Data via Wrapyfi.

This script provides mechanisms to encode and decode PyTorch tensor data using Wrapyfi. It utilizes base64 encoding to convert binary data into ASCII strings.

The script contains a class, *PytorchTensor*, registered as a plugin to manage the conversion of PyTorch tensor data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- **PyTorch: An open-source machine learning library developed by Facebook's AI Research lab (refer to https://pytorch.org/get-started/locally/ for installation instructions)**
  Note: If PyTorch is not available, HAVE_TORCH will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:

`pip install torch` # Basic installation of PyTorch

class wrapyfi.plugins.pytorch_tensor.**PytorchTensor**(*load_torch_device=None*, *map_torch_devices=None*, ***kwargs*)

Bases: *Plugin*

__init__(*load_torch_device=None*, *map_torch_devices=None*, ***kwargs*)

Initialize the PytorchTensor plugin.

Parameters

- **load_torch_device** – str: Default PyTorch device to load tensors onto
- **map_torch_devices** – dict: A dictionary mapping encoded device strings to decoding devices

encode(*obj*, **args*, ***kwargs*)

Encode PyTorch tensor data into a base64 ASCII string.

Parameters

- **obj** – torch.Tensor: The PyTorch tensor data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, encoded data string, and device string

decode(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a base64 ASCII string back into PyTorch tensor data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string and device string
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, torch.Tensor]: A tuple containing: - bool: Always True, indicating that the decoding was successful - torch.Tensor: The decoded PyTorch tensor data

wrapyfi.plugins.tensorflow_tensor module

Encoder and Decoder for TensorFlow Tensor Data via Wrapyfi.

This script provides mechanisms to encode and decode TensorFlow tensor data using Wrapyfi. It utilizes base64 encoding to convert binary data into ASCII strings.

The script contains a class, *TensorflowTensor*, registered as a plugin to manage the conversion of TensorFlow tensor data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)

- **TensorFlow: An end-to-end open-source platform for machine learning (refer to https://www.tensorflow.org/install for installation instructions)**
  Note: If TensorFlow is not available, HAVE_TENSORFLOW will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:
`pip install tensorflow` # Basic installation of TensorFlow

`class wrapyfi.plugins.tensorflow_tensor.TensorflowTensor(**kwargs)`

Bases: *Plugin*

`__init__(**kwargs)`

Initialize the TensorflowTensor plugin.

`encode(obj, *args, **kwargs)`

Encode TensorFlow tensor data into a base64 ASCII string.

Parameters

- **obj** – tensorflow.Tensor: The TensorFlow tensor data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name and encoded data string

`decode(obj_type, obj_full, *args, **kwargs)`

Decode a base64 ASCII string back into TensorFlow tensor data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, tensorflow.Tensor]: A tuple containing: - bool: Always True, indicating that the decoding was successful - tensorflow.Tensor: The decoded TensorFlow tensor data

wrapyfi.plugins.trax_array module

Encoder and Decoder for Trax Array Data via Wrapyfi.

This script provides mechanisms to encode and decode Trax Array Data using Wrapyfi. It utilizes base64 encoding and pickle serialization.

The script contains a class, *TraxArray*, registered as a plugin to manage the conversion of Trax Array data (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)

- **Trax: A deep learning library that focuses on clear code and speed (refer to https://trax-ml.readthedocs.io/en/latest/notebooks/trax_intro.html for installation instructions)**
  Note: If Trax is not available, HAVE_TRAX will be set to False and the plugin will be registered with no types. Trax uses JAX or TensorFlow-NumPy as its backend, so they must be installed as well. Trax installs JAX as a dependency, but TensorFlow must be installed separately.

You can install the necessary packages using pip:
`pip install trax` # Basic installation of Trax

class wrapyfi.plugins.trax_array.**TraxArray**(***kwargs*)

Bases: *Plugin*

__init__(***kwargs*)

Initialize the TraxArray plugin.

encode(*obj*, **args*, ***kwargs*)

Encode Trax Array data using pickle and base64.

Parameters

- **obj** – jaxlib.xla_extension.ArrayImpl: The Trax Array data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, pickled data string, and any buffer data

decode(*obj_type*, *obj_full*, **args*, ***kwargs*)

Decode a pickled and base64 encoded string back into Trax Array data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the pickled data string and any buffer data
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, pa.StructArray]: A tuple containing: - bool: Always True, indicating that the decoding was successful - jaxlib.xla_extension.ArrayImpl: The decoded Trax Array data

wrapyfi.plugins.xarray_data module

Encoder and Decoder for XArray DataArray/Dataset Data via Wrapyfi.

This script provides mechanisms to encode and decode XArray Data using Wrapyfi. It utilizes base64 encoding and JSON serialization.

The script contains a class, *XArrayData*, registered as a plugin to manage the conversion of XArray Data (DataArray and Dataset) (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- pandas: A data structures library for data analysis, time series, and statistics (refer to https://pandas.pydata.org/pandas-docs/stable/getting_started/install.html for installation instructions)
- **XArray: N-D labeled arrays and datasets in Python (refer to http://xarray.pydata.org/en/stable/getting-started-guide/installing.html for installation instructions)**
  Note: If XArray is not available, HAVE_XARRAY will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:
`pip install pandas xarray` # Basic installation of XArray

`class wrapyfi.plugins.xarray_data.XArrayData(**kwargs)`

Bases: *Plugin*

`__init__(**kwargs)`

Initialize the XArrayData plugin.

`encode(obj, *args, **kwargs)`

Encode XArray Data using JSON and base64.

Parameters

- **obj** – Union[xr.DataArray, xr.Dataset]: The XArray Data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, encoded data string, data type, and object name

`decode(obj_type, obj_full, *args, **kwargs)`

Decode a JSON and base64 encoded string back into XArray Data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string, data type, and object name
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, Union[xr.DataArray, xr.Dataset]]: A tuple containing: - bool: Always True, indicating that the decoding was successful - Union[xr.DataArray, xr.Dataset]: The decoded XArray Data

wrapyfi.plugins.zarr_array module

Encoder and Decoder for Zarr Array/Group Data via Wrapyfi.

This script provides mechanisms to encode and decode Zarr Data using Wrapyfi. It utilizes base64 encoding and zip compression.

The script contains a class, *ZarrData*, registered as a plugin to manage the conversion of Zarr Data (Array and Group) (if available) between its original and encoded forms.

Requirements:

- Wrapyfi: Middleware communication wrapper (refer to the Wrapyfi documentation for installation instructions)
- **Zarr: A format for the storage of chunked, compressed, N-dimensional arrays (refer to https://zarr.readthedocs.io/en/stable/ for installation instructions)**
  Note: If Zarr is not available, HAVE_ZARR will be set to False and the plugin will be registered with no types.

You can install the necessary packages using pip:
`pip install zarr` # Basic installation of Zarr

`class wrapyfi.plugins.zarr_array.ZarrData(**kwargs)`

Bases: *Plugin*

`__init__(**kwargs)`

Initialize the ZarrData plugin.

`encode(obj, *args, **kwargs)`

Encode Zarr Data using zip compression and base64.

Parameters

- **obj** – Union[zarr.Array, zarr.Group]: The Zarr Data to encode
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, dict]: A tuple containing: - bool: Always True, indicating that the encoding was successful - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name, encoded data string, data type, and object name.

`decode(obj_type, obj_full, *args, **kwargs)`

Decode a zip compressed and base64 encoded string back into Zarr Data.

Parameters

- **obj_type** – type: The expected type of the decoded object (not used)
- **obj_full** – tuple: A tuple containing the encoded data string, data type, and object name
- **args** – tuple: Additional arguments (not used)
- **kwargs** – dict: Additional keyword arguments (not used)

Returns

Tuple[bool, Union[zarr.Array, zarr.Group]]: A tuple containing: - bool: Always True, indicating that the decoding was successful - Union[zarr.Array, zarr.Group]: The decoded Zarr Data

Module contents

wrapyfi.publishers package

Submodules

wrapyfi.publishers.ros module

class wrapyfi.publishers.ros.**ROSPublisher**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *ros_kwargs: dict | None = None*, ***kwargs*)

Bases: *Publisher*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *ros_kwargs: dict | None = None*, ***kwargs*)

Initialize the publisher.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **queue_size** – int: Queue size for the publisher. Default is 5
- **ros_kwargs** – dict: Additional kwargs for the ROS middleware
- **kwargs** – dict: Additional kwargs for the publisher

await_connection(*publisher*, *out_topic: str | None = None*, *repeats: int | None = None*)

Wait for at least one subscriber to connect to the publisher.

Parameters

- **publisher** – rospy.Publisher: Publisher to await connection to
- **out_topic** – str: Name of the output topic
- **repeats** – int: Number of repeats to await connection. None for infinite. Default is None

Returns

bool: True if connection established, False otherwise

close()

Close the publisher

class wrapyfi.publishers.ros.**ROSNativeObjectPublisher**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *serializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROSPublisher*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The NativeObject publisher using the ROS String message assuming a combination of python native objects.

and numpy arrays as input. Serializes the data (including plugins) using the encoder and sends it as a string :param name: str: Name of the publisher :param out_topic: str: Name of the output topic preceded by '/' (e.g. '/topic') :param carrier: str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp' :param should_wait: bool: Whether to wait for at least one listener before unblocking the script. Default is True :param queue_size: int: Queue size for the publisher. Default is 5 :param serializer_kwargs: dict: Additional kwargs for the serializer

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection.

Parameters
repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

publish(*obj*)

Publish the object to the middlware.

Parameters
obj – object: Object to publish

class wrapyfi.publishers.ros.**ROSImagePublisher**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

Bases: *ROSPublisher*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

The ImagePublisher using the ROS Image message assuming a numpy array as input.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **queue_size** – int: Queue size for the publisher. Default is 5
- **width** – int: Width of the image. Default is -1 meaning that the width is not fixed
- **height** – int: Height of the image. Default is -1 meaning that the height is not fixed
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be compressed as JPG. Default is False

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection.

Parameters
repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

publish(*img: numpy.ndarray*)

Publish the image to the middleware.

Parameters
img – np.ndarray: Image to publish formatted as a cv2 image np.ndarray[img_height, img_width, channels]

class wrapyfi.publishers.ros.**ROSAudioChunkPublisher**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

Bases: *ROSPublisher*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

The AudioChunkPublisher using the ROS Audio message assuming a numpy array as input.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **queue_size** – int: Queue size for the publisher. Default is 5
- **channels** – int: Number of channels. Default is 1
- **rate** – int: Sampling rate. Default is 44100
- **chunk** – int: Chunk size. Default is -1 meaning that the chunk size is not fixed

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection.

Parameters
repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

publish(*aud: Tuple[numpy.ndarray, int]*)

Publish the audio chunk to the middleware.

Parameters
aud – Tuple[np.ndarray, int]: Audio chunk to publish formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

class wrapyfi.publishers.ros.**ROSPropertiesPublisher**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *persistent: bool = True*, ***kwargs*)

Bases: *ROSPublisher*

Sets rospy parameters. Behaves differently from other data types by directly setting ROS parameters. Note that the listener is not guaranteed to receive the updated signal, since the listener can trigger before property is set. The property decorated method returns accept native python objects (excluding None), but care should be taken when using dictionaries, since they are analogous with node namespaces: http://wiki.ros.org/rospy/Overview/Parameter%20Server

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *persistent: bool = True*, ***kwargs*)

The PropertiesPublisher using the ROS parameter server.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **persistent** – bool: True if the parameter should be kept on closing node, False if it should be deleted or reset to its state before the node was started. Default is True

establish(***kwargs*)

Store the original property value in case it needs to be reset.

publish(*obj*)

Publish the property to the middleware (parameter server).

Parameters

obj – object: Property to publish. If dict, will be set as a namespace

close()

Close the publisher and reset the property to its original value if not persistent.

class wrapyfi.publishers.ros.**ROSMessagePublisher**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, ***kwargs*)

Bases: *ROSPublisher*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *queue_size: int = 5*, ***kwargs*)

The ROSMessagePublisher using the ROS message type inferred from the message type. Supports standard ROS msgs.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **queue_size** – int: Queue size for the publisher. Default is 5

establish(*repeats: int | None = None*, *obj=None*, ***kwargs*)

Establish the connection and import the message requirements.

Parameters

- **repeats** – int: Number of repeats to await connection. None for infinite. Default is None
- **obj** – object: Object to establish the connection to

Returns

bool: True if connection established, False otherwise

publish(*obj*)

Publish the object to the middleware.

Parameters

obj – object: ROS message to publish

wrapyfi.publishers.ros2 module

class wrapyfi.publishers.ros2.**ROS2Publisher**(*name: str*, *out_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *ros2_kwargs: dict | None = None*, ***kwargs*)

Bases: *Publisher*, Node

__init__(*name: str*, *out_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *ros2_kwargs: dict | None = None*, ***kwargs*)

Initialize the publisher.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **queue_size** – int: Queue size for the publisher. Default is 5
- **ros2_kwargs** – dict: Additional kwargs for the ROS 2 middleware
- **kwargs** – dict: Additional kwargs for the publisher

await_connection(*publisher*, *out_topic: str | None = None*, *repeats: int | None = None*)

Wait for at least one subscriber to connect to the publisher.

Parameters

- **publisher** – rclpy.publisher.Publisher: Publisher to await connection to
- **out_topic** – str: Name of the output topic
- **repeats** – int: Number of repeats to await connection. None for infinite. Default is None

Returns

bool: True if connection established, False otherwise

close()

Close the publisher

class wrapyfi.publishers.ros2.**ROS2NativeObjectPublisher**(*name*, *out_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *serializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROS2Publisher*

__init__(*name*, *out_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The NativeObject publisher using the ROS 2 String message assuming a combination of python native objects and numpy arrays as input. Serializes the data (including plugins) using the encoder and sends it as a string.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **queue_size** – int: Queue size for the publisher. Default is 5
- **serializer_kwargs** – dict: Additional kwargs for the serializer

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection.

Parameters
repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

publish(*obj*)

Publish the object to the middleware

Parameters
obj – object: Object to publish

class wrapyfi.publishers.ros2.**ROS2ImagePublisher**(*name: str*, *out_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

Bases: *ROS2Publisher*

__init__(*name: str*, *out_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

The ImagePublisher using the ROS 2 Image message assuming a numpy array as input.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **queue_size** – int: Queue size for the publisher. Default is 5
- **width** – int: Width of the image. Default is -1 meaning that the width is not fixed

- **height** – int: Height of the image. Default is -1 meaning that the height is not fixed
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be compressed as JPG. Default is False

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection.

Parameters
repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

publish(*img: numpy.ndarray*)

Publish the image to the middleware.

Parameters
img – np.ndarray: Image to publish formatted as a cv2 image np.ndarray[img_height, img_width, channels]

class wrapyfi.publishers.ros2.**ROS2AudioChunkPublisher**(*name: str*, *out_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

Bases: *ROS2Publisher*

__init__(*name: str*, *out_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

The AudioChunkPublisher using the ROS 2 Audio message assuming a numpy array as input.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **queue_size** – int: Queue size for the publisher. Default is 5
- **channels** – int: Number of channels. Default is 1
- **rate** – int: Sampling rate. Default is 44100
- **chunk** – int: Chunk size. Default is -1 meaning that the chunk size is not fixed

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection.

Parameters
repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

publish(*aud: Tuple[numpy.ndarray, int]*)

Publish the audio chunk to the middleware.

Parameters
aud – Tuple[np.ndarray, int]: Audio chunk to publish formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

class wrapyfi.publishers.ros2.**ROS2PropertiesPublisher**(*name*, *out_topic*, ***kwargs*)

Bases: *ROS2Publisher*

__init__(*name*, *out_topic*, ***kwargs*)

Initialize the publisher.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **queue_size** – int: Queue size for the publisher. Default is 5
- **ros2_kwargs** – dict: Additional kwargs for the ROS 2 middleware
- **kwargs** – dict: Additional kwargs for the publisher

class wrapyfi.publishers.ros2.**ROS2MessagePublisher**(*name: str*, *out_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, ***kwargs*)

Bases: *ROS2Publisher*

__init__(*name: str*, *out_topic: str*, *should_wait: bool = True*, *queue_size: int = 5*, ***kwargs*)

The ROS2MessagePublisher using the ROS 2 message type determined dynamically.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **queue_size** – int: Queue size for the publisher. Default is 5

get_message_type(*msg*)

Get the type of a specific message.

Parameters
msg – ROS 2 message object

Returns
type: The type of the provided message

establish(*msg*, *repeats: int | None = None*, ***kwargs*)

Establish the connection using the provided message to determine the type.

Parameters

- **msg** – ROS2Message: Message to determine the type.
- **repeats** – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

publish(*msg*)

Publish the data to the middleware.

Parameters

msg – ROS2Message: Message to publish. This should be formatted according to the expected message type.

wrapyfi.publishers.yarp module

class wrapyfi.publishers.yarp.**YarpPublisher**(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *out_topic_connect: str | None = None*, *yarp_kwargs: dict | None = None*, ***kwargs*)

Bases: *Publisher*

__init__(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *out_topic_connect: str | None = None*, *yarp_kwargs: dict | None = None*, ***kwargs*)

Initialize the publisher.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **persistent** – bool: Whether the publisher port should remain connected after closure. Default is True
- **out_topic_connect** – str: Name of the output topic connection alias '/' (e.g. '/topic:out') to connect to. None appends ':out' to the out_topic. Default is None
- **yarp_kwargs** – dict: Additional kwargs for the Yarp middleware
- **kwargs** – dict: Additional kwargs for the publisher

await_connection(*port*, *out_topic: str | None = None*, *repeats: int | None = None*)

Wait for at least one subscriber to connect to the publisher.

Parameters

- **port** – yarp.Port: Port to await connection to
- **out_topic** – str: Name of the output topic
- **repeats** – int: Number of repeats to await connection. None for infinite. Default is None

Returns

bool: True if connection established, False otherwise

close()

Close the publisher

class wrapyfi.publishers.yarp.**YarpNativeObjectPublisher**(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *out_topic_connect: str | None = None*, *serializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *YarpPublisher*

__init__(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *out_topic_connect: str | None = None*, *serializer_kwargs: dict | None = None*, ***kwargs*)

The NativeObject publisher using the BufferedPortBottle string construct assuming a combination of python native objects and numpy arrays as input. Serializes the data (including plugins) using the encoder and sends it as a string.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **persistent** – bool: Whether the publisher port should remain connected after closure. Default is True
- **out_topic_connect** – str: Name of the output topic connection alias '/' (e.g. '/topic:out') to connect to. None appends ':out' to the out_topic. Default is None
- **serializer_kwargs** – dict: Additional kwargs for the serializer

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection

Parameters

repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns

bool: True if connection established, False otherwise

publish(*obj*)

Publish the object to the middleware.

Parameters

obj – object: Object to publish

class wrapyfi.publishers.yarp.**YarpImagePublisher**(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *out_topic_connect: str | None = None*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

Bases: *YarpPublisher*

__init__(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *out_topic_connect: str | None = None*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

The Image publisher using the BufferedPortImage construct assuming a numpy array as input.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **persistent** – bool: Whether the publisher port should remain connected after closure. Default is True
- **out_topic_connect** – str: Name of the output topic connection alias '/' (e.g. '/topic:out') to connect to. None appends ':out' to the out_topic. Default is None
- **width** – int: Width of the image. Default is -1 meaning the width of the input image
- **height** – int: Height of the image. Default is -1 meaning the height of the input image
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be compressed as JPG. Default is False

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection.

Parameters

repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns

bool: True if connection established, False otherwise

publish(*img: numpy.ndarray*)

Publish the image to the middleware.

Parameters

img – np.ndarray: Image to publish formatted as a cv2 image np.ndarray[img_height, img_width, channels] to publish

class wrapyfi.publishers.yarp.**YarpAudioChunkPublisher**(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *out_topic_connect: str | None = None*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

Bases: *YarpPublisher*

__init__(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *should_wait: bool = True*, *persistent: bool = True*, *out_topic_connect: str | None = None*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

The AudioChunk publisher using the Sound construct assuming a numpy array as input.

Parameters

- **name** – str: Name of the publisher

- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **persistent** – bool: Whether the publisher port should remain connected after closure. Default is True
- **out_topic_connect** – str: Name of the output topic connection alias '/' (e.g. '/topic:out') to connect to. None appends ':out' to the out_topic. Default is None
- **channels** – int: Number of channels. Default is 1
- **rate** – int: Sampling rate. Default is 44100
- **chunk** – int: Chunk size. Default is -1 meaning that the chunk size is not fixed

establish(*repeats: int | None = None*, ***kwargs*)

Establish the connection.

Parameters

repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns

bool: True if connection established, False otherwise

publish(*aud: Tuple[numpy.ndarray, int]*)

Publish the audio chunk to the middleware.

Parameters

aud – Tuple[np.ndarray, int]: Audio chunk to publish formatted as ((chunk_size, channels), samplerate)

class wrapyfi.publishers.yarp.**YarpPropertiesPublisher**(*name*, *out_topic*, ***kwargs*)

Bases: *YarpPublisher*

__init__(*name*, *out_topic*, ***kwargs*)

Initialize the publisher.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **persistent** – bool: Whether the publisher port should remain connected after closure. Default is True
- **out_topic_connect** – str: Name of the output topic connection alias '/' (e.g. '/topic:out') to connect to. None appends ':out' to the out_topic. Default is None
- **yarp_kwargs** – dict: Additional kwargs for the Yarp middleware
- **kwargs** – dict: Additional kwargs for the publisher

wrapyfi.publishers.zeromq module

class wrapyfi.publishers.zeromq.**ZeroMQPublisher**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *socket_ip: str = '127.0.0.1'*, *socket_pub_port: int = 5555*, *socket_sub_port: int = 5556*, *start_proxy_broker: bool = True*, *proxy_broker_spawn: str = 'process'*, *pubsub_monitor_topic: str = 'ZEROMQ/CONNECTIONS'*, *pubsub_monitor_listener_spawn: str | None = 'process'*, *zeromq_kwargs: dict | None = None*, ***kwargs*)

Bases: *Publisher*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *socket_ip: str = '127.0.0.1'*, *socket_pub_port: int = 5555*, *socket_sub_port: int = 5556*, *start_proxy_broker: bool = True*, *proxy_broker_spawn: str = 'process'*, *pubsub_monitor_topic: str = 'ZEROMQ/CONNECTIONS'*, *pubsub_monitor_listener_spawn: str | None = 'process'*, *zeromq_kwargs: dict | None = None*, ***kwargs*)

Initialize the publisher and start the proxy broker if necessary.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **socket_ip** – str: IP address of the socket. Default is '127.0.0.1'
- **socket_pub_port** – int: Port of the socket for publishing. Default is 5555
- **socket_sub_port** – int: Port of the socket for subscribing. Default is 5556
- **start_proxy_broker** – bool: Whether to start a proxy broker. Default is True
- **proxy_broker_spawn** – str: Whether to spawn the proxy broker as a process or thread. Default is 'process'
- **pubsub_monitor_topic** – str: Topic to monitor the connections. Default is 'ZEROMQ/CONNECTIONS'
- **pubsub_monitor_listener_spawn** – str: Whether to spawn the PUB/SUB monitor listener as a process or thread. Default is 'process'
- **zeromq_kwargs** – dict: Additional kwargs for the ZeroMQ Pub/Sub middleware
- **kwargs** – Additional kwargs for the publisher

await_connection(*out_topic: str | None = None*, *repeats: int | None = None*)

Wait for the connection to be established.

Parameters

- **out_topic** – str: Name of the output topic
- **repeats** – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

`close()`

Close the publisher.

`class wrapyfi.publishers.zeromq.ZeroMQNativeObjectPublisher(name: str, out_topic: str, carrier: str = 'tcp', should_wait: bool = True, multi_threaded: bool = False, serializer_kwargs: dict | None = None, **kwargs)`

Bases: *ZeroMQPublisher*

`__init__(name: str, out_topic: str, carrier: str = 'tcp', should_wait: bool = True, multi_threaded: bool = False, serializer_kwargs: dict | None = None, **kwargs)`

The NativeObjectPublisher using the ZeroMQ message construct assuming a combination of python native objects and numpy arrays as input. Serializes the data (including plugins) using the encoder and sends it as a string.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **multi_threaded** – bool: Whether to use a separate socket for each thread. Default is False
- **serializer_kwargs** – dict: Additional kwargs for the serializer
- **kwargs** – dict: Additional kwargs for the publisher

`establish(repeats: int | None = None, **kwargs)`

Establish the connection to the publisher.

Parameters
repeats – int: Number of repeats to await connection. None for infinite. Default is None

Returns
bool: True if connection established, False otherwise

`publish(obj)`

Publish the object to the middleware.

Parameters
obj – object: Object to publish

`class wrapyfi.publishers.zeromq.ZeroMQImagePublisher(name: str, out_topic: str, carrier: str = 'tcp', should_wait: bool = True, multi_threaded: bool = False, width: int = -1, height: int = -1, rgb: bool = True, fp: bool = False, jpg: bool = False, **kwargs)`

Bases: *ZeroMQNativeObjectPublisher*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *multi_threaded: bool = False*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, ***kwargs*)

The ImagePublisher using the ZeroMQ message construct assuming a numpy array as input.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **multi_threaded** – bool: Whether to use a separate socket for each thread. Default is False
- **width** – int: Width of the image. Default is -1 meaning that the width is not fixed
- **height** – int: Height of the image. Default is -1 meaning that the height is not fixed
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be compressed as JPG. Default is False

publish(*img: numpy.ndarray*)

Publish the image to the middleware.

Parameters

img – np.ndarray: Image to publish formatted as a cv2 image np.ndarray[img_height, img_width, channels]

class wrapyfi.publishers.zeromq.**ZeroMQAudioChunkPublisher**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *multi_threaded: bool = False*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

Bases: *ZeroMQNativeObjectPublisher*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *should_wait: bool = True*, *multi_threaded: bool = False*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, ***kwargs*)

The AudioChunkPublisher using the ZeroMQ message construct assuming a numpy array as input.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **multi_threaded** – bool: Whether to use a separate socket for each thread. Default is False
- **channels** – int: Number of channels. Default is 1

- **rate** – int: Sampling rate. Default is 44100
- **chunk** – int: Chunk size. Default is -1 meaning that the chunk size is not fixed

publish(*aud: Tuple[numpy.ndarray, int]*)

Publish the audio chunk to the middleware.

Parameters

aud – Tuple[np.ndarray, int]: Audio chunk to publish formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

class wrapyfi.publishers.zeromq.**ZeroMQPropertiesPublisher**(*name*, *out_topic*, ***kwargs*)

Bases: *ZeroMQPublisher*

__init__(*name*, *out_topic*, ***kwargs*)

Initialize the publisher and start the proxy broker if necessary.

Parameters

- **name** – str: Name of the publisher
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **should_wait** – bool: Whether to wait for at least one listener before unblocking the script. Default is True
- **socket_ip** – str: IP address of the socket. Default is '127.0.0.1'
- **socket_pub_port** – int: Port of the socket for publishing. Default is 5555
- **socket_sub_port** – int: Port of the socket for subscribing. Default is 5556
- **start_proxy_broker** – bool: Whether to start a proxy broker. Default is True
- **proxy_broker_spawn** – str: Whether to spawn the proxy broker as a process or thread. Default is 'process'
- **pubsub_monitor_topic** – str: Topic to monitor the connections. Default is 'ZEROMQ/CONNECTIONS'
- **pubsub_monitor_listener_spawn** – str: Whether to spawn the PUB/SUB monitor listener as a process or thread. Default is 'process'
- **zeromq_kwargs** – dict: Additional kwargs for the ZeroMQ Pub/Sub middleware
- **kwargs** – Additional kwargs for the publisher

Module contents

class wrapyfi.publishers.**FallbackPublisher**(*name: str*, *out_topic: str*, *carrier: str = ''*, *should_wait: bool = True*, *missing_middleware_object: str = ''*, ***kwargs*)

Bases: *Publisher*

__init__(*name: str*, *out_topic: str*, *carrier: str = ''*, *should_wait: bool = True*, *missing_middleware_object: str = ''*, ***kwargs*)

Initialize the Publisher.

Parameters

- **name** – str: The name of the publisher

- **out_topic** – str: The name of the output topic
- **carrier** – str: The name of the carrier to use
- **should_wait** – bool: Whether to wait for the publisher to be established or not

establish(*repeats: int = -1*, ***kwargs*)

Establish the publisher.

publish(*obj*)

Publish an object.

close()

Close the connection.

wrapyfi.servers package

Submodules

wrapyfi.servers.ros module

class wrapyfi.servers.ros.**ROSServer**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *ros_kwargs: dict | None = None*, ***kwargs*)

Bases: *Server*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *ros_kwargs: dict | None = None*, ***kwargs*)

Initialize the server.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for REQ/REP pattern. Default is 'tcp'
- **ros_kwargs** – dict: Additional kwargs for the ROS middleware
- **kwargs** – dict: Additional kwargs for the server

close()

Close the server.

class wrapyfi.servers.ros.**ROSNativeObjectServer**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROSServer*

SEND_QUEUE = <queue.Queue object>

RECEIVE_QUEUE = <queue.Queue object>

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling native Python objects, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the server

- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for REQ/REP pattern. Default is 'tcp'
- **serializer_kwargs** – dict: Additional kwargs for the serializer
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the connection to the server.

await_request(**args*, ***kwargs*)

Await and deserialize the client's request, returning the extracted arguments and keyword arguments. The method blocks until a message is received, then attempts to deserialize it using the configured JSON decoder hook, returning the extracted arguments and keyword arguments.

Returns

Tuple[list, dict]: A tuple containing two items: - A list of arguments extracted from the received message - A dictionary of keyword arguments extracted from the received message

reply(*obj*)

Serialize the provided object and send it as a reply to the client.

Parameters

obj – Any: The Python object to be serialized and sent

class wrapyfi.servers.ros.**ROSImageServer**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROSServer*

SEND_QUEUE = <queue.Queue object>

RECEIVE_QUEUE = <queue.Queue object>

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling native Python objects, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for REQ/REP pattern. Default is 'tcp'
- **width** – int: Width of the image. Default is -1 (use the width of the received image)
- **height** – int: Height of the image. Default is -1 (use the height of the received image)
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the connection to the server.

await_request(**args*, ***kwargs*)

Await and deserialize the client's request, returning the extracted arguments and keyword arguments. The method blocks until a message is received, then attempts to deserialize it using the configured JSON decoder hook, returning the extracted arguments and keyword arguments.

Returns

Tuple[list, dict]: A tuple containing two items: - A list of arguments extracted from the received message - A dictionary of keyword arguments extracted from the received message

reply(*img: numpy.ndarray*)

Serialize the provided image and send it as a reply to the client.

Parameters

img – np.ndarray: Image to publish

class wrapyfi.servers.ros.**ROSAudioChunkServer**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROSServer*

SEND_QUEUE = <queue.Queue object>

RECEIVE_QUEUE = <queue.Queue object>

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling audio data as numpy arrays.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol. ROS currently only supports TCP for REQ/REP pattern. Default is 'tcp'
- **channels** – int: Number of channels in the audio. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: Number of samples in the audio chunk. Default is -1 (use the chunk size of the received audio)
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the connection to the server.

await_request(**args*, ***kwargs*)

Await and deserialize the client's request, returning the extracted arguments and keyword arguments. The method blocks until a message is received, then attempts to deserialize it using the configured JSON decoder hook, returning the extracted arguments and keyword arguments.

Returns

Tuple[list, dict]: A tuple containing two items: - A list of arguments extracted from the received message - A dictionary of keyword arguments extracted from the received message

reply(*aud: Tuple[numpy.ndarray, int]*)

Serialize the provided audio data and send it as a reply to the client.

Parameters
aud – Tuple[np.ndarray, int]: Audio chunk to publish formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

wrapyfi.servers.ros2 module

class wrapyfi.servers.ros2.**ROS2Server**(*name: str*, *out_topic: str*, *ros2_kwargs: dict | None = None*, ***kwargs*)

Bases: *Server*, Node

__init__(*name: str*, *out_topic: str*, *ros2_kwargs: dict | None = None*, ***kwargs*)

Initialize the server.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **ros2_kwargs** – dict: Additional kwargs for the ROS 2 middleware
- **kwargs** – dict: Additional kwargs for the server

close()

Close the server.

class wrapyfi.servers.ros2.**ROS2NativeObjectServer**(*name: str*, *out_topic: str*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROS2Server*

SEND_QUEUE = <queue.Queue object>

RECEIVE_QUEUE = <queue.Queue object>

__init__(*name: str*, *out_topic: str*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling native Python objects, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **serializer_kwargs** – dict: Additional kwargs for the serializer
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the connection to the server

await_request(**args*, ***kwargs*)

Await and deserialize the client's request, returning the extracted arguments and keyword arguments. The method blocks until a message is received, then attempts to deserialize it using the configured JSON decoder hook, returning the extracted arguments and keyword arguments.

Returns
Tuple[list, dict]: A tuple containing two items: - A list of arguments extracted from the received message - A dictionary of keyword arguments extracted from the received message

reply(*obj*)

Serialize the provided Python object to a JSON string and send it as a reply to the client. The method uses the configured JSON encoder for serialization before sending the resultant string to the client.

Parameters
obj – Any: The Python object to be serialized and sent

class wrapyfi.servers.ros2.**ROS2ImageServer**(*name: str*, *out_topic: str*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROS2Server*

SEND_QUEUE = <queue.Queue object>

RECEIVE_QUEUE = <queue.Queue object>

__init__(*name: str*, *out_topic: str*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling native Python objects, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **width** – int: Width of the image. Default is -1 (use the width of the received image)
- **height** – int: Height of the image. Default is -1 (use the height of the received image)
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be decompressed from JPG. Default is False
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the connection to the server.

await_request(**args*, ***kwargs*)

Await and deserialize the client's request, returning the extracted arguments and keyword arguments. The method blocks until a message is received, then attempts to deserialize it using the configured JSON decoder hook, returning the extracted arguments and keyword arguments.

Returns
Tuple[list, dict]: A tuple containing two items: - A list of arguments extracted from the received message - A dictionary of keyword arguments extracted from the received message

reply(*img: numpy.ndarray*)

Serialize the provided image data and send it as a reply to the client.

Parameters
img – np.ndarray: Image to send formatted as a cv2 image - np.ndarray[img_height, img_width, channels]

class wrapyfi.servers.ros2.**ROS2AudioChunkServer**(*name: str*, *out_topic: str*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ROS2Server*

SEND_QUEUE = <queue.Queue object>

RECEIVE_QUEUE = <queue.Queue object>

__init__(*name: str*, *out_topic: str*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling audio data as numpy arrays.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **channels** – int: Number of channels in the audio. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: Number of samples in the audio chunk. Default is -1 (use the chunk size of the received audio)
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the connection to the server.

await_request(**args*, ***kwargs*)

Await and deserialize the client's request, returning the extracted arguments and keyword arguments. The method blocks until a message is received, then attempts to deserialize it using the configured JSON decoder hook, returning the extracted arguments and keyword arguments.

Returns

Tuple[list, dict]: A tuple containing two items: - A list of arguments extracted from the received message - A dictionary of keyword arguments extracted from the received message

reply(*aud: Tuple[numpy.ndarray, int]*)

Serialize the provided audio data and send it as a reply to the client.

Parameters

aud – Tuple[np.ndarray, int]: Audio chunk to publish formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

wrapyfi.servers.yarp module

class wrapyfi.servers.yarp.**YarpServer**(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *out_topic_connect: str | None = None*, *persistent: bool = True*, *yarp_kwargs: dict | None = None*, ***kwargs*)

Bases: *Server*

__init__(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *out_topic_connect: str | None = None*, *persistent: bool = True*, *yarp_kwargs: dict | None = None*, ***kwargs*)

Initialize the server.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'

- **out_topic_connect** – str: Name of the output topic connection alias '/' (e.g. '/topic:out') to connect to. None appends ':out' to the out_topic. Default is None
- **yarp_kwargs** – dict: Additional kwargs for the Yarp middleware
- **kwargs** – dict: Additional kwargs for the server

close()

Close the server.

class wrapyfi.servers.yarp.**YarpNativeObjectServer**(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *out_topic_connect: str | None = None*, *persistent: bool = True*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *YarpServer*

__init__(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *out_topic_connect: str | None = None*, *persistent: bool = True*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling native Python objects, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **out_topic_connect** – str: Name of the output topic connection alias '/' (e.g. '/topic:out') to connect to. None appends ':out' to the out_topic. Default is None
- **persistent** – bool: Whether the server port should remain connected after closure. Default is True
- **serializer_kwargs** – dict: Additional kwargs for the serializer
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish()

Establish the connection to the server.

await_request(**args*, ***kwargs*)

Await and deserialize the client's request, returning the extracted arguments and keyword arguments. The method blocks until a message is received, then attempts to deserialize it using the configured JSON decoder hook, returning the extracted arguments and keyword arguments.

Returns

Tuple[list, dict]: A tuple containing two items: - A list of arguments extracted from the received message - A dictionary of keyword arguments extracted from the received message

reply(*obj*)

Serialize the provided Python object to a JSON string and send it as a reply to the client. The method uses the configured JSON encoder for serialization before sending the resultant string to the client.

Parameters

obj – Any: The Python object to be serialized and sent

class wrapyfi.servers.yarp.**YarpImageServer**(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *out_topic_connect: str | None = None*, *persistent: bool = True*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *YarpNativeObjectServer*

__init__(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *out_topic_connect: str | None = None*, *persistent: bool = True*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling image data as numpy arrays, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'
- **out_topic_connect** – str: Name of the output topic connection alias '/' (e.g. '/topic:out') to connect to. None appends ':out' to the out_topic. Default is None
- **persistent** – bool: Whether the server port should remain connected after closure. Default is True
- **width** – int: Width of the image. Default is -1 (use the width of the received image)
- **height** – int: Height of the image. Default is -1 (use the height of the received image)
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

reply(*img: numpy.ndarray*)

Serialize the provided image data and send it as a reply to the client.

Parameters

img – np.ndarray: Image to send formatted as a cv2 image - np.ndarray[img_height, img_width, channels]

class wrapyfi.servers.yarp.**YarpAudioChunkServer**(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *out_topic_connect: str | None = None*, *persistent: bool = True*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *YarpNativeObjectServer*

__init__(*name: str*, *out_topic: str*, *carrier: Literal['tcp', 'udp', 'mcast'] = 'tcp'*, *out_topic_connect: str | None = None*, *persistent: bool = True*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling audio data as numpy arrays, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Name of the output topic preceded by '/' (e.g. '/topic')
- **carrier** – str: Carrier protocol (e.g. 'tcp'). Default is 'tcp'

- **out_topic_connect** – str: Name of the output topic connection alias '/' (e.g. '/topic:out') to connect to. None appends ':out' to the out_topic. Default is None
- **persistent** – bool: Whether the server port should remain connected after closure. Default is True
- **channels** – int: Number of channels in the audio. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: Number of samples in the audio chunk. Default is -1 (use the chunk size of the received audio)
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

reply(*aud: Tuple[numpy.ndarray, int]*)

Serialize the provided audio data and send it as a reply to the client.

Parameters

aud – Tuple[np.ndarray, int]: Audio chunk to publish formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

wrapyfi.servers.zeromq module

class wrapyfi.servers.zeromq.**ZeroMQServer**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *socket_ip: str = '127.0.0.1'*, *socket_rep_port: int = 5558*, *socket_req_port: int = 5559*, *start_proxy_broker: bool = True*, *proxy_broker_spawn: bool = 'process'*, *zeromq_kwargs: dict | None = None*, ***kwargs*)

Bases: *Server*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *socket_ip: str = '127.0.0.1'*, *socket_rep_port: int = 5558*, *socket_req_port: int = 5559*, *start_proxy_broker: bool = True*, *proxy_broker_spawn: bool = 'process'*, *zeromq_kwargs: dict | None = None*, ***kwargs*)

Initialize the server and start the device broker if necessary.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Topics are not supported for the REQ/REP pattern in ZeroMQ. Any given topic is ignored
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for REQ/REP pattern. Default is 'tcp'
- **socket_ip** – str: IP address of the socket. Default is '127.0.0.1'
- **socket_rep_port** – int: Port of the socket for REP pattern. Default is 5558
- **socket_req_port** – int: Port of the socket for REQ pattern. Default is 5559
- **start_proxy_broker** – bool: Whether to start a device broker. Default is True
- **proxy_broker_spawn** – str: Whether to spawn the device broker as a process or thread. Default is 'process'
- **zeromq_kwargs** – dict: Additional kwargs for the ZeroMQ Req/Rep middleware
- **kwargs** – dict: Additional kwargs for the server

close()

Close the server.

class wrapyfi.servers.zeromq.**ZeroMQNativeObjectServer**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ZeroMQServer*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *serializer_kwargs: dict | None = None*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling native Python objects, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Topics are not supported for the REQ/REP pattern in ZeroMQ. Any given topic is ignored
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for PUB/SUB pattern. Default is 'tcp'
- **serializer_kwargs** – dict: Additional kwargs for the serializer
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

establish(***kwargs*)

Establish the connection to the server.

await_request(**args*, ***kwargs*)

Await and deserialize the client's request, returning the extracted arguments and keyword arguments. The method blocks until a message is received, then attempts to deserialize it using the configured JSON decoder hook, returning the extracted arguments and keyword arguments.

Returns

Tuple[list, dict]: A tuple containing two items: - A list of arguments extracted from the received message - A dictionary of keyword arguments extracted from the received message

reply(*obj*)

Serialize the provided Python object to a JSON string and send it as a reply to the client. The method uses the configured JSON encoder for serialization before sending the resultant string to the client.

Parameters

obj – Any: The Python object to be serialized and sent

class wrapyfi.servers.zeromq.**ZeroMQImageServer**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ZeroMQNativeObjectServer*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *width: int = -1*, *height: int = -1*, *rgb: bool = True*, *fp: bool = False*, *jpg: bool = False*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling image data as numpy arrays, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the server

- **out_topic** – str: Topics are not supported for the REQ/REP pattern in ZeroMQ. Any given topic is ignored
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for REQ/REP pattern. Default is 'tcp'
- **width** – int: Width of the image. Default is -1 (use the width of the received image)
- **height** – int: Height of the image. Default is -1 (use the height of the received image)
- **rgb** – bool: True if the image is RGB, False if it is grayscale. Default is True
- **fp** – bool: True if the image is floating point, False if it is integer. Default is False
- **jpg** – bool: True if the image should be decompressed from JPG. Default is False
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

reply(*img: numpy.ndarray*)

Serialize the provided image data and send it as a reply to the client.

Parameters

img – np.ndarray: Image to send formatted as a cv2 image - np.ndarray[img_height, img_width, channels]

class wrapyfi.servers.zeromq.**ZeroMQAudioChunkServer**(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Bases: *ZeroMQNativeObjectServer*

__init__(*name: str*, *out_topic: str*, *carrier: str = 'tcp'*, *channels: int = 1*, *rate: int = 44100*, *chunk: int = -1*, *deserializer_kwargs: dict | None = None*, ***kwargs*)

Specific server handling audio data as numpy arrays, serializing them to JSON strings for transmission.

Parameters

- **name** – str: Name of the server
- **out_topic** – str: Topics are not supported for the REQ/REP pattern in ZeroMQ. Any given topic is ignored
- **carrier** – str: Carrier protocol. ZeroMQ currently only supports TCP for REQ/REP pattern. Default is 'tcp'
- **channels** – int: Number of channels in the audio. Default is 1
- **rate** – int: Sampling rate of the audio. Default is 44100
- **chunk** – int: Number of samples in the audio chunk. Default is -1 (use the chunk size of the received audio)
- **deserializer_kwargs** – dict: Additional kwargs for the deserializer

reply(*aud: Tuple[numpy.ndarray, int]*)

Serialize the provided audio data and send it as a reply to the client.

Parameters

aud – Tuple[np.ndarray, int]: Audio chunk to publish formatted as (np.ndarray[audio_chunk, channels], int[samplerate])

Module contents

class wrapyfi.servers.**FallbackServer**(*name: str*, *out_topic: str*, *carrier: str = ''*, *missing_middleware_object: str = ''*, ***kwargs*)

Bases: *Server*

__init__(*name: str*, *out_topic: str*, *carrier: str = ''*, *missing_middleware_object: str = ''*, ***kwargs*)

Initialize the server.

Parameters

- **name** – str: The name of the server
- **out_topic** – str: The topic to publish to
- **carrier** – str: The middleware carrier to use
- **out_topic_connect** – str: The topic to connect to (this is deprecated and will be removed in the future since its usage is limited to YARP)

establish(*repeats: int = -1*, ***kwargs*)

Establish the server.

await_request(**args*, ***kwargs*)

Await a request from a client.

reply(*obj*)

Reply to a client request.

close()

Close the connection.

wrapyfi.standalone package

Submodules

wrapyfi.standalone.zeromq_param_server module

wrapyfi.standalone.zeromq_param_server.**access_nested_dict**(*d*, *keys*)

wrapyfi.standalone.zeromq_param_server.**parse_prefix**(*param_full: str*, *topics: dict*)

wrapyfi.standalone.zeromq_param_server.**reverse_parse_prefix**(*topics: dict*, *prefix: str = ''*)

wrapyfi.standalone.zeromq_param_server.**main**(*role*, *param_ip*, *param_pub_port*, *param_sub_port*, *param_reqrep_port*, *param_poll_interval*, *param_poll_repeat*, ***kwargs*)

wrapyfi.standalone.zeromq_param_server.**parse_args**()

wrapyfi.standalone.zeromq_proxy_broker module

wrapyfi.standalone.zeromq_proxy_broker.**main**(*comm_type*, *socket_ip*, *socket_pub_port*, *socket_sub_port*, *socket_rep_port*, *socket_req_port*, ***kwargs*)

wrapyfi.standalone.zeromq_proxy_broker.**parse_args**()

wrapyfi.standalone.zeromq_pubsub_topic_monitor module

wrapyfi.standalone.zeromq_pubsub_topic_monitor.**monitor_active_connections**(*socket_pub_address*, *topic*)

wrapyfi.standalone.zeromq_pubsub_topic_monitor.**parse_args**()

Module contents

wrapyfi.tests package

Subpackages

wrapyfi.tests.tools package

Submodules

wrapyfi.tests.tools.benchmarking_native_object module

class wrapyfi.tests.tools.benchmarking_native_object.**Benchmarker**

Bases: *MiddlewareCommunicator*

static get_numpy_object(*dims*)

static get_pandas_object(*dims*)

static get_pillow_object(*dims*)

static get_tensorflow_object(*dims*)

static get_jax_object(*dims*)

static get_mxnet_object(*dims*)

static get_mxnet_gpu_object(*dims*, *gpu=0*)

static get_pytorch_object(*dims*)

static get_pytorch_gpu_object(*dims*, *gpu=0*)

static get_paddle_object(*dims*)

static get_paddle_gpu_object(*dims*, *gpu=0*)

get_all_objects(*count*, *plugin_name*)

get_yarp_native_objects(*count*, *plugin_name*)

get_ros_native_objects(*count*, *plugin_name*)

get_ros2_native_objects(*count*, *plugin_name*)

get_zeromq_native_objects(*count*, *plugin_name*)

wrapyfi.tests.tools.benchmarking_native_object.**parse_args**()

wrapyfi.tests.tools.class_test module

Module contents

Submodules

wrapyfi.tests.test_middleware module

class wrapyfi.tests.test_middleware.**ZeroMQTestMiddleware**(*methodName='runTest'*)

Bases: TestCase

MWARE = 'zeromq'

test_publish_listen()

Test the publish and listen functionality of the middleware. This verifies that the middleware can send and receive messages using the PUB/SUB pattern.

class wrapyfi.tests.test_middleware.**ROS2TestMiddleware**(*methodName='runTest'*)

Bases: *ZeroMQTestMiddleware*

Test the ROS 2 wrapper. This test class inherits from the ZeroMQ test class, so all tests from the ZeroMQ test class are also run for the ROS 2 wrapper.

MWARE = 'ros2'

class wrapyfi.tests.test_middleware.**YarpTestMiddleware**(*methodName='runTest'*)

Bases: *ZeroMQTestMiddleware*

Test the YARP wrapper. This test class inherits from the ZeroMQ test class, so all tests from the ZeroMQ test class are also run for the YARP wrapper.

MWARE = 'yarp'

class wrapyfi.tests.test_middleware.**ROSTestMiddleware**(*methodName='runTest'*)

Bases: *ZeroMQTestMiddleware*

Test the ROS wrapper. This test class inherits from the ZeroMQ test class, so all tests from the ZeroMQ test class are also run for the ROS wrapper.

MWARE = 'ros'

wrapyfi.tests.test_wrapper module

class wrapyfi.tests.test_wrapper.**ZeroMQTestWrapper**(*methodName='runTest'*)

Bases: TestCase

MWARE = 'zeromq'

test_activate_communication()

Test the activate communication functionality of the middleware. When the `activate_communication` method is called, the `__WRAPYFI_INSTANCES` attribute of the decorated function should be updated with the instance of the class that called the method. Destroying the class instance should remove the instance from the list and the the connection should no longer be active. The `activate_communication` method should also be callable multiple times on the same function.

test_close()

Test the close functionality of the middleware. When the `close` method is called, the instances should be reordered in the `__WRAPYFI_INSTANCES` attribute of the decorated function. Destroying the class instance should remove the instance from the list and the index of the connection should change. The `close` method should also be callable multiple times on different instances of a class.

test_get_communicators()

Test the get communicators functionality of the middleware. When the `get_communicators` method is called, the method should return a list of all available middleware wrappers.

class wrapyfi.tests.test_wrapper.**ROS2TestWrapper**(*methodName='runTest'*)

Bases: *ZeroMQTestWrapper*

Test the ROS 2 wrapper. This test class inherits from the ZeroMQ test class, so all tests from the ZeroMQ test class are also run for the ROS 2 wrapper.

MWARE = 'ros2'

class wrapyfi.tests.test_wrapper.**YarpTestWrapper**(*methodName='runTest'*)

Bases: *ZeroMQTestWrapper*

Test the YARP wrapper. This test class inherits from the ZeroMQ test class, so all tests from the ZeroMQ test class are also run for the YARP wrapper.

MWARE = 'yarp'

class wrapyfi.tests.test_wrapper.**ROSTestWrapper**(*methodName='runTest'*)

Bases: *ZeroMQTestWrapper*

Test the ROS wrapper. This test class inherits from the ZeroMQ test class, so all tests from the ZeroMQ test class are also run for the ROS wrapper.

MWARE = 'ros'

Module contents

23.1.2 Submodules

23.1.3 wrapyfi.encoders module

23.1.4 wrapyfi.utils module

wrapyfi.utils.**deepcopy**(*obj: Any*, *exclude_keys: list | tuple | None = None*, *shallow_keys: list | tuple | None = None*)

Deep copy an object, excluding specified keys.

Parameters

- **obj** – Any: The object to deep copy
- **exclude_keys** – Union[list, tuple]: A list of keys to exclude from the deep copy
- **shallow_keys** – Union[list, tuple]: A list of keys to shallow copy

wrapyfi.utils.**get_default_args**(*fnc: Callable[[...], Any]*)

Get the default arguments for a function.

Parameters

fnc – Callable[..., Any]: The function to get the default arguments for

wrapyfi.utils.**match_args**(*args: list | tuple*, *kwargs: dict*, *src_args: list | tuple*, *src_kwargs: dict*)

Match and Substitute Arguments and Keyword Arguments using Specified Source Values.

Navigate through the provided *args* and *kwargs*, identifying entries prefixed with "$" and substituting them with values from *src_args* and *src_kwargs* respectively, to dynamically modify the function call parameters using the source values.

Parameters

- **args** – Union[list, tuple]: A list of arguments, potentially containing strings that indicate substitutable entries. Substitutable entries are prefixed with "$" and followed by either:
  - A digit (indicating an index to reference a value from *src_args*), or
  - Non-digit characters (indicating a key to reference a value from *src_kwargs*).
- **kwargs** – dict: A dictionary of keyword arguments, where values might be strings indicating substitutable entries, similar to the entries in *args*.
- **src_args** – Union[list, tuple]: A list of source arguments, intended to be referenced by substitutable entries within *args*.
- **src_kwargs** – dict: A dictionary of source keyword arguments, intended to be referenced by substitutable entries within *args* and *kwargs*.

Returns

Tuple[list, dict]: A tuple containing: - list: The new arguments, formed by substituting specified entries from *args* using *src_args* and *src_kwargs*. - dict: The new keyword arguments, formed by substituting specified entries from *kwargs* using *src_args* and *src_kwargs*.

wrapyfi.utils.**dynamic_module_import**(*modules: List[str]*, *globals: dict*)

Dynamically import modules.

Parameters

- **modules** – List[str]: A list of module names to import
- **globals** – dict: The globals dictionary to update

class wrapyfi.utils.**SingletonOptimized**

Bases: type

A singleton metaclass that is thread-safe and optimized for speed.

Source: https://stackoverflow.com/a/6798042

class wrapyfi.utils.**Plugin**

Bases: object

Base class for encoding and decoding plugins.

encode(**args*, ***kwargs*)

Encode data into a base64 string.

Parameters

- **args** – tuple: Additional arguments
- **kwargs** – dict: Additional keyword arguments

Returns

Tuple[bool, dict]: A tuple containing: - bool: True if the encoding was successful, False otherwise - dict: A dictionary containing:

- '__wrapyfi__': A tuple containing the class name and encoded data string

decode(**args*, ***kwargs*)

Decode a base64 string back into data.

Parameters

- **args** – tuple: Additional arguments
- **kwargs** – dict: Additional keyword arguments

Returns

Tuple[bool, object]: A tuple containing: - bool: True if the decoding was successful, False otherwise - object: The decoded data

class wrapyfi.utils.**PluginRegistrar**

Bases: object

Class for registering encoding and decoding plugins.

encoder_registry = {}

decoder_registry = {'AstropyData': <class 'examples.encoders.plugins.astropy_tables.AstropyData'>}

static register(*types=None*)

Register a plugin for encoding and decoding a specific type.

Parameters

types – tuple: The type(s) to register the plugin for

static scan()

Scan the plugins directory (Wrapyfi builtin and external) for plugins to register. This method is called automatically when the module is imported.

23.1.5 Module contents

unknown_url

wrapyfi.**get_project_info_from_setup**()

PYTHON MODULE INDEX

W

INDEX

Symbols

A

B

C

D

E

M

P

R

S

T

U

W

X

Y

Z