ZeROSMQ Documentation

Release 1.0

jreyesr

Apr 17, 2019

Contents:

ZeROSMQ is a Python library that implements ROS-like inter-process communication by using topics and nodes. The communications are handled by ZeroMQ in the most inefficient way possible.

ZeROSMQ allows Python programs to pass messages to each other, while at the same time decoupling the implementation of the different programs. It is especially suited for data collection applications, where a program can read data from a sensor and broadcast it in real time, so that other, independent programs can receive and process it.

ZeROSMQ's basic usage is as follows:

```
# In data publishers
import zerosmq
zerosmq.init("publisher")
zerosmq.register_publish("SOME_TOPIC") # Report intention to publish on topic SOME_
↪TOPIC
while True:
  data=read_some_sensor()
  zerosmq.publish("SOME_TOPIC",data) # Send the sensor data to the topic
  sleep(1) # Sleep for a second


# In data subscribers
import zerosmq
zerosmq.init("subscriber")
zerosmq.subscribe("SOME_TOPIC") # Subscribe to SOME_TOPIC to receive any messages␣
↪sent to it
while True:
  data=zerosmq.receive("SOME_TOPIC") # Try to receive an update
  if data: # If data is not None, then show it
    print(data)
```

ZeROSMQ also contains some utilities that aid in debugging and using the library:

- A program that shows a graph of all nodes and how they communicate (see *Graphing nodes* for more information).
- A program that plots the data sent to a topic or group of topics (see *Plotting data* for more information).

ZeROSMQ's code is hosted on GitHub.

CHAPTER 1

Basics

Just like in ROS, a "node" is a program (in ZeROSMQ, a Python script) that communicates with other nodes. A "topic" is a pipe over which nodes can exchange information. Topics follow the publisher-subscriber pattern. Many nodes can request permission to publish on a certain topic, and many nodes can subscribe to the same topic, which enables them to receive any messages sent by any publisher on that topic. All communications are handled by a "master" node, running on a well-known port, which must be running before any other nodes.

For installation instructions, jump *here*. For usage instructions, jump *here*. If you want to see an example, go *here*.

CHAPTER 2

Installation

0. ZeROSMQ has been tested on Python 3.7.1, but it *should* also work on Python 3.6 (earlier versions won't work because ZeROSMQ uses f-strings).

1. ZeROSMQ requires pyzmq. If you wish to use *the graphing tool*, you also need to install graphviz. If you wish to use *the live plotter*, you also need to install pyqtgraph. It is highly recommended to install GNU Parallel for sanity-preservation purposes: you can run all nodes with a single shell script, in parallel, get all outputs in the same window, and close them with a single `Ctrl+C`. See *the "Shell script"* section for details on the recommended shell script.

2. Copy the `zerosmq.py` and `zerosmq-master.py` files to a folder of your choice. If you want to graph the nodes, also copy `drawmap.py`. If you want to plot any topic, also copy `plotdata.py`.

3. Add any nodes that you want to the same folder. See *Usage* for the code that you should use in the nodes.

4. Run the master node (`python zerosmq-master.py` or similar) and any other nodes (`python <your-node>.py`). If you have *GNU Parallel and a shell script*, execute `./run.sh` instead (assuming your shell script is called `run.sh`).

CHAPTER 3

Usage

This section documents the usage of ZeROSMQ's functions. See below for a list of capabilities.

3.1 Initialization

You will need to `import zerosmq` in your nodes. After that, you should call `zerosmq.init(LABEL)`, replacing LABEL by a string which will appear on any message sent by that node to `log(msg)`. LABEL should be different for each node.

Note: You can use `sys.argv[0]` for LABEL, which will tag messages with the filename of the script (note that this will not work as intended if you run multiple instances of the same script in parallel).

3.2 Publishing

After registering for publication on a specific topic with `zerosmq.register_publish(topic)`, you may use `zerosmq.publish(topic,message)` to send a message to all nodes that subscribed to that topic. The same warnings about single words on topics apply here. Messages must be strings (there is no automatic type conversion like in ROS, or at least not yet). Messages can have spaces, though. There is no length limitation.

Warning: If you attempt to call `zerosmq.publish(topic,message)` before calling `zerosmq.register_publish(topic)`, the node will stop execution because of an AssertionError.

3.3 Subscribing

After a node has subscribed to a topic with `zerosmq.subscribe(topic)`, it can call `zerosmq.receive(topic)` to receive one update from that topic, if an update is available. If there are more updates pending, you must call `zerosmq.receive(topic)` to get them, one at a time. If there are no updates available for that topic, `zerosmq.receive(topic)` will return None, which will signal that there are no messages left at the moment.

If you want to receive a message from *any* topic (maybe you just subscribed to a single topic, or you don't care which topic you receive, maybe for a logger) you may use `zerosmq.receive_any()`. This function checks all subscribed topics and returns the first message that it finds, if any. If none of the subscribed topics have a new message, it returns None.

Warning: If you attempt to call `zerosmq.receive(topic)` before calling `zerosmq.subscribe(topic)`, the node will also stop execution, just as if you attempted to publish before registering.

3.4 Registering for publications and subscribing

You need to register for publications before you can publish on a certain topic. To do so, call `zerosmq.register_publish("SOME_TOPIC")`, replacing SOME_TOPIC by a more appropriate label.

Warning: Labels must be a single word!

If your label has spaces, replace them by underscores or delete them. In its present form, ZeROSMQ makes no effort whatsoever to ensure that topics have a single word, and ZeroMQ expects topics to have a single word, so it is your job to ensure it.

You also need to subscribe to a topic before you receive updates for it. To do so, call `zerosmq.subscribe("SOME_TOPIC")`, replacing SOME_TOPIC with the desired topic. The same warning applies regarding whitespace and single words.

Note: it is recommended to register for all publications first, then wait for a while, and then subscribe. This gives the master time to process all registrations. You can do so, for example, by calling `import time` and then calling `time.sleep(1)` between `zerosmq.register_publish()` calls and `zerosmq.subscribe()` calls (obviously, you must order them so that `zerosmq.register_publish()` calls appear first). This sleeps the node for one second and gives any other nodes time to announce their own publications.

CHAPTER 4

Shell script

If you installed GNU Parallel, you can run all nodes from a single shell script, and get all their outputs on the same console. This saves time and effort, since you would typically have multiple consoles open, one for each node. This route will probably not work if a node needs input or otherwise interacts with the user, but it works fine for the standard "data collection into data processing into data output/logging" workflow that tends to happen in ROS.

To run all nodes from a single script, do the following:

1. Create a shell script. Name it however you want (this documentation will use `run.sh`)
2. Write the following line in the script:

```
parallel --ungroup --jobs 0 python ::: ../zerosmq-master.py <your-first-node.py>
↪<your-second-node.py> <your-third-node.py>
```

3. (Of course, you should change `<your-first-node.py>` for your actual first node, and so on).
4. There is no limit to the number of nodes (well, there probably is, but I haven't found it yet). You may need to change the Python interpreter to `python3.7` or similar, if you have more interpreters.
5. All nodes will run on a single console window. The output from all nodes will appear on it (that's why you should *really* use the `log(msg)` function provided with the library, instead of `print()`). If you press `Ctrl+C`, all nodes will be stopped, including the master node.

CHAPTER 5

Technical details

Note: (Skipping this section is heavily recommended. Read at your own risk)

ZeROSMQ uses ZeroMQ and its implementation of PUB-SUB sockets to build a publisher-subscriber system. The master runs a control service on TCP port 5555 by default. Any other nodes connect to it and request permission to publish or subscribe to a topic. The master responds by creating dedicated sockets for every topic on a random dynamic port, if required, and sending its port number to the requester.

When a node first requests permission to publish on a certain topic, the master creates a SUB socket and sends its port number to the node. The node creates a PUB socket, which allows it to send publications to the node. If another node requests permission to publish on the same node, the master reuses the same socket (ZeroMQ allows multiple processes to share a socket).

Warning: The master expects that no nodes will subscribe to a topic before at least one node has registered for publishing on it. If this rule is not respected, bad things may happen. This is the reason for the 1-second delay recommended *here*.

When a node first requests permission to subscribe to a topic, the master creates a PUB socket and sends its port number to the node. The node creates a SUB socket, which enables it to receive messages. The same reusing of sockets happens here.

The master spends almost all its time (except when it's attending a request from a node) scanning the control socket for requests and all its open SUB sockets (which correspond to PUB sockets on publisher nodes) for incoming messages. When a message appears on any SUB socket (which corresponds to a certain topic), the message is copied to the matching PUB socket (which corresponds to SUB sockets on subscriber nodes), if such socket exists.

In effect, the master acts as a "post exchange". All messages arrive at the master and are forwarded from there. Queues are handled by ZeroMQ.

The topic graphing code is handled by Graphviz. Graphviz takes files in DOT code (a certain specialized graph-drawing language that describes all nodes, edges and connections between nodes). The DOT source code is generated on-the-fly by the master and sent to the client that requested it (yes, it's horribly hacky, but it appears to work). The

client receives the source code and passes it to the `graphviz` library, which accepts raw DOT source code and generates an image from it.

CHAPTER 6

Utilities

ZeROSMQ contains several utilities that help development and data visualization:

- By default, the master runs its control service on port TCP 5555. If you don't want to use it (maybe it's already taken, or maybe it's blocked) you can change it: in the master, find the call to `init()`, at the end, and override the default port (`init(12345)`, for example). This will start the master on your custom port. After that, on all nodes, you need to also override the port in the `zerosmq.init('TAG')` call, by changing it to `zerosmq.init('TAG', 12345)`.
- There is a logging function that uses the tag that you passed to `zerosmq.init()`. To use it, call `zerosmq.log('some message')` from anywhere in your code, after `zerosmq.init()`. This will print the message that you passed, plus a tag at the beginning to help you distinguish it from other nodes' messages.
- There are `zerosmq.unsubscribe('topic')` and `zerosmq.stop_publishing('topic')` functions, which stop a subscription and publications in a given topic, respectively.

Warning: These functions are not tested. Use at your own risk.

- There is a graphing utility which creates an image showing all nodes and their communications. It is described *here*.
- There is also a plotting utility which can show a graph of any topic or set of topics, in real time. It is described *here*.
- ZeROSMQ can record all data published to it, and replay it later. This allows for simpler debugging, and also aids with data processing, since it can be done after the data is collected. This functionality is described *here*.

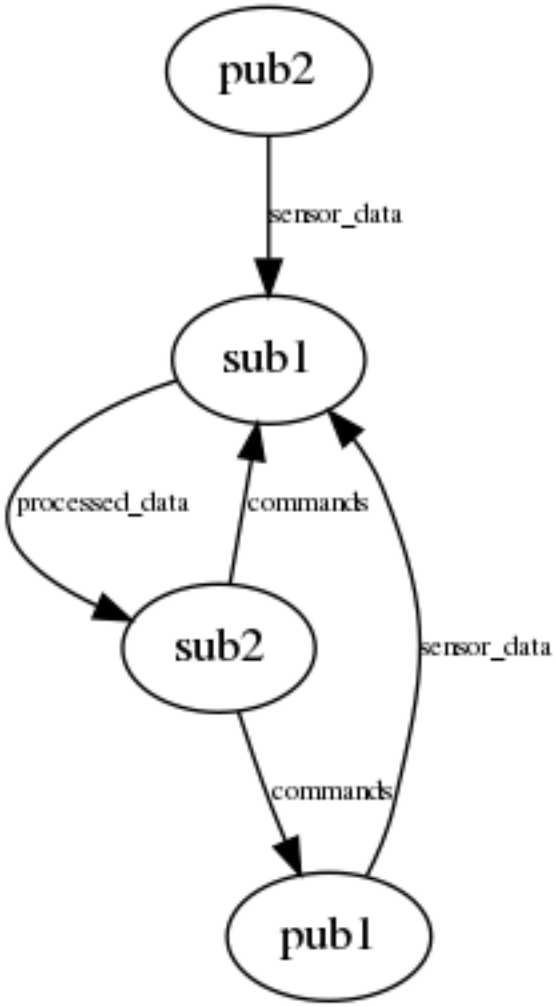
6.1 Graphing nodes

There is an utility that allows you to graph a network of all nodes, plus any topics that connect them. It was inspired by rqt_graph on ROS, minus the nice colors and interactivity. The file *drawmap.py* contains said utility. To use it:

0. Ensure that you have the `graphviz` Python module installed.

1. Run the master node and any other nodes, as usual.
2. When all nodes are up and running, execute `drawmap.py` (`python drawmap.py` or similar).
3. The script should generate and create a PNG image with a map of any nodes and the topics that connect them. If you don't see the image, it should be on the same folder, under the name `temp.gv.png`. The file `temp.gv` is the DOT source which generated the image, should you want it.

See below for an example diagram. `pub1` and `pub2` both generate sensor data and send it to `sub1`. `sub1` processes it and publishes the processed data, which is received by `sub2`. `sub2` monitors the processed data and, if a certain condition is met, will command both `pub1` and `sub1` to stop their activities, but not `pub2`.



6.2 Plotting data

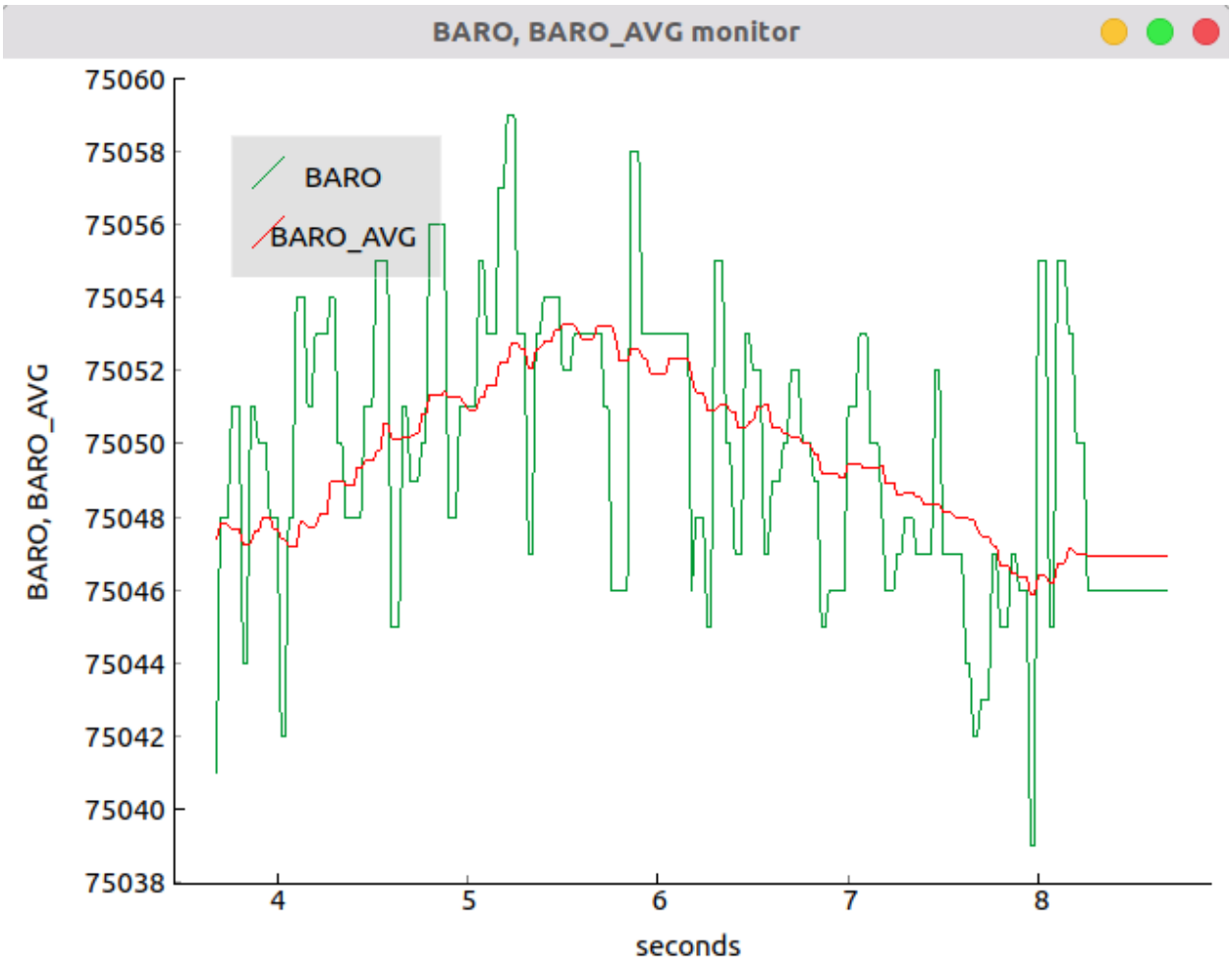
There is an utility which plots any topic or set of topics, in real-time. It is quite similar to (and inspired by) both rqt_plot in ROS and the Arduino IDE serial plotter (by the way, did *you* know it?). It is contained in the file `plotdata.py`. To use it:

0. Ensure that you have the `pyqtgraph` Python module installed.
1. Open the `plotdata.py` file. You should change the top lines. The variable TIME_SPAN sets the length of the plot (it rolls to the left, and TIME_SPAN sets how much time a specific data point will stay on the graph

before disappearing). TOPICS is a list of topics that you want to graph. All topics should *only* contain a single number to be graphed (say, the temperature or the atmospheric pressure).

2. Save the `plotdata.py` file.
3. Run the master node and any other nodes, as usual.
4. When you wish to start plotting data, execute `plotdata.py` (`python plotdata.py` or similar)
5. The script should show a window plotting the data in the specified topics.

See below for an image. Green is real-time barometer data. Red is the same data, processed by a 15-point rolling average.



6.3 Recording and replaying data

ZeROSMQ contains code to record all published data (or a subset of it) to a file, and allow its playback later, which will produce the same effects as when the data was obtained the first time. It is inspired by ROS's bags, and contained in the files `recorddata.py` and `replaydata.py`. Furthermore, the speed at which data is replayed can be configured.

6.3.1 Recording data

To record data:

1. Open the `recorddata.py` file. The variable TOPICS at the top controls which topics will be recorded (for example, if a node subscribes to topic A, processes its data and publishes the processed data on topic B, you may not want to record B, since when you play back the data messages of topic B will appear twice, once from the recorded data and once from the node's processing of topic A's recorded data). If you wish to record all data, leave TOPICS as it is.
2. Save the `recorddata.py` file.
3. Run the master node and any other nodes, as usual.
4. When you wish to start recording data, execute `recorddata.py` (`python recorddata.py` or similar).
5. A file called "RecordData_thecurrentdate.txt" will be created ("thecurrentdate" is the *actual* current date).

6.3.2 Replaying data

To replay recorded data:

1. Open the `replaydata.py` file and edit the variable RECORDED_FILE, at the top, to contain the filename of the recording that you want to play back. If you want to have faster/slower playback, set the SPEED variable. A value of 1.0 will cause real-time playback. Higher values will cause faster playback, and slower values will cause slower playback. A value of 0 would completely stop playback.

Warning: The maximum speed-up achievable depends on your computer (or so it appears). I haven't been able to obtain a speed-up greater than 2 or so, and at that speed the script spends its entire time sending data, with no pauses. Further speed-ups are desired, so feel free to add them in if you can.

Note: You can also set the file name by passing it as a console argument to `replaydata.py` (`python replaydata.py yourfilename.txt`). If you do so, you don't need to edit `replaydata.py`.

2. Save the `replaydata.py` file.
3. Run the master node and any **data processing** nodes (don't run data generators).

Warning: When playing back data, ensure that you don't execute any nodes that publish on any topics that you recorded. If you do so, you will get duplicated data. You can only record sensor data (if you are doing a data acquisition project) and delay all data processing to the playback stage. Otherwise, you can record all topics (sensor data and processed data), but you shouldn't run the data processors on the playback stage.

4. Run the `replaydata.py` file (`python replaydata.py` or similar).
5. The `replaydata` node will scan the file, find all recorded topics, register for publication on all of them and publish all saved messages. Any listening nodes will experience the same effect as if the real data generators were publishing.

CHAPTER 7

Example code

There is some example code in the `examples/` directory. The code implements two publisher nodes and two subscriber nodes. The two publisher nodes (`pub1.py` and `pub2.py`) simulate two redundant sensors which measure the same value. Since no actual sensors are present, the values are randomized. Both nodes publish data on topic SENSOR_DATA. The first subscriber node (`sub1.py`) processes the value (it multiplies it by two) and publishes the "processed data" on topic PROCESSED_DATA. The second subscriber node (`sub2.py`) reads the processed data. If the processed data has a value over 3 (maybe an overrange condition?) it sends back a signal on the topic COMMANDS, which orders nodes to stop data collection. Nodes `pub1.py` and `sub1.py` listen to this order. Node `pub2.py` doesn't listen to it, so it continues generating data after the first publisher has stopped.

When you run `./run.sh` on the `examples/` directory, you will see all nodes subscribing and registering for publication. After that, you will see nodes `pub1.py` and `pub2.py` printing their "sensor data" and sending it to the same topic (SENSOR_DATA). `sub1.py` will print its received data. `sub2.py` will print all received data and, if necessary, send the stop signal. When the stop signal is sent, `pub1.py` (identified as Sensor A) and `sub1.py` will stop operations. `pub2.py` (identified as Sensor B) will continue generating data, but `sub1.py` will no longer process it.

At any moment, you can call `python ../drawmap.py` from the `examples/` folder to create the topic map. Both the source code and the image are included.

Also, you can call `python ../plotdata.py` from the `examples/` folder (after editing the file so that TOPICS contains only the SENSOR_DATA topic) to plot the data sent by the fake sensors.