pyacq Documentation

Release 2.0

Samuel Garcia, Alexandra Corneyllie, Luke Campagnola

Nov 05, 2018

Contents

Contents:

CHAPTER 1

Introduction

1.1 What is Pyacq?

Pyacq is an open-source system for distributed data acquisition and stream processing. Its functionality is organized into nodes that individually handle acquisition, filtering, visualization, and recording. Nodes are created on demand and connected to form a graph of data streams. Nodes may be created and connected within a single thread, or distributed across multiple threads, processes, and hosts.

Fig. 1: An example setup for a neurophysiology experiment.

The figure above shows an example use case: two data acquisition devices–a camera and an electrode array recorded through an ADC–stream data simultaneously to storage. The electrode data is passed through an online spike-sorting analyzer whose output is also stored. Finally, the camera's video feed and the online analysis are streamed to a graphical interface that displays data as it is acquired. The nodes in the graph may run together on a single machine or distributed across a network.

The simplified example below gives a brief overview of the code needed to create part of the graph shown above:

```python
import pyacq

# Connect to a remote host and create a new process there
manager = pyacq.create_manager('rpc')
worker_host = manager.add_host('tcp://10.0.0.103:5678')
worker = worker_host.create_nodegroup()

# Create nodes for data acquisition, analysis, storage, and display
device = manager.create_node('NiDAQmx')
analyzer = manager.create_node('Spikesorter', host=worker)
recorder = manager.create_node('HDF5Recorder', host=worker)
viewer = manager.create_node('QOscilloscope')

# Connect all nodes together
```

(continues on next page)

(continued from previous page)

```
analyzer.input.connect(device.output)
recorder.input.connect(analyzer.output)
viewer.input.connect(analyzer.output)

# Begin acquiring and streaming data
manager.start_all()
```

1.2 License

Pyacq is supported by the French National Center for Scientific Research (CNRS) and the Lyon Neuroscience Research Center (CRNL). It is distributed under the BSD 3-clause license.

1.3 Architecture

Pyacq consists of 1) a collection of nodes with various capabilities for acquisition, processing, and visualization, and 2) a set of core tools that facilitate distributed control and data streaming.

Pyacq is built on several open-source tools including Python 3, numpy for data handling, ZeroMQ for interprocess and network communication, and PyQt for graphical user interface.

1.4 Overview of node types

Acquisition	**Processing**	**Visualization**
PyAudio	*Triggering*	*Oscilloscope*
Webcam (libav, imageio)	*Filtering*	*Wavelet spectrogram*
BrainAmp		*Video display*
Emotiv		
OpenBCI		

1.5 Installation

- Pyacq requires Python 3; support for Python 2 is not planned.
- Several packages are required, but most can be installed with pip:

```
$ pip install pyzmq pytest numpy scipy pyqtgraph vispy colorama msgpack-python␣
↪pyaudio blosc
```

- One final dependency, PyQt4, cannot be installed with pip. Linux distributions typically provide this package. OSX users can get PyQt4 (and most other dependencies) using the Anaconda Python distribution. Windows users can also use Anaconda or download PyQt4 directly from the Riverbank Computing website.
- To install Pyacq, use the standard distutils approach:

```
$ python setup.py install
```

CHAPTER 2

Interacting with nodes

Pyacq delegates its data handling tasks to objects called Nodes. Each type of Node implements a different part of the pipeline such as acquiring data from a device, filtering or online analysis, visualization, or data storage. Nodes are connected by Streams to form a graph in which data flows from device to screen or to disk.

2.1 Creating nodes

In the simplest case, nodes may be created directly by instantiating their classes:

```
audio = pyacq.devices.PyAudio()
viewer = pyacq.viewers.QOscilloscope()
```

For cases that require multiple processes or that are distributed across machines, Pyacq provides mechanisms for creating and managing Nodes remotely:

```
manager = pyacq.create_manager()
nodegroup = manager.default_nodegroup
audio = nodegroup.create_node('PyAudio')
viewer = nodegroup.create_node('QOscilloscope')
```

It is also possible to use both locally- and remotely-instantiated Nodes in the same application. See *Managing distributed nodes* for more information about managing remote Nodes and their processes.

2.2 Configuring and connecting nodes

Nodes are configured and connected in a few steps that must be executed in order:

1. Call `node.configure(...)` to set global parameters for the node such as sample rate, channel selections, etc. Each Node class defines and documents the parameters accepted by its configure method.

2. Configure the node's output streams (if any) by calling `node.outputs['output_name'].configure(...)`. This determines the method of communication that the stream will use–plain TCP data stream, shared memory, etc.–and any associated options such as compression and chunk size.
3. Connect inputs to their sources (if any) by calling `node.inputs['input_name'].connect(other_node.outputs['output_name'])`. The input will be automatically configured to match its associated output.
4. Call `node.initialize()`, which will verify input/output settings, allocate memory, prepare devices, etc.

The following code example, taken from `examples/pyaudio_oscope.py`, demonstrates the creation and configuration of two nodes: the first uses PyAudio to stream an audio signal from a microphone, and the second displays the streamed data.

```
# Create and configure audio input node
dev = PyAudio()
dev.configure(nb_channel=1, sample_rate=44100., input_device_index=default_input,
              format='int16', chunksize=1024)
dev.output.configure(protocol='tcp', interface='127.0.0.1', transfertmode='plaindata')
dev.initialize()

# Create an oscilloscope to display data.
viewer = QOscilloscope()
viewer.configure(with_user_dialog = True)

# Connect audio stream to oscilloscope
viewer.input.connect(dev.output)
viewer.initialize()
```

2.3 Starting and stopping

After a node is created, configured, and initialized, it is ready to begin acquiring or processing data. Calling `node.start()` instructs the node to immediately begin reading data from its inputs and/or sending data through its outputs. Calling `node.stop()` will stop all processing until `start()` is called again:

```
dev.start()
viewer.start()

...

dev.stop()
viewer.stop()
```

To permanently deallocate any memory and device resources used by a node, call `node.close()`. Nodes may be started and stopped multiple times, but may not be reinitialized once they are closed.

2.4 Interacting with remote nodes

It is often useful or necessary to have nodes distributed across multiple threads, processes, or machines (see *Managing distributed nodes*). Pyacq uses a remote procedure call (RPC) system with object proxies to allow remotely-hosted nodes to be treated almost exactly like local nodes:

```
# local:
local_node = MyNodeType()
local_node.configure(...)

# remote:
remote_node = nodegroup.create_node('MyNodeType')
remote_node.configure(...)
remote_node.output.configure(...)

local_node.input.connect(remote_node.output)
local_node.initialize()
remote_node.initialize()

local_node.start()
remote_node.start()
```

In this example, calling any method on `remote_node` causes a message to be sent to the process that owns the node, asking it to invoke the method on our behalf. The calling process blocks until the return value is sent back. Similarly, any attributes accessed from `remote_node` (such as `remote_node.output`) are automatically returned as proxies to the remote process.

One major difference between local and proxied objects is that remote methods may be invoked asynchronously. This done by adding the special keyword argument `_sync='async'` to the method call, which causes the call to immediately return a *Future* object (see concurrent.Future in the Python library reference) that may be used to check the status of the request:

```
future = remote_node.configure(..., _sync='async')

while not future.done():
    # do something while we wait for response

# get the result of calling configure()
result = future.result()
```

More information about the RPC system can be found in the *API reference*.

CHAPTER 3

Managing distributed nodes

In Pyacq it is often useful to have nodes distributed across multiple threads, processes, or machines. Although it is straightforward to manually create and communicate with other processes, it can become cumbersome as the number of distributed resources increases. Pyacq provides high-level tools for managing processes and the nodes hosted within them:

- `Manager` is a central point of control for connecting to remote hosts; starting, stopping, and monitoring distributed processes; and collecting node connectivity information.
- `Host` is a server that runs on remote machines to allow Pyacq to connect and spawn new worker processes.
- `NodeGroup` is a simple object that manages multiple nodes within a single process.

The general procedure for running a set of distributed nodes looks like:

1. Run Host servers on each remote machine (these can be left running indefinitely).
2. Create a Manger from the main process.
3. Ask the Manager to connect to each Host server.
4. Create NodeGroups as needed. Each NodeGroup appears inside a newly spawned process on any of the available hosts.
5. Create Nodes as needed within each NodeGroup.
6. Configure, start, and stop Nodes.
7. Close the Manager. This will shut down all NodeGroups across all hosts.

3.1 Creating a manager

Each application should only start one Manager at most by calling the `create_manager()` function:

```
import pyacq
manager = pyacq.create_manager()
```

By default, the Manager is created in a new process and a `proxy` to the Manager is returned. This allows the Manager to listen and respond in the background to requests made by the Hosts, NodeGroups, and Nodes associated with the application.

Calling `create_manager()` also starts a *log server* to which all error messages will be sent. Any spawned processes that are associated with this application will forward their log messages, uncaught exceptions, and stdout/stderr output back to the log server.

The log server runs inside a new thread of the main process. By default, it prints each received log record along with information about the source host, process, and thread that generated the record. All log records are sorted by their timestamps before being displayed, so it is important that the system clocks are precisely synchronized.

3.2 Connecting to remote hosts

In order to connect to another machine on the network, the remote machine must be running a server that allows the manager to start and stop new processes. This can be done by running the host server script provided with Pyacq:

```
$ python tools/host_server.py tcp://10.0.0.53:8000
```

The IP address and port on which the server should run must be provided as shown above. For each machine that runs a host server, we ask the Manager to make contact with the Host:

```
host = manager.add_host('tcp://10.0.0.53:8000')
```

Making this conection ensures that 1) the Manager is aware that it needs to monitor its resources on the host, 2) the Host will inform the Manager if any of its processes dies unexpectedly and 3) the Host will forward all log records, exceptions, and stdout/stderr output back to the Manager's log server.

3.3 Creating remote Nodes

Although there are few differences between interacting with remote versus local Nodes, a little more effort is required to create a Node on a remote host. We will start by creating a new process on the remote host using `Manager.create_nodegroup()`, then create a new Node using `NodeGroup.create_node()`:

```
# Create a new process with a NodeGroup on the remote host
nodegroup = manager.create_nodegroup(host)

# Next, request the NodeGroup to create a new Node
node = nodegroup.create_node('PyAudio', **kwargs)
```

We now have a `proxy` to a `Node` that has been created in the remote process. We can use this proxy to configure, initialize, start, and stop the Node, *exactly as we would with a locally instantiated Node*:

```
node.configure(...)
node.initialize(...)
node.start()
node.stop()
```

Optionally, we can also request the NodeGroup to remove the Node (if we omit this step, then the Manager will take care of it when it exits):

```
nodegroup.remove_node(node)
```

3.4 Registering new Node classes

Whereas local Nodes are instantiated directly from their classes, remote Nodes are instantiated using their class *names*. Consequently, custom Node classes must be registered through the remote NodeGroup using *register_node_type_from_module()*:

```
nodegroup.register_node_type_from_module('my.module.name', 'MyClassName')
```

This requests the remote NodeGroup to import the named module and to register the named Node subclass found there. Following this call, it is possible to create new instances of your custom Node class within the remote NodeGroup:

```
my_node = nodegroup.create_node('MyClassName', ...)
```

See also:

NodeGroup.register_node_type_from_module()

CHAPTER 4

Data streams

As data is acquired in Pyacq, it is transmitted from Node to Node within the *graph* using *Stream classes*. Each Node has one or more input and/or output streams that may be connected together, and each stream can be configured to transmit different types and shapes of data:

```
device.output.configure(protocol='tcp', interface='127.0.0.1',
                        transfermode='plaindata')
viewer.input.connect(device.output)
recorder.input.connect(device.output)
```

For the most part, Nodes will automatically decide the configuration options for their input/output streams based on the data they receive or generate. Some options, however, must be configured manually. In the sections below we describe the basic operating theory for streams and the associated configuration options.

4.1 Streamable Data Types

Pyacq's streams can, in principle, carry any type of time-varying signal that can be represented by a numpy array. In practice, this is expressed in a few simple cases:

- One or more analog signals such as an audio stream or multichannel EEG data. If multiple signals are transmitted in a single stream, they must be time-locked such that, for each time point represented in the data, every channel must have exactly one value recorded (in other words, it must be possible to represent the data as a 2D array of floating-point values).
- One or more time-locked digital signals. These are typically recorded TTL signals such as a lever-press indicator or the frame exposure signal from a camera.
- A video feed from a camera. Although it would be possible to carry multiple time-locked video signals in a single stream, this might be more naturally implemented by creating a single stream per video feed.
- A stream of events, where each event is a `(time, value)` pair that indicates the time that the event occurred and an integer descriptor of the event. This can be used in a similar way to digital signals (for recording button presses, beam crossings, etc.), but where the events are sparsely coded and the complete sample-by-sample recording of the digital signal is either unnecessary or unavailable.

Streams can be used to transmit multidimensional arrays, and for the most part, the shape of these arrays is left to the user to decide. The only requirement is that the first array axis should represent time. Conceptually, stream data represents an array where axis 0 can have an arbitrary length that grows over time as data is collected. In practice, this data is represented in chunks as numpy arrays with a fixed size for axis 0.

4.2 Data Transmission

Data transmission through a stream occurs in several stages:

1. **Pre-filtering:** As data is passed to an output stream, it is passed through a user-defined sequence of filtering functions. These are used, for example, to scale, cast, or reshape data as needed to meet the stream requirements.
2. **Chunking:** The output stream collects data until a minimum chunk size is reached. The chunk size is determined by the *output stream configuration* and may depend on the data type. For example, a 100 kHz analog signal might be transmitted over a stream in 1000-sample chunks, whereas a video feed might be transmitted one frame at a time.
3. **Transmission:** The chunk is transmitted to all input streams that are connected to the output. The mechanism used to transmit data depends on the `protocol` and `transfermode` arguments used during *output stream configuration*:
   - Plain data stream over TCP: data is sent by TCP using a ZeroMQ socket.
   - Plain data stream within process: data is sent using a ZeroMQ “inproc” socket, which avoids uncecessary memory copies.
   - Shared memory: data is written into shared memory, and the new memory pointer is sent using a TCP or inproc ZeroMQ socket.
4. **Reassembly:** Each connected input stream independently receives data chunks and reassembles the stream.
5. **Post-filtering:** The reconstructed stream data is passed through another user-defined sequence of filtering functions before being made available to the stream user.

When transmitting plain data streams, Pyacq tries to maximize throughput by avoiding any unnecessary data copies. In most cases, a copy is required only if the input array does not occupy a contiguous block of memory.

See also:

OutputStream.configure()

4.2.1 A Simple Example

In this example, we pass an array from one thread to another:

```
import numpy as np
import pyacq
import threading

data = np.array([[1,2], [3,4], [5,6]])

# Create and configure the output stream (sender)
out = pyacq.OutputStream()
out.configure(dtype=data.dtype)

# Create the input stream (receiver) and connect it to
# the output stream
```

(continues on next page)

(continued from previous page)

```
inp = pyacq.InputStream()
inp.connect(out)

# Start a background thread that just receives and prints
# data from the input stream
def receiver():
    global inp
    while True:
        d = inp.recv()
        print("Received: %s" % repr(d))

thread = threading.Thread(target=receiver, daemon=True)
thread.start()

# Send data through the stream
out.send(data)
```

If we run this code from an interactive shell, the last few lines might look like:

```
>>> out.send(data)
>>> Received: (6, array([[1, 2],
       [3, 4],
       [5, 6]]))
```

At this point, we may continue calling `out.send()` indefinitely.

Notes:

- In this example, data is sent over the stream using the default method: each chunk is serialized and transmitted over a tcp socket. This default works well when sending data between processes; for threads, however, we can achieve much better performance with other methods. (see `OutputStream.configure()`)
- InputStream and OutputStream are not thread-safe. Once the input thread is started, we should not attempt to access the InputStream's attributes or methods from the main thread. Likewise, we should not attempt to call ip.connect(out) from within the input thread.
- In this example we have not provided any way to ask the stream thread to exit. Setting `daemon=True` when creating the thread ensures that, once the main thread exits, the stream thread will not prevent the process from exiting as well.

4.2.2 Streaming between processes

In the example above, we used `inp.connect(out)` to establish the connection between the ends of the stream. How does this work when we have the input and output in different processes, or on different machines? We use pyacq's RPC system to allow the streams to negotiate a connection, exactly as if they had been created in the same process:

```
import pyacq

# Start a local RPC server so that a remote InputStream will be able
# to make configuration requests from a local OutputStream:
s = pyacq.RPCServer()
s.run_lazy()

# Create the output stream in the local process
o = pyacq.OutputStream()
```

(continues on next page)

(continued from previous page)

```
o.configure(dtype=float)

# Spawn a new process and create an InputStream there
p = pyacq.ProcessSpawner()
rpyacq = p.client._import('pyacq')
i = rpyacq.InputStream()

# Connect the streams exactly as if they were local
i.connect(o)
```

Although this example is somewhat contrived, it demonstrates the general approach: assuming both processes are running an RPC server, one will be able to initiate a stream connection as long as it has an RPC proxy to the stream from the other process.

4.2.3 Using Streams in Custom Node Types

Node classes are responsible for handling most of the configuration for their input/output streams as well as for sending, receiving, and reconstructing data through the streams. This functionality is mostly hidden from Node users; however, if you plan to write custom Node classes, then it is necessary to understand this process in more detail.

Node subclasses may declare any required properties for their input and output streams through the `_input_specs` and `_output_specs` class attributes. Each attribute is a dict whose keys are the names of the streams and whose values provide the default configuration arguments for the stream. For example:

```
class MyFilterNode(Node):
    _input_specs = {
        'analog_in': dict(streamtype='analogsignal', dtype='int16', shape=(-1, 2),
                          compression='', timeaxis=0, sample_rate=50000.),
        'trigger_in': dict(streamtype='digitalsignal', dtype='uint32', shape=(-1),
                           compression='', timeaxis=0, sample_rate=200000.),
    }

    _output_specs = {
        'filtered_out': dict(streamtype='analogsignal', dtype='float32', shape=(-1,␣
↪2),
                             compression='', timeaxis=0, sample_rate=50000)}
```

This *Node* subclass declares two input streams and one output stream: an analog input called "analog_in", a digital input called "trigger_in", and an analog output called "filtered_out". The configuration parameters specified for each stream are passed to the `spec` argument of `InputStream.__init__()` or `OutputStream.__init__()`.

When the user calls *Node.configure()*, the Node will have its last opportunity to create extra streams (if any) and apply all configuration options to its streams.

Nodes call *OutputStream.send()* to send new data via their output streams, and *InputStream.recv()* to receive data from their input streams. If the stream is a plaindata type, then calling *recv()* will return the next data chunk. In contrast, sharedmem streams only return the poisition within the shared memory array of the next data chunk. In this case, it may be more useful to call `InputStream.get_array_slice()` to return part of the shared memory buffer.

See also:

- The *Noise generator* example demonstrates a simple node with an output stream.
- The *Stream monitor* example demonstrates a simple node with an input stream.

4.2.4 Using streams in GUI nodes

User interface nodes pose a unique challenge because they must somehow work with the Qt event loop. Using a `QTimer` to poll an input node is a valid option, but this requires a tradeoff between latency and CPU usage–a node that responds quickly to stream input would have to poll with a short timer interval, which can be computationally expensive.

A better alternative is to have a background thread block while receiving data on the input stream, and then send a signal to the GUI event loop whenever it receives a packet. This is the purpose of `pyacq.core.ThreadPollInput`.

For example:

```
instream = InputStream()
instream.connect(outstream)

poller = ThreadPollInput(input_stream=instream, return_data=True)

def callback(position, data):
    print("Received new data packet at position %d" % position)

poller.new_data.connect(callback)
```

In this example, we assume a Qt event loop is already running. The `pyacq.core.ThreadPollInput` instance starts a background thread to receive data from `instream`. When data is received, a signal is emitted and the callback is invoked by the Qt event loop. Because this stream is being accessed by another thread, it must **not** be accessed from the main GUI thread until `poller.stop()` and `poller.wait()` have been called.

The default behavior for `pyacq.core.ThreadPollInput` is to emit a signal whenever it receives a data packet. However, this behavior can be customized by overriding the :func:pyacq.core.ThreadPollInput.processData‘ method in a subclass.

4.2.5 Stream management tools

Pyacq provides two simple tools for managing data as it moves between streams:

`pyacq.core.StreamConverter` receives data from an output stream and immediately sends it through another output stream, which could have a different configuration. As an example, one could receive data from a sharedmem stream and then use a `pyacq.core.StreamConverter` to forward the data over a tcp socket:

```
conv = StreamConverter()
conv.configure()

# data arrives via outstream
conv.input.connect(outstream)

# ..and is forwarded via conv.output
conv.output.configure(protocol='tcp')
conv.initialize()

# now we may connect another InputStream to conv.output
```

`pyacq.core.ChannelSplitter` takes a multi-channel stream as input and forwards data from individual channels (or groups of channels) via multiple outputs. This is used primarily when streaming multichannel data to a cluster of nodes that will preform a parallel computation. Although it would be possible to simply send all channel data to all nodes, this could incur a performance penalty depending on the stream protocol. By splitting the stream before sending it to the compute nodes, we can avoid this extra overhead.

CHAPTER 5

Pyacq API Reference

Contents:

5.1 Pyacq Core

5.1.1 Process Management Classes

`pyacq.core.`**`create_manager`**`(`*mode='rpc'*, *auto_close_at_exit=True*`)`

Create a new Manager either in this process or in a new process.

This function also starts a log server to which all log records will be forwarded from the manager and all processes started by the manager. See *LogServer* for more information.

Parameters

mode [str] Must be 'local' to create the Manager in the current process, or 'rpc' to create the Manager in a new process (in which case a proxy to the remote manager will be returned).

auto_close_at_exit [bool] If True, then call *Manager.close()* automatically when the calling process exits (only used when `mode=='rpc'`).

`class` `pyacq.core.`**`Manager`**

Manager is a central point of control for connecting to hosts and creating Nodegroups.

Manager instances should be created using *create_manager()*.

`close``()`

Close the Manager.

If a default host was created by this Manager, then it will be closed as well.

`create_nodegroup``(`*name=None*, *host=None*, *qt=True*, ***kwds*`)`

Create a new NodeGroup process and return a proxy to the NodeGroup.

A NodeGroup is a process that manages one or more Nodes for device interaction, computation, or GUI.

Parameters

name [str] A name for the new NodeGroup's process. This name is used to uniquely identify log messages originating from this nodegroup.

host [None | str | ObjectProxy<Host>] Optional address of the Host that should be used to spawn the new NodeGroup, or a proxy to the Host itself. If omitted, then the NodeGroup is spawned from the Manager's default host.

qt [bool] Whether to start a QApplication in the new process. Default is True.

All extra keyword arguments are passed to 'Host.create_nodegroup()'.

`get_host`(*addr*)
Connect the manager to a Host's RPC server and return a proxy to the host.

Hosts are used as a stable service on remote machines from which new Nodegroups can be spawned or closed.

`get_logger_info`()
Return the address of the log server and the level of the root logger.

`list_hosts`()
Return a list of the Hosts that the Manager is connected to.

class `pyacq.core.`**`NodeGroup`**(*host*, *manager*)
NodeGroup is responsible for managing a collection of Nodes within a single process.

NodeGroups themselves are created and destroyed by Hosts, which manage all NodeGroups on a particular machine.

`add_node`(*node*)
Add a Node to this NodeGroup.

`any_node_running`()
Return True if any of the Nodes in this group are running.

`create_node`(*node_class*, **args*, ***kwds*)
Create a new Node and add it to this NodeGroup.

Return the new Node.

`list_node_types`()
Return a list of the class names for all registered node types.

`register_node_type_from_module`(*modname*, *classname*)
Register a Node subclass with this NodeGroup.

This allows custom Node subclasses to be instantiated in this NodeGroup using `NodeGroup.create_node()`.

`remove_node`(*node*)
Remove a Node from this NodeGroup.

`start_all_nodes`()
Call *Node.start()* for all Nodes in this group.

`stop_all_nodes`()
Call *Node.stop()* for all Nodes in this group.

class `pyacq.core.host.`**`Host`**(*name*, *poll_procs=False*)
Host serves as a pre-existing contact point for spawning new processes on a remote machine.

One Host instance must be running on each machine that will be connected to by a Manager. The Host is only responsible for creating and destroying NodeGroups.

check_spawners()
Check for any processes that have exited and report them to their manager.

This method is called by a timer if the host is created with *poll_procs* True.

close_all_nodegroups(*force=False*)
Close all NodeGroups belonging to this host.

create_nodegroup(*name*, *manager=None*, *qt=True*, ***kwds*)
Create a new NodeGroup in a new process and return a proxy to it.

Parameters

name [str] The name of the new NodeGroup. This will also be used as the name of the process in log records sent to the Manager.

manager [Manager | ObjectProxy<Manager> | None] The Manager to which this NodeGroup belongs.

qt [bool] Whether to start a QApplication in the new process. Default is True.

All extra keyword arguments are passed to 'ProcessSpawner()'.

class pyacq.core.**Node**(*name=''*, *parent=None*)
A Node is the basic element for generating and processing data streams in pyacq.

Nodes may be used to interact with devices, generate data, store data, perform computations, or display user interfaces. Each node may have multiple input and output streams that connect to other nodes. For example:

```
[ data acquisition node ] -> [ processing node ] -> [ display node ]
                                                 -> [ recording node ]
```

An application may directly create and connect the Nodes it needs, or it may use a Manager to create a network of nodes distributed across multiple processes or machines.

The order of operations when creating and operating a node is very important:

1. Instantiate the node directly or remotely using *NodeGroup.create_node*.
2. Call *Node.configure(. . .)* to set global parameters such as sample rate, channel selections, etc.
3. Connect inputs to their sources (if applicable): *Node.inputs['input_name'].connect(other_node.outpouts['output_name'])*
4. Configure outputs: *Node.outputs['output_name'].configure(. . .)*
5. Call *Node.initialize()*, which will verify input/output settings, allocate memory, prepare devices, etc.
6. Call *Node.start()* and *Node.stop()* to begin/end reading from input streams and writing to output streams. These may be called multiple times.
7. Close the node with *Node.close()*. If the node was created remotely, this is handled by the NodeGroup to which it belongs.

Notes

For convenience, if a Node has only 1 input or 1 output:

- *Node.inputs['input_name']* can be written *Node.input*
- *Node.outputs['output_name']* can be written *Node.output*

When there are several outputs or inputs, this shorthand is not permitted.

The state of a Node can be requested using thread-safe methods:

- *Node.running()*
- *Node.configured()*
- *Node.initialized()*

after_input_connect(*inputname*)
This method is called when one of the Node's inputs has been connected.

It may be reimplemented by subclasses.

after_output_configure(*outputname*)
This method is called when one of the Node's outputs has been configured.

It may be reimplemented by subclasses.

check_input_specs()
This method is called during *Node.initialize()* and may be reimplemented by subclasses to ensure that inputs are correctly configured before the node is started.

In case of misconfiguration, this method must raise an exception.

check_output_specs()
This method is called during *Node.initialize()* and may be reimplemented by subclasses to ensure that outputs are correctly configured before the node is started.

In case of misconfiguration, this method must raise an exception.

close()
Close the Node.

This causes all input/output connections to be closed. Nodes must be stopped before they can be closed.

closed()
Return True if the Node has already been closed.

This method is thread-safe.

configure(***kargs*)
Configure the Node.

This method is used to set global parameters such as sample rate, channel selections, etc. Each Node subclass determines the allowed arguments to this method by reimplementing *Node._configure()*.

configured()
Return True if the Node has already been configured.

This method is thread-safe.

initialize()
Initialize the Node.

This method prepares the node for operation by allocating memory, preparing devices, checking input and output specifications, etc. Node subclasses determine the behavior of this method by reimplementing *Node._initialize()*.

initialized()
Return True if the Node has already been initialized.

This method is thread-safe.

input
Return the single input for this Node.

If the node does not have exactly one input, then raise AssertionError.

output
Return the single output for this Node.

If the node does not have exactly one put, then raise AssertionError.

running()
Return True if the Node is running.

This method is thread-safe.

start()
Start the Node.

When the node is running it will read from its input streams and write to its output streams (if any). Nodes must be configured and initialized before they are started, and can be stopped and restarted any number of times.

stop()
Stop the Node (see *start()*).

5.1.2 Stream Classes

class pyacq.core.**InputStream**(*spec=None*, *node=None*, *name=None*)
Class for streaming data from an OutputStream.

Streams allow data to be sent between objects that may exist on different threads, processes, or machines. They offer a variety of transfer methods including TCP for remote connections and IPC for local connections.

Typical InputStream usage:

1. Use `InputStream.connect()` to connect to an `OutputStream` defined elsewhere. Usually, the argument will actually be a proxy to a remote `OutputStream`.
2. Poll for incoming data packets with `InputStream.poll()`.
3. Receive the next packet with `InputStream.recv()`.

Optionally, use `InputStream.set_buffer()` to attach a `RingBuffer` for easier data handling.

close()
Close the stream.

This closes the socket. No data can be received after this point.

connect(*output*)
Connect an output to this input.

Any data send over the stream using `output.send()` can be retrieved using `input.recv()`.

Parameters

output [OutputStream (or proxy to a remote OutputStream)] The OutputStream to connect.

empty_queue()
Receive all pending messing in the zmq queue without consuming them. This is usefull when a Node do not start at the same time than other nodes but was already connected. In that case the zmq water mecanism put messages in a queue and when you start cusuming you get old message. This can be annoying. This recv every thing with timeout=0 and so empty the queue.

get_data(**args*, ***kargs*)
Return a segment of the RingBuffer attached to this InputStream.

If no RingBuffer is attached, raise an exception.

For parameters, see `RingBuffer.get_data()`.

See also: `InputStream.set_buffer()`.

`poll`(*timeout=None*)
Poll the socket of input stream.

Return True if a new packet is available.

`recv`(***kargs*)
Receive a chunk of data.

Returns

index: int The absolute sample index. This is the index of the last sample + 1.

data: np.ndarray or bytes The received chunk of data. If the stream uses `transfermode='sharedarray'`, then the data is returned as None and you must use `input_stream[start:stop]` to read from the shared array or `input_stream.recv(with_data=True)` to return the received data chunk.

`reset_buffer_index`()
Reset the buffer index. Usefull for multiple start/stop on Node to reset the index.

`set_buffer`(*size=None*, *double=True*, *axisorder=None*, *shmem=None*, *fill=None*)
Ensure that this InputStream has a RingBuffer at least as large as *size* and with the specified double-mode and axis order.

If necessary, this will attach a new RingBuffer to the stream and remove any existing buffer.

class `pyacq.core.`**`OutputStream`**(*spec=None*, *node=None*, *name=None*)
Class for streaming data to an InputStream.

Streams allow data to be sent between objects that may exist on different threads, processes, or machines. They offer a variety of transfer methods including TCP for remote connections and IPC for local connections.

Parameters

spec [dict] Required parameters for this stream. These may not be overridden when calling `configure()` later on.

node [Node or None]

name [str or None]

`close`()
Close the output.

This closes the socket and releases shared memory, if necessary.

`configure`(***kargs*)
Configure the output stream.

Parameters

protocol ['tcp', 'udp', 'inproc' or 'inpc' (linux only)] The type of protocol used for the zmq.PUB socket

interface [str] The bind adress for the zmq.PUB socket

port [str] The port for the zmq.PUB socket

transfermode: str The method used for data transfer:

- 'plaindata': data are sent over a plain socket in two parts: (frame index, data).

- ‘sharedmem’: data are stored in shared memory in a ring buffer and the current frame index is sent over the socket.
- ‘shared_cuda_buffer’: (planned) data are stored in shared Cuda buffer and the current frame index is sent over the socket.
- ‘share_opencl_buffer’: (planned) data are stored in shared OpenCL buffer and the current frame index is sent over the socket.

All registered transfer modes can be found in *pyacq.core.stream.all_transfermodes*.

streamtype: ‘analogsignal’, ‘digitalsignal’, ‘event’ or ‘image/video’ The nature of data to be transferred.

dtype: str (‘float32’,’float64’, [(‘r’, ‘uint16’), (‘g’, ‘uint16’), , (‘b’, ‘uint16’)], . . .) The numpy.dtype of the data buffer. It can be a composed dtype for event or images.

shape: list The shape of each data frame. If the stream will send chunks of variable length, then use -1 for the first (time) dimension.

- For `streamtype=image`, the shape should be `(-1, H, W)` or `(n_frames, H, W)`.
- For `streamtype=analogsignal` the shape should be `(n_samples, n_channels)` or `(-1, n_channels)`.

compression: ‘’, ‘blosclz’, ‘blosc-lz4’ The compression for the data stream. The default uses no compression.

scale: float An optional scale factor + offset to apply to the data before it is sent over the stream. `output = offset + scale * input`

offset: See *scale*.

units: str Units of the stream data. Mainly used for ‘analogsignal’.

sample_rate: float or None Sample rate of the stream in Hz.

kwargs : All extra keyword arguments are passed to the DataSender constructor for the chosen transfermode (for example, see *SharedMemSender*).

reset_buffer_index()

Reset the buffer index. Usefull for multiple start/stop on Node to reset the index.

send(*data*, *index=None*, ***kargs*)

Send a data chunk and its frame index.

Parameters

index: int The absolute sample index. This is the index of the last sample + 1.

data: np.ndarray or bytes The chunk of data to send.

class pyacq.core.stream.plaindatastream.**PlainDataSender**(*socket*, *params*)

Helper class to send data serialized over socket.

Note: this class is usually not instantiated directly; use `OutputStream.configure(transfermode='plaindata')`.

To avoid unnecessary copies (and thus optimize transmission speed), data is sent exactly as it appears in memory including array strides.

This class supports compression.

class `pyacq.core.stream.sharedmemstream.`**`SharedMemSender`**(*socket*, *params*)

Stream sender that uses shared memory for efficient interprocess communication. Only the data pointer is sent over the socket.

Note: this class is usually not instantiated directly; use `OutputStream.configure(transfermode='sharedmem')`.

Extra parameters accepted when configuring the output stream:

- buffer_size (int) the size of the shared memory buffer in *frames*. The total shape of the allocated buffer is `(buffer_size,) + shape`.
- double (bool) if True, then the buffer size is doubled and all frames are written to the buffer twice. This makes it possible to guarantee zero-copy reads by any connected InputStream.
- axisorder (tuple) The order that buffer axes should be arranged in memory. This makes it possible to optimize for specific algorithms that expect either row-major or column-major alignment. The default is row-major; the time axis comes first in the axis order.
- fill (float) Value used to fill the buffer where no data is available.

5.1.3 Data Handling Classes

class `pyacq.core.`**`RingBuffer`**(*shape*, *dtype*, *double=True*, *shmem=None*, *fill=None*, *axisorder=None*)

Class that collects data as it arrives from an InputStream and writes it into a single- or double-ring buffer.

This allows the user to request the concatenated history of data received by the stream, up to a predefined length. Double ring buffers allow faster, copyless reads at the expense of doubled write time and memory footprint.

`get_data`(*start*, *stop*, *copy=False*, *join=True*)

Return a segment of the ring buffer.

Parameters

start [int] The starting index of the segment to return.

stop [int] The stop index of the segment to return (the sample at this index will not be included in the returned data)

copy [bool] If True, then a copy of the data is returned to ensure that modifying the data will not affect the ring buffer. If False, then a reference to the buffer will be returned if possible. Default is False.

join [bool] If True, then a single contiguous array is returned for the entire requested segment. If False, then two separate arrays are returned for the beginning and end of the requested segment. This can be used to avoid an unnecessary copy when the buffer has double=False and the caller does not require a contiguous array.

class `pyacq.core.stream.sharedarray.`**`SharedMem`**(*nbytes*, *shm_id=None*)

Class to create a shared memory buffer.

This class uses mmap so that unrelated processes (not forked) can share it.

It is usually not necessary to instantiate this class directly; use `OutputStream.configure(transfermode='sharedmem')`.

Parameters

size [int] Buffer size in bytes.

shm_id [str or None] The id of an existing SharedMem to open. If None, then a new shared memory file is created. On linux this is the filename, on Windows this is the tagname.

close()
Close this buffer.

to_dict()
Return a dict that can be serialized and sent to other processes to access this buffer.

to_numpy(*offset*, *dtype*, *shape*, *strides=None*)
Return a numpy array pointing to part (or all) of this buffer.

5.2 Remote Process Control API

Contents:

5.2.1 Overview: Remote Process Control

Pyacq implements a system for spawning and controlling remote processes through object proxies. This allows remote objects to be treated almost exactly as if they were local objects, with the exception that methods of object proxies may be called asynchronously.

The remote process control system consists of several components:

- *RPCServer* uses ZeroMQ to listen for serialized requests to control the process by invoking methods, returning objects, etc. RPCServer instances are automatically created in subprocesses when using ProcessSpawner.
- *RPCClient* sends messages and receives responses from an RPCServer in another thread, process, or host. Each RPCClient instance connects to only one RPCServer. RPCClient instances are created automatically when using ProcessSpawner, or can be created manually using RPCClient.get_client.
- *ObjectProxy* is the class used to represent any type of remote object. Invoking methods on an ObjectProxy causes a request to be sent to the remote process, and the result is eventually returned via the ObjectProxy.
- *ProcessSpawner* is used to spawn new processes on the same machine as the caller. New processes will automatically start an RPCServer, and an RPCClient will be created in the parent process.
- *Serializers* (currently msgpack and json are supported) are used to encode basic python types for transfer over ZeroMQ sockets. Clients are free to pick whichever serializer they prefer. List of data types:
- *Logging tools* that allow log records, uncaught excaptions, and stdout/stderr data to be sent to a remote log server. These are essential for debugging multiprocess applications.

The following simple example makes use of most of these components, although only ProcessSpawner and Object-Proxy are directly visible to the user:

```
from pyacq.core.rpc import ProcessSpawner

# Start a new process with an RPCServer running inside
proc = ProcessSpawner()

# Ask the remote process to import a module and return a proxy to it
remote_col = proc.client._import('collections')

# Create a new object (an ordered dict) in the remote process
remote_dict = remote_col.OrderedDict()

# Interact with the new object exactly as if it were local:
remote_dict['x'] = 1
```

(continues on next page)

(continued from previous page)

```
assert 'x' in remote_dict.keys()
assert remote_dict['x'] == 1
```

Using object proxies allows remote objects to be accessed using the same syntax as if they were local. However, there are two major differences to consider when using remote objects:

First, function arguments and return values in Python are passed by reference. This means that both the caller and the callee operate on the *same* Python object. Since it is not possible to share python objects between processes, we are restricted to sending them either by copy or by proxy. By default, arguments and return values for remote functions are serialized if possible, or passed by proxy otherwise.

Second, remote functions can be called asynchronously. By default, calling a remote function will block until the return value has arrived. However, any remote function call can be made asynchronous by adding a special argument: `_sync='async'`. In this case, the function call will immediately return a *`Future`* object that can be used to access the return value when it arrives.

5.2.2 ObjectProxy and Future

After initial setup, these classes are the main API through which a remote process is controlled.

class `pyacq.core.rpc.`**`ObjectProxy`**(*rpc_addr*, *obj_id*, *ref_id*, *type_str=”*, *attributes=()*, ***kwds*)

Proxy to an object stored by a remote *`RPCServer`*.

A proxy behaves in most ways like the object that it wraps–you can request the same attributes, call methods, etc. There are a few important differences:

- A proxy can be on a different thread, process, or machine than its object, so long as the object’s thread has an RPCServer and the proxy’s thread has an associated RPCClient.
- Attribute lookups and method calls can be slower because the request and response must traverse a socket. These can also be performed asynchronously to avoid blocking the client thread.
- Function argument and return types must be serializable or proxyable. Most basic types can be serialized, including numpy arrays. All other objects are automatically proxied, but there are some cases when this will not work well.
- `__repr__()` and `__str__()` are overridden on proxies to allow safe debugging.
- `__hash__()` is overridden to ensure that remote hash values are not used locally.

For the most part, object proxies can be used exactly as if they are local objects:

```
client = RPCClient(address)  # connect to RPCServer
rsys = client._import('sys') # returns proxy to sys module on remote process
rsys.stdout.write            # proxy to remote sys.stdout.write
rsys.stdout.write('hello')   # calls sys.stdout.write('hello') on remote machine
                             # and returns the result (None)
```

When calling a proxy to a remote function, the call can be made synchronous (caller is blocked until result can be returned), asynchronous (Future is returned immediately and result can be accessed later), or return can be disabled entirely:

```
ros = proc._import('os')

# synchronous call; caller blocks until result is returned
pid = ros.getpid()

```

(continues on next page)

(continued from previous page)

```
# asynchronous call
request = ros.getpid(_sync='async')
while not request.hasResult():
    time.sleep(0.01)
pid = request.result()

# disable return when we know it isn't needed.
rsys.stdout.write('hello', _sync='off')
```

Additionally, values returned from a remote function call are automatically returned either by value (must be picklable) or by proxy. This behavior can be forced:

```
rnp = proc._import('numpy')
arrProxy = rnp.array([1,2,3,4], _return_type='proxy')
arrValue = rnp.array([1,2,3,4], _return_type='value')
```

The default sync and return_type behaviors (as well as others) can be set for each proxy individually using ObjectProxy._set_proxy_options() or globally using proc.set_proxy_options().

It is also possible to send arguments by proxy if an RPCServer is running in the caller's thread (this can be used, for example, to connect Qt signals across network connections):

```
def callback():
    print("called back.")

# Remote process can invoke our callback function as long as there is
# a server running here to process the request.
remote_object.set_callback(proxy(callback))
```

`__call__`(**args*, ***kwargs*)

Call the proxied object from the remote process.

All positional and keyword arguments (except those listed below) are sent to the remote procedure call.

In synchronous mode (see parameters below), this method blocks until the remote return value has been received, and then returns that value. In asynchronous mode, this method returns a `Future` instance immediately, which may be used to retrieve the return value later. If return is disabled, then the method immediately returns None.

If the remote call raises an exception on the remote process, then this method will raise RemoteCallException if in synchronous mode, or calling `Future.result()` will raise RemoteCallException if in asynchronous mode. If return is disabled, then remote exceptions will be ignored.

Parameters

_sync: 'off', 'sync', or 'async' Set the sync mode for this call. The default value is determined by the 'sync' argument to `_set_proxy_options()`.

_return_type: 'value', 'proxy', or 'auto' Set the return type for this call. The default value is determined by the 'return_type' argument to `_set_proxy_options()`.

_timeout: float Set the timeout for this call. The default value is determined by the 'timeout' argument to `_set_proxy_options()`.

See also:

`RPCClient.call_obj`, `Future`

`__getattr__`(*attr*)
Calls __getattr__ on the remote object and returns the attribute by value or by proxy depending on the options set (see ObjectProxy._set_proxy_options and RemoteEventHandler.set_proxy_options)

If the option 'defer_getattr' is True for this proxy, then a new proxy object is returned _without_ asking the remote object whether the named attribute exists. This can save time when making multiple chained attribute requests, but may also defer a possible AttributeError until later, making them more difficult to debug.

`__setattr__`(**args*)
Implement setattr(self, name, value).

`_delete`(*sync='sync'*, ***kwds*)
Ask the RPC server to release the reference held by this proxy.

Note: this does not guarantee the remote object will be deleted; only that its reference count will be reduced. Any copies of this proxy will no longer be usable.

`_get_value`()
Return the value of the proxied object.

If the object is not serializable, then raise an exception.

`_set_proxy_options`(***kwds*)
Change the behavior of this proxy. For all options, a value of None will cause the proxy to instead use the default behavior defined by its parent Process.

Parameters

sync ['sync', 'async', 'off', or None] If 'async', then calling methods will return a `Future` object that can be used to inquire later about the result of the method call. If 'sync', then calling a method will block until the remote process has returned its result or the timeout has elapsed (in this case, a Request object is returned instead). If 'off', then the remote process is instructed *not* to reply and the method call will return None immediately. This option can be overridden by supplying a `_sync` keyword argument when calling the method (see `__call__()`).

return_type ['auto', 'proxy', 'value', or None] If 'proxy', then the value returned when calling a method will be a proxy to the object on the remote process. If 'value', then attempt to pickle the returned object and send it back. If 'auto', then the decision is made by consulting the 'no_proxy_types' option. This option can be overridden by supplying a `_return_type` keyword argument when calling the method (see `__call__()`).

auto_proxy [bool or None] If True, arguments to __call__ are automatically converted to proxy unless their type is listed in no_proxy_types (see below). If False, arguments are left untouched. Use proxy(obj) to manually convert arguments before sending.

timeout [float or None] Length of time to wait during synchronous requests before returning a Request object instead. This option can be overridden by supplying a `_timeout` keyword argument when calling a method (see `__call__()`).

defer_getattr [True, False, or None] If False, all attribute requests will be sent to the remote process immediately and will block until a response is received (or timeout has elapsed). If True, requesting an attribute from the proxy returns a new proxy immediately. The remote process is *not* contacted to make this request. This is faster, but it is possible to request an attribute that does not exist on the proxied object. In this case, AttributeError will not be raised until an attempt is made to look up the attribute on the remote process.

no_proxy_types [list] List of object types that should *not* be proxied when sent to the remote process.

auto_delete [bool] If True, then the proxy will automatically call *self._delete()* when it is collected by Python.

class `pyacq.core.rpc.`**`Future`**(*client*, *call_id*)

Represents a return value from a remote procedure call that has not yet arrived.

Instances of Future are returned from *`ObjectProxy.__call__()`* when used with `_sync='async'`. This is the mechanism through which remote functions may be called asynchronously.

Use `done()` to determine whether the return value (or an error message) has arrived, and *`result()`* to get the return value. If the remote call raised an exception, then calling *`result()`* will raise RemoteCallException with a transcript of the original exception.

See *concurrent.futures.Future* in the Python documentation for more information.

`cancel`()

Cancel the future if possible.

Returns True if the future was cancelled, False otherwise. A future cannot be cancelled if it is running or has already completed.

`result`(*timeout=None*)

Return the result of this Future.

If the result is not yet available, then this call will block until the result has arrived or the timeout elapses.

5.2.3 ProcessSpawner

class `pyacq.core.rpc.`**`ProcessSpawner`**(*name=None*, *address='tcp://127.0.0.1:*'*, *qt=False*, *log_addr=None*, *log_level=None*, *executable=None*)

Utility for spawning and bootstrapping a new process with an *`RPCServer`*.

Automatically creates an *`RPCClient`* that is connected to the remote process (`spawner.client`).

Parameters

name [str | None] Optional process name that will be assigned to all remote log records.

address [str] ZMQ socket address that the new process's RPCServer will bind to. Default is `'tcp://127.0.0.1:*'`.

Note: binding RPCServer to a public IP address is a potential security hazard (see *`RPCServer`*).

qt [bool] If True, then start a Qt application in the remote process, and use a *`QtRPCServer`*.

log_addr [str] Optional log server address to which the new process will send its log records. This will also cause the new process's stdout and stderr to be captured and forwarded as log records.

log_level [int] Optional initial log level to assign to the root logger in the new process.

executable [str | None] Optional python executable to invoke. The default value is *sys.executable*.

Examples

```
# start a new process
proc = ProcessSpawner()

# ask the child process to do some work
mod = proc._import('my.module')
mod.do_work()

# close the child process
proc.close()
proc.wait()
```

client = None
An RPCClient instance that is connected to the RPCServer in the remote process

kill()
Kill the spawned process immediately.

poll()
Return the spawned process's return code, or None if it has not exited yet.

stop()
Stop the spawned process by asking its RPC server to close.

wait(*timeout=10*)
Wait for the process to exit and return its return code.

5.2.4 RPCClient and RPCServer

These classes implement the low-level communication between a server and client. They are rarely used directly by the user except occasionally for initial setup and closing.

class `pyacq.core.rpc.`**RPCClient**(*address*, *reentrant=True*, *serializer='msgpack'*)
Connection to an `RPCServer`.

Each RPCClient connects to only one server, and may be used from only one thread. RPCClient instances are created automatically either through `ProcessSpawner` or by requesting attributes form an `ObjectProxy`. In general, it is not necessary for the user to interact directly with RPCClient.

Parameters

address [URL] Address of RPC server to connect to.

reentrant [bool] If True, then this client will allow the server running in the same thread (if any) to process requests whenever the client is waiting for a response. This is necessary to avoid deadlocks in case of reentrant RPC requests (eg, server A calls server B, which then calls server A again). Default is True.

call_obj(*obj*, *args=None*, *kwargs=None*, ***kwds*)
Invoke a remote callable object.

Parameters

obj [`ObjectProxy`] A proxy that references an object owned by the connected RPC-Server.

args [tuple] Arguments to pass to the remote call.

kwargs [dict] Keyword arguments to pass to the remote call.

kwds : All extra keyword arguments are passed to `send()`.

close()
Close this client's socket (but leave the server running).

close_server(*sync='sync'*, *timeout=1.0*, ***kwds*)
Ask the server to close.

The server returns True if it has closed. All clients known to the server will be informed that the server has disconnected.

If the server has already disconnected from this client, then the method returns True without error.

delete(*obj*, ***kwds*)
Delete an object proxy.

This informs the remote process that an `ObjectProxy` is no longer needed. The remote process will decrement a reference counter and delete the referenced object if it is no longer held by any proxies.

Parameters

obj [`ObjectProxy`] A proxy that references an object owned by the connected RPC-Server.

kwds : All extra keyword arguments are passed to `send()`.

Notes

After a proxy is deleted, it cannot be used to access the remote object even if the server has not released the remote object yet. This also applies to proxies that are sent to a third process. For example, consider three processes A, B, C: first A acquires a proxy to an object owned by B. A sends the proxy to C, and then deletes the proxy. If C attempts to access this proxy, an exception will be raised because B has already remoted the reference held by this proxy. However, if C independently acquires a proxy to the same object owned by B, then that proxy will continue to function even after A deletes its proxy.

disconnected()
Boolean indicating whether the server has disconnected from the client.

ensure_connection(*timeout=1.0*)
Make sure RPC server is connected and available.

static get_client(*address*)
Return the RPC client for this thread and a given server address.

If no client exists already, then a new one will be created. If the server is running in the current thread, then return None.

See also:

`RPCServer.address`

get_obj(*obj*, ***kwds*)
Return a copy of a remote object.

Parameters

obj [`ObjectProxy`] A proxy that references an object owned by the connected RPC-Server. The object will be serialized and returned if possible, otherwise a new proxy is returned.

kwds : All extra keyword arguments are passed to `send()`.

measure_clock_diff()
Measure the clock offset between this host and the remote host.

`ping`(*sync='sync'*, ***kwds*)
Ping the server.

This can be used to test connectivity to the server.

`process_msg`(*msg*)
Handle one message received from the remote process.

This takes care of assigning return values or exceptions to existing Future instances.

`process_until_future`(*future*, *timeout=None*)
Process all incoming messages until receiving a result for *future*.

If the future result is not raised before the timeout, then raise TimeoutError.

While waiting, the RPCServer for this thread (if any) is also allowed to process requests.

Parameters

future [concurrent.Future instance] The Future to wait for. When the response for this Future arrives from the server, the method returns.

timeout [float] Maximum time (seconds) to wait for a response.

`send`(*action*, *opts=None*, *return_type='auto'*, *sync='sync'*, *timeout=10.0*)
Send a request to the remote process.

It is not necessary to call this method directly; instead use `call_obj()`, `get_obj()`, `__getitem__()`, `__setitem__()`, `transfer()`, `delete()`, `import()`, or `ping()`.

The request is given a unique ID that is included in the response from the server (if any).

Parameters

action [str] The action to invoke on the remote process. See list of actions below.

opts [None or dict] Extra options to be sent with the request. Each action requires a different set of options. See list of actions below.

return_type ['auto' | 'proxy'] If 'proxy', then the return value is sent by proxy. If 'auto', then the server decides based on the return type whether to send a proxy.

sync [str] If 'sync', then block and return the result when it becomes available. If 'async', then return a Future instance immediately. If 'off', then ask the remote server NOT to send a response and return None immediately.

timeout [float] The amount of time to wait for a response when in synchronous operation (sync='sync'). If the timeout elapses before a response is received, then raise TimeoutError.

Notes

The following table lists the actions that are recognized by RPCServer. The *action* argument to *send()* may be any string from the *Action* column below, and the *opts* argument must be a dict with the keys listed in the *Options* column.

Action	Description	Options
call_obj	Invoke a callable	obj: a proxy to the callable object args: a tuple of positional arguments kwargs: a dict of keyword arguments
get_obj	Return the object referenced by a proxy	obj: a proxy to the object to return
get_item	Return a named object	name: string name of the object to return
set_item	Set a named object	name: string name to set value: object to assign to name
delete	Delete a proxy reference	obj_id: proxy object ID ref_id: proxy reference ID
import	Import and return a proxy to a module	module: name of module to import
ping	Return 'pong'	

`transfer`(*obj*, ***kwds*)

Send an object to the remote process and return a proxy to it.

Parameters

obj [object] Any object to send to the remote process. If the object is not serializable then a proxy will be sent if possible.

kwds : All extra keyword arguments are passed to `send()`.

class `pyacq.core.rpc.`**`RPCServer`**(*address='tcp://127.0.0.1:*'*)

Remote procedure call server for invoking requests on proxied objects.

RPCServer instances are automatically created when using `ProcessSpawner`. It is rarely necessary for the user to interact directly with RPCServer.

There may be at most one RPCServer per thread. RPCServers can be run in a few different modes:

- **Exclusive event loop**: call *run_forever()* to cause the server to listen indefinitely for incoming request messages.

- **Lazy event loop**: call *run_lazy()* to register the server with the current thread. The server's socket will be polled whenever an RPCClient is waiting for a response (this allows reentrant function calls). You can also manually listen for requests with *_read_and_process_one()* in this mode.
- **Qt event loop**: use `QtRPCServer`. In this mode, messages are polled in a separate thread, but then sent to the Qt event loop by signal and processed there. The server is registered as running in the Qt thread.

Parameters

name [str] Name used to identify this server.

address [URL] Address for RPC server to bind to. Default is `'tcp://127.0.0.1:*'`.

Note: binding RPCServer to a public IP address is a potential security hazard.

Notes

RPCServer is not a secure server. It is intended to be used only on trusted networks; anyone with tcp access to the server can execute arbitrary code on the server.

RPCServer is not a thread-safe class. Only use `RPCClient` to communicate with RPCServer from other threads.

Examples

```
# In host/process/thread 1:
server = RPCServer()
rpc_addr = server.address

# Publish an object for others to access easily
server['object_name'] = MyClass()

# In host/process/thread 2: (you must communicate rpc_addr manually)
client = RPCClient(rpc_addr)

# Get a proxy to published object; use this (almost) exactly as you
# would a local object:
remote_obj = client['object_name']
remote_obj.method(...)

# Or, you can remotely import and operate a module:
remote_module = client._import("my.module.name")
remote_obj = remote_module.MyClass()
remote_obj.method(...)

# See ObjectProxy for more information on interacting with remote
# objects, including (a)synchronous communication.
```

`address = None`
The zmq address where this server is listening (e.g. 'tcp:///127.0.0.1:5678')

`close``()`
Ask the server to close.

This method is thread-safe.

get_proxy(*obj*, ***kwds*)
Return an ObjectProxy referring to a local object.

This proxy can be sent via RPC to any other node.

static **get_server**()
Return the server running in this thread, or None if there is no server.

static **local_client**()
Return the RPCClient used for accessing the server running in the current thread.

process_action(*action*, *opts*, *return_type*, *caller*)
Invoke a single action and return the result.

static **register_server**(*srv*)
Register a server as the (only) server running in this thread.

This static method fails if another server is already registered for this thread.

run_forever()
Read and process RPC requests until the server is asked to close.

run_lazy()
Register this server as being active for the current thread, but do not actually begin processing requests.

RPCClients in the same thread will allow the server to process requests while they are waiting for responses. This can prevent deadlocks that occur when

This can also be used to allow the user to manually process requests.

running()
Boolean indicating whether the server is still running.

start_timer(*callback*, *interval*, ***kwds*)
Start a timer that invokes *callback* at regular intervals.

Parameters

callback [callable] Callable object to invoke. This must either be an ObjectProxy or an object that is safe to call from the server's thread.

interval [float] Minimum time to wait between callback invocations (start to start).

static **unregister_server**(*srv*)
Unregister a server from this thread.

unwrap_proxy(*proxy*)
Return the local python object referenced by *proxy*.

class pyacq.core.rpc.**QtRPCServer**(*address='tcp://127.0.0.1:*'*, *quit_on_close=True*)
RPCServer that lives in a Qt GUI thread.

This server may be used to create and manage QObjects, QWidgets, etc. It uses a separate thread to poll for RPC requests, which are then sent to the Qt event loop using by signal. This allows the RPC actions to be executed in a Qt GUI thread without using a timer to poll the RPC socket. Responses are sent back to the poller thread by a secondary socket.

QtRPCServer may be started in newly spawned processes using `ProcessSpawner`.

Parameters

address [str] ZMQ address to listen on. Default is `'tcp://127.0.0.1:*'`.

Note: binding RPCServer to a public IP address is a potential security hazard. See `RPCServer`.

quit_on_close [bool] If True, then call *QApplication.quit()* when the server is closed.

Examples

Spawning in a new process:

```
# Create new process.
proc = ProcessSpawner(qt=True)

# Display a widget from the new process.
qtgui = proc._import('PyQt4.QtGui')
w = qtgui.QWidget()
w.show()
```

Starting in an existing Qt application:

```
# Create server.
server = QtRPCServer()

# Start listening for requests in a background thread (this call
# returns immediately).
server.run_forever()
```

process_action(*action*, *opts*, *return_type*, *caller*)
Invoke a single action and return the result.

run_forever()
Read and process RPC requests until the server is asked to close.

5.2.5 Serializers

Serializers provide a mechanism for some data types to be copied from one process to another by converting Python objects into byte strings and vice-versa. Currently, two serializer classes are supported:

- **msgpack** provides efficient serialization for all supported types, including large binary data.
- **json** is somewhat less efficient in encoding large binary data, but is more universally supported across platforms where msgpack may be unavailable.

The basic types supported by both serializers are `int`, `float`, `str`, `dict`, and `list`. Further data types are serialized by first converting to a dict containing the key `___type_name___` in order to distinguish it from normal dicts (see *Serializer.encode()* and *Serializer.decode()*):

```
datetime = {
    '___type_name___': 'datetime',
    'data': obj.strftime('%Y-%m-%dT%H:%M:%S.%f')
}

date = {
    '___type_name___': 'date',
    'data': obj.strftime('%Y-%m-%d')
}

nonetype = {
    '___type_name___': 'none'
}
```

(continues on next page)

(continued from previous page)

```
objectproxy = {
    '___type_name___': 'proxy',
    'rpc_addr': obj._rpc_addr,
    'obj_id': obj._obj_id,
    'ref_id': obj._ref_id,
    'type_str': obj._type_str,
    'attributes': obj._attributes,
}
```

Types containing byte strings are handled differently between msgpack and json. In msgpack, byte strings are natively supported:

```
np.ndarray = {
    '___type_name___': 'ndarray',
    'data': array.tostring(),
    'dtype': str(array.dtype),
    'shape': array.shape
}

# no need to convert; msgpack already handles this type
bytes = bytes_obj
```

However json does not support byte strings, so in this case the strings must be base-64 encoded before being serialized:

```
ndarray = {
    '___type_name___': 'ndarray',
    'data': base64.b64encode(array.data).decode(),
    'dtype': str(array.dtype),
    'shape': array.shape
}

bytes = {
    '__type_name__': 'bytes',
    'data': base64.b64encode(bytes_obj).decode()
}
```

Note that both serializers convert tuples into lists automatically. This is undesirable, but is currently not configurable in a consistent way across both serializers.

It is possible to add support for new serializers by creating a subclass of `Serializer` and modifying `pyacq.core.rpc.serializer.all_serializers`.

class `pyacq.core.rpc.serializer.`**`Serializer`**(*server=None*, *client=None*)

Base serializer class on which msgpack and json serializers (and potentially others) are built.

Subclasses must be registered by adding to the `all_serializers` global.

Supports ndarray, date, datetime, and bytes for transfer in addition to the standard types supported by json and msgpack. All other types are converted to an object proxy that can be used to access methods / attributes of the object remotely (this requires that the object be owned by an RPC server).

Note that tuples are converted to lists in transit. See: https://github.com/msgpack/msgpack-python/issues/98

decode(*dct*)

Convert from serializable objects back to original types.

dumps(*obj*)

Convert obj to serialized string.

encode (*obj*)
Convert various types to serializable objects.

Provides support for ndarray, datetime, date, and None. Other types are converted to proxies.

loads (*msg*)
Convert from serialized string to python object.

Proxies that reference objects owned by the server are converted back into the local object. All other proxies are left as-is.

5.2.6 Logging tools

These tools allow log records and unhandled exceptions to be forwarded to a central log server. Basic usage consists of:

1. Start a log server in any process using `start_log_server()`.
2. Attach a handler to the root logger (see Python logging documentation). If the log server is running in a process that can output to a terminal, then `RPCLogHandler` can be used to display log records color-coded by source.
3. Set the log level of the root logger. Using INFO or DEBUG levels will reveal details about RPC communications between processes.
4. In the remote process set the log level and call `set_host_name()`, `set_process_name()`, `set_thread_name()`, and `set_logger_address()` (note that `ProcessSpawner` handles this step automatically).

`pyacq.core.rpc.log.`**start_log_server** (*logger*)
Create a global log server and attach it to a logger.

Use *get_logger_address()* to return the socket address for the server after it has started. On a remote process, call *set_logger_address()* to connect it to the server. Then all messages logged remotely will be forwarded to the server and handled by the logging system there.

class `pyacq.core.rpc.log.`**LogServer** (*logger*, *address='tcp://127.0.0.1:*'*, *sort=True*)
Thread for receiving log records via zmq socket.

Messages are immediately passed to a python logger for local handling.

Parameters

logger [Logger] The python logger that should handle incoming messages.

run ()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class `pyacq.core.rpc.log.`**LogSender** (*address=None*, *logger=None*)
Handler for forwarding log messages to a remote LogServer via zmq socket.

Instances of this class can be attached to any python logger using *logger.addHandler(log_sender)*.

This can be used with *LogServer* to collect log messages from many remote processes to a central logger.

Note: We do not use RPC for this because we have to avoid generating extra log messages.

Parameters

address [str | None] The socket address of a log server. If None, then the sender is not connected to a server and *connect()* must be called later.

logger [str | None] The name of the python logger to which this handler should be attached. If None, then the handler is not attached (use ‘’ for the root logger).

close()
Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

connect(*addr*)
Set the address of the LogServer to which log messages should be sent. This value should be acquired from *log_server.address* or *get_logger_address()*.

handle(*record*)
Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

class pyacq.core.rpc.log.**RPCLogHandler**(*stream=<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*)
StreamHandler that sorts incoming log records by their creation time and writes to stderr. Messages are also colored by their log level and the host/process/thread that created the record.

Credit: https://gist.github.com/kergoth/813057

Parameters

stream [file-like] The stream to which messages should be sent. The default is sys.stderr.

pyacq.core.rpc.log.**log_exceptions**()
Install a hook that creates log messages from unhandled exceptions.

pyacq.core.rpc.log.**set_host_name**(*name*)
Set the name of this host used for logging.

pyacq.core.rpc.log.**get_host_name**()
Return the name of this host used for logging.

pyacq.core.rpc.log.**set_process_name**(*name*)
Set the name of this process used for logging.

pyacq.core.rpc.log.**get_process_name**()
Return the name of this process used for logging.

pyacq.core.rpc.log.**set_thread_name**(*name*, *tid=None*)
Set the name of a thread used for logging.

If no thread ID is given, then the current thread’s ID is used.

pyacq.core.rpc.log.**get_thread_name**(*tid=None*)
Return the name of a thread used for logging.

If no thread ID is given, then the current thread’s ID is used.

pyacq.core.rpc.log.**set_logger_address**(*addr*)
Set the address to which all log messages should be sent.

This function creates a global LogSender and attaches it to the root logger.

pyacq.core.rpc.log.**get_logger_address**()
Return the address of the LogServer used by this process.

If a LogServer has been created in this process, then its address is returned. Otherwise, the last address set with *set_logger_address()* is used.

5.3 Device Nodes

5.3.1 Audio

class pyacq.devices.**PyAudio**(***kargs*)
Simple wrapper around PyAudio for input and output to audio devices.

check_input_specs()
This method is called during *Node.initialize()* and may be reimplemented by subclasses to ensure that inputs are correctly configured before the node is started.

In case of misconfiguration, this method must raise an exception.

check_output_specs()
This method is called during *Node.initialize()* and may be reimplemented by subclasses to ensure that outputs are correctly configured before the node is started.

In case of misconfiguration, this method must raise an exception.

configure(**args*, ***kwargs*)

Parameters

nb_channel [int] Number of audio channels

sample_rate: float Sample rate. This value is rounded to integer.

input_device_index [int or None] Input device index (see *list_device_specs()* and pyaudio documentation). If None then no recording will be requested from the device, and the node will have no output.

output_device_index: in or None Output device index (see *list_device_specs()* and pyaudio documentation). If None then no playback will be requested from the device, and the node will have no input.

format [str in ('int16', 'int32' or 'float32')] Internal data format for pyaudio.

chunksize [int (1024 by default)] Size of each chunk. Smaller chunks result in lower overall latency, but may also cause buffering issues (cracks/pops in sound).

default_input_device()
Return the index of the default input device.

default_output_device()
Return the index of the default output device.

5.3.2 Cameras

class pyacq.devices.**WebCamAV**(***kargs*)
Simple webcam device using the *av* python module, which is a wrapper around ffmpeg or libav.

See http://mikeboers.github.io/PyAV/index.html.

class pyacq.devices.**WebCamImageIO**(***kargs*)
Simple webcam device using the imageio python module.

5.3.3 EEG

class `pyacq.devices.brainampsocket.`**`BrainAmpSocket`**(***kargs*)

BrainAmp EEG amplifier from Brain Products http://www.brainproducts.com/.

This class is a bridge between pyacq and the socket-based data streaming provided by the Vision recorder acquisition software.

`after_output_configure`(*outputname*)

This method is called when one of the Node's outputs has been configured.

It may be reimplemented by subclasses.

class `pyacq.devices.`**`Emotiv`**(***kargs*)

Simple eeg emotiv device to access eeg, impedances and gyro data in a Node.

Reverse engineering and original crack code written by Cody Brocious (http://github.com/daeken) Kyle Machulis (http://github.com/qdot) Many thanks for their contribution.

Emotiv USB emit 32-bytes reports at a rate of 128Hz, encrypted via AES see https://github.com/qdot/emokit/blob/master/doc/emotiv_protocol.asciidoc for more details

`after_output_configure`(*outputname*)

This method is called when one of the Node's outputs has been configured.

It may be reimplemented by subclasses.

class `pyacq.devices.`**`OpenBCI`**(***kargs*)

This class is a bridge between Pyacq and the 32bit board OpenBCI amplifier from the open source project http://openbci.com. Daisy board version for now

#TODO : this is a very basic code to grab data from 8 channel Daisy OpenBCI board. # next version will improve dialog with the board and auto-initialisation

`after_output_configure`(*outputname*)

This method is called when one of the Node's outputs has been configured.

It may be reimplemented by subclasses.

5.3.4 Testing

class `pyacq.devices.`**`NumpyDeviceBuffer`**(***kargs*)

A fake analogsignal device.

This node streams data from a predefined buffer in an endless loop.

`after_output_configure`(*outputname*)

This method is called when one of the Node's outputs has been configured.

It may be reimplemented by subclasses.

`configure`(**args*, ***kwargs*)

Parameters

nb_channel: int Number of output channels.

sample_interval: float Time duration of a single data sample. This determines the rate at which data is sent.

chunksize: int Length of chunks to send.

buffer: array Data to send. Must have *buffer.shape[0] == nb_channel*.

5.4 Visualization Nodes

5.4.1 Analog signal visualizers

class pyacq.viewers.**QOscilloscope**(***kargs*)

Continuous, multi-channel oscilloscope based on Qt and pyqtgraph.

class pyacq.viewers.**QTriggeredOscilloscope**(***kargs*)

5.4.2 Spectral visualizers

class pyacq.viewers.**QTimeFreq**(***kargs*)

Class for visualizing the frequency spectrogram with a Morlet continuous wavelet transform.

This allows better visualization than the standard FFT spectrogram because it provides better temporal resolution for high-frequency signals without sacrificing frequency resolution for low-frequency signals. See https://en.wikipedia.org/wiki/Morlet_wavelet

This class internally uses one TimeFreqWorker per channel, which allows multiple signals to be transformed in parallel.

The node operates in one of 2 modes:

- Each TimeFreqWorker lives in the same QApplication as the QTimeFreq node (nodegroup_friends=None).
- Each TimeFreqWorker is spawned in another NodeGroup to distribute the load (nodegroup_friends=[some_list_of_nodegroup]).

This viewer needs manual tuning for performance: small refresh_interval, high number of freqs, hight f_stop, and high xsize can all lead to heavy CPU load.

This node requires its input stream to use:

- `transfermode==sharedarray`
- `axisorder==[1,0]`

If the input stream does not meet these requirements, then a StreamConverter will be created to proxy the input.

QTimeFreq can be configured on the fly by changing QTimeFreq.params and QTimeFreq.by_channel_params. By default, double-clicking on the viewer will open a GUI dialog for these parameters.

Usage:

```
viewer = QTimeFreq()
viewer.configure(with_user_dialog=True, nodegroup_friends=None)
viewer.input.connect(somedevice.output)
viewer.initialize()
viewer.show()
viewer.start()

viewer.params['nb_column'] = 4
viewer.params['refresh_interval'] = 1000
```

5.4.3 Image visualizers

class pyacq.viewers.**ImageViewer**(***kargs*)

A simple image viewer using pyqtgraph.

5.5 Signal Processing Nodes

5.5.1 Triggering Nodes

class pyacq.dsp.**AnalogTrigger**(*parent=None*, ***kargs*)

No so efficient but quite robust trigger on analogsignal.

This act like a standart trigger with a threshold and a front. The channel can be selected among all.

All params can be set online via AnalogTrigger.params['XXX'] = ...

The main feature is the debounce mode combinated with debounce_time:

- **'no-debounce'** all crossing threshold is a trigger
- **'after-stable'** when interval between a series of triggers is too short, the **lastet one** is taken is account.
- **'before-stable'** when interval between a series of triggers is too short, the **first one** is taken is account.

check_input_specs()

This method is called during *Node.initialize()* and may be reimplemented by subclasses to ensure that inputs are correctly configured before the node is started.

In case of misconfiguration, this method must raise an exception.

class pyacq.dsp.**DigitalTrigger**(*parent=None*, ***kargs*)

check_input_specs()

This method is called during *Node.initialize()* and may be reimplemented by subclasses to ensure that inputs are correctly configured before the node is started.

In case of misconfiguration, this method must raise an exception.

class pyacq.dsp.**TriggerAccumulator**(*parent=None*, ***kargs*)

Node that accumulate in a ring buffer chunk of a multi signals on trigger events.

This Node have no output because the stack size of signals chunks is online configurable. sharred memory is difficult because shape can change.

The internal self.stack have 3 dims: 0 - trigger 1 - nb channel 2 - times

The self.total_trig indicate the number of triggers since the last reset_stack().

TriggerAccumulator.params['stask_size'] control the number of event in the stack. Note the stask behave as a ring buffer along the axis 0. So if self.total_trig>stask_size you need to play with modulo to acces the last event.

On each new chunk this new_chunk is emmited. Note that this do not occurs on new trigger but a bit after when the right_sweep is reached on signals stream.

after_input_connect(*inputname*)

This method is called when one of the Node's inputs has been connected.

It may be reimplemented by subclasses.

5.5.2 Filtering nodes

class pyacq.dsp.**OverlapFiltfilt**(*parent=None*, ***kargs*)

Node for filtering with forward-backward method (filtfilt) using second order (sos) coefficient and a sliding, overlapping window.

Because the signal is filtered piecewise, the result will differ slightly from the ideal case, in which the entire signal would be filtered over all time at once. To ensure accurate results, the chunksize and overlapsize parameters must be chosen carefully: a small chunksize will affect low frequencies, and a small overlapsize may result in transients at the border between chunks. We recommend comparing the output of this node to an ideal offline filter to ensure that the residuals are acceptably small.

The chunksize need to be fixed. For overlapsize there are 2 cases:

1. `overlapsize < chunksize/2` : natural case; each chunk partially overlaps. The overlapping regions are on the ends of each chunk, whereas the central part of the chunk has no overlap.
2. `overlapsize>chunksize/2` : chunks are fully overlapping; there is no central part.

In the 2 cases, for each arrival of a new chunk at `[-chunksize:]`, the computed chunk at `[-(chunksize+overlapsize):-overlapsize]` is released.

The `coefficients.shape` must be (nb_section, 6).

If pyopencl is avaible you can use `SosFilter.configure(engine='opencl')`. In that case the coefficients.shape can also be (nb_channel, nb_section, 6) this helps for having different filters on each channel.

The opencl engine inernally requires data to be in (channel, sample) order. If the input data does not have this order, then it must be copied and performance will be affected.

after_input_connect(*inputname*)
This method is called when one of the Node's inputs has been connected.

It may be reimplemented by subclasses.

class pyacq.dsp.**SosFilter**(*parent=None*, ***kargs*)
Node for filtering multi channel signals. This uses a second order filter, it is a casde of IIR filter of order 2.

Example:

```
dev = NumpyDeviceBuffer()
dev.configure(...)
dev.output.configure(...)
dev.initialize(...)

f1, f2 = 40., 60.
coefficients = scipy.signal.iirfilter(7, [f1/sample_rate*2, f2/sample_rate*2],
            btype='bandpass', ftype='butter', output='sos')
filter = SosFilter()
filter.configure(coefficients=coefficients)
filter.input.connect(dev.output)
filter.output.configure(...)
filter.initialize()
```

The `coefficients.shape` must be (nb_section, 6).

If pyopencl is avaible you can use `SosFilter.configure(engine='opencl')`. In that case the coefficients.shape can also be (nb_channel, nb_section, 6) this helps for having different filters on each channel.

The opencl engine inernally requires data to be in (channel, sample) order. If the input data does not have this order, then it must be copied and performance will be affected.

after_input_connect(*inputname*)
This method is called when one of the Node's inputs has been connected.

It may be reimplemented by subclasses.

CHAPTER 6

Examples

Contents:

6.1 Local and remote Nodes

```
"""
Local and remote Nodes

This example demonstrates the use of Node instances both in the local process
and in a remote process. In either case, the way we interact with the Node is
essentially the same.
"""

from pyacq import create_manager, ImageViewer, WebCamAV
from pyqtgraph.Qt import QtCore, QtGui
import time
import pyqtgraph as pg




def dev_remote_viewer_local():
    man = create_manager()

    # this create the dev in a separate process (NodeGroup)
    nodegroup = man.create_nodegroup()

    dev = nodegroup.create_node('WebCamAV', name = 'cam0')
    dev.configure(camera_num = 0)
    dev.output.configure(protocol = 'tcp', interface = '127.0.0.1', transfermode =
↪'plaindata')
    dev.initialize()
```

(continues on next page)

(continued from previous page)

```
    #view is a Node in local QApp
    app = pg.mkQApp()

    viewer = ImageViewer()
    viewer.configure()
    viewer.input.connect(dev.output)
    viewer.initialize()
    viewer.show()

    dev.start()
    viewer.start()

    app.exec_()


def dev_local_viewer_local():
    # no manager
    # device + view is a Node in local QApp
    # Nodes are controled directly

    app = pg.mkQApp()

    dev = WebCamAV()
    dev.configure(camera_num = 0)
    dev.output.configure(protocol = 'tcp', interface = '127.0.0.1', transfermode =
↪'plaindata')
    dev.initialize()


    viewer = ImageViewer()
    viewer.configure()
    viewer.input.connect(dev.output)
    viewer.initialize()
    viewer.show()

    dev.start()
    viewer.start()

    app.exec_()


def dev_remote_viewer_remote():
    # no QApp all Nodes are remoted even the viewer.
    # note that dev and viewer are in the same NodeGroup
    # so they are in the same process

    man = create_manager()
    nodegroup = man.create_nodegroup()

    dev = nodegroup.create_node('WebCamAV', name = 'cam0')
    dev.configure(camera_num = 0)
    dev.output.configure(protocol = 'tcp', interface = '127.0.0.1', transfermode =
↪'plaindata')
    dev.initialize()

    viewer = nodegroup.create_node('ImageViewer', name = 'viewer0')
```

(continues on next page)

(continued from previous page)

```
    viewer.configure()
    viewer.input.connect(dev.output)
    viewer.initialize()
    viewer.show()

    dev.start()
    viewer.start()


    time.sleep(10.)


# uncomment one if this 3 lines and compare the process number
dev_remote_viewer_local()
#dev_local_viewer_local()
#dev_remote_viewer_remote()
```

6.2 Noise generator node

```
# -*- coding: utf-8 -*-
# Copyright (c) 2016, French National Center for Scientific Research (CNRS)
# Distributed under the (new) BSD License. See LICENSE for more info.
"""
Noise generator node

Simple example of a custom Node class that generates a stream of random
values.

"""
import numpy as np

from pyacq.core import Node, register_node_type
from pyqtgraph.Qt import QtCore, QtGui


class NoiseGenerator(Node):
    """A simple example node that generates gaussian noise.
    """
    _output_specs = {'signals': dict(streamtype='analogsignal', dtype='float32',
                                     shape=(-1, 1), compression='')}

    def __init__(self, **kargs):
        Node.__init__(self, **kargs)
        self.timer = QtCore.QTimer(singleShot=False)
        self.timer.timeout.connect(self.send_data)

    def _configure(self, chunksize=100, sample_rate=1000.):
        self.chunksize = chunksize
        self.sample_rate = sample_rate

        self.output.spec['shape'] = (-1, 1)
        self.output.spec['sample_rate'] = sample_rate
        self.output.spec['buffer_size'] = 1000
```

(continues on next page)

(continued from previous page)

```
    def _initialize(self):
        self.head = 0

    def _start(self):
        self.timer.start(int(1000 * self.chunksize / self.sample_rate))

    def _stop(self):
        self.timer.stop()

    def _close(self):
        pass

    def send_data(self):
        self.head += self.chunksize
        self.output.send(np.random.normal(size=(self.chunksize, 1)).astype('float32'),
↪ index=self.head)


# Not necessary for this example, but registering the node class would make it
# easier for us to instantiate this type of node in a remote process via
# Manager.create_node()
register_node_type(NoiseGenerator)



if __name__ == '__main__':
    from pyacq.viewers import QOscilloscope
    app = QtGui.QApplication([])

    # Create a noise generator node
    ng = NoiseGenerator()
    ng.configure()
    ng.output.configure(protocol='inproc', transfermode='sharedmem')
    ng.initialize()

    # Create an oscilloscope node to view the noise stream
    osc = QOscilloscope()
    osc.configure(with_user_dialog=True)
    osc.input.connect(ng.output)
    osc.initialize()
    osc.show()

    # start both nodes
    osc.start()
    ng.start()
```

6.3 Simple webcam viewer

```
"""
Simple webcam viewer

Streams video frames from a WebCamAV Node to an ImageViewer Node.
"""
```

(continues on next page)

(continued from previous page)

```python
from pyacq import create_manager, ImageViewer
from pyqtgraph.Qt import QtCore, QtGui


man = create_manager()

# this create the dev in a separate process (NodeGroup)
nodegroup = man.create_nodegroup()
dev = nodegroup.create_node('WebCamAV', name = 'cam0')
dev.configure(camera_num = 0)
dev.output.configure(protocol = 'tcp', interface = '127.0.0.1', transfermode =
↪'plaindata')
dev.initialize()

#view is a Node in local QApp
app = QtGui.QApplication([])

viewer = ImageViewer()
viewer.configure()
viewer.input.connect(dev.output)
viewer.initialize()
viewer.show()

dev.start()
viewer.start()

app.exec_()
```

6.4 PyAudio oscilloscope (remote)

```python
"""
PyAudio oscilloscope (remote)

Simple demonstration of streaming data from a PyAudio device to a QOscilloscope
viewer.

Both device and viewer nodes are created locally without a manager.
"""

import pyqtgraph as pg

from pyacq.viewers import QOscilloscope
import pyacq.core.rpc as rpc

app = pg.mkQApp()


# Create PyAudio device node in remote process
dev_proc = rpc.ProcessSpawner()
dev = dev_proc.client._import('pyacq.devices.audio_pyaudio').PyAudio()

# Print a list of available input devices (but ultimately we will just use the
# default device).
```

(continues on next page)

(continued from previous page)

```
default_input = dev.default_input_device()
print("\nAvaliable devices:")
for device in dev.list_device_specs():
    index = device['index']
    star = "*" if index == default_input else " "
    print("  %s %d: %s" % (star, index, device['name']))

# Configure PyAudio device with a single (default) input channel.
dev.configure(nb_channel=1, sample_rate=44100., input_device_index=default_input,
              format='int16', chunksize=1024)
dev.output.configure(protocol='tcp', interface='127.0.0.1', transfermode='plaindata')
dev.initialize()


# Create an oscilloscope to display data.
viewer = QOscilloscope()
viewer.configure(with_user_dialog = True)

# Connect audio stream to oscilloscope
viewer.input.connect(dev.output)

viewer.initialize()
viewer.show()
viewer.params['decimation_method'] = 'min_max'
viewer.by_channel_params['Signal0', 'gain'] = 0.001

# Start both nodes
dev.start()
viewer.start()

app.exec_()
```

6.5 PyAudio wavelet spectrogram

```
"""
PyAudio wavelet spectrogram

Streams audio data to a QTimeFreq Node, which displays a frequency spectrogram
from a Morlet continuous wavelet transform.
"""

from pyacq.devices.audio_pyaudio import PyAudio
from pyacq.viewers import QTimeFreq
from pyacq.core import create_manager
import pyqtgraph as pg


# Start Qt application
app = pg.mkQApp()


# Create a manager to spawn worker process to record and process audio
man = create_manager()
ng = man.create_nodegroup()
```

(continues on next page)

(continued from previous page)

```python

# Create PyAudio device node in remote process
dev = ng.create_node('PyAudio')

# Configure PyAudio device with a single (default) input channel.
default_input = dev.default_input_device()
dev.configure(nb_channel=1, sample_rate=44100., input_device_index=default_input,
              format='int16', chunksize=1024)
dev.output.configure(protocol='tcp', interface='127.0.0.1', transfermode='plaindata')
dev.initialize()


# We are only recording a single audio channel, so we create one extra
# nodegroup for processing TFR. For multi-channel signals, create more
# nodegroups.
workers = [man.create_nodegroup()]


# Create a viewer in the local application, using the remote process for
# frequency analysis
viewer = QTimeFreq()
viewer.configure(with_user_dialog=True, nodegroup_friends=workers)
viewer.input.connect(dev.output)
viewer.initialize()
viewer.show()

viewer.params['refresh_interval'] = 100
viewer.params['timefreq', 'f_start'] = 50
viewer.params['timefreq', 'f_stop'] = 5000
viewer.params['timefreq', 'deltafreq'] = 500
viewer.by_channel_params['Signal0', 'clim'] = 2500


# Start both nodes
dev.start()
viewer.start()


if __name__ == '__main__':
    import sys
    if sys.flags.interactive == 0:
        app.exec_()
```

6.6 PyAudio triggered oscilloscope

```python
"""
PyAudio triggered oscilloscope

Streams audio data to a QTriggeredOscilloscope Node, which displays the
incoming waveform time-aligned to the rising phase of the sound wave.
"""

from pyacq.devices.audio_pyaudio import PyAudio
```

(continues on next page)

(continued from previous page)

```
from pyacq.viewers import QTriggeredOscilloscope
import pyqtgraph as pg


# Start Qt application
app = pg.mkQApp()


# Create PyAudio device node
dev = PyAudio()

# Print a list of available input devices (but ultimately we will just use the
# default device).
default_input = dev.default_input_device()
print("\nAvaliable devices:")
for device in dev.list_device_specs():
    index = device['index']
    star = "*" if index == default_input else " "
    print("  %s %d: %s" % (star, index, device['name']))

# Configure PyAudio device with a single (default) input channel.
dev.configure(nb_channel=1, sample_rate=44100., input_device_index=default_input,
              format='int16', chunksize=1024)
dev.output.configure(protocol='tcp', interface='127.0.0.1', transfermode='plaindata')
dev.initialize()


# Create a triggered oscilloscope to display data.
viewer = QTriggeredOscilloscope()
viewer.configure(with_user_dialog = True)

# Connect audio stream to oscilloscope
viewer.input.connect(dev.output)

viewer.initialize()
viewer.show()
#viewer.params['decimation_method'] = 'min_max'
#viewer.by_channel_params['Signal0', 'gain'] = 0.001

viewer.trigger.params['threshold'] = 1.
viewer.trigger.params['debounce_mode'] = 'after-stable'
viewer.trigger.params['front'] = '+'
viewer.trigger.params['debounce_time'] = 0.1
viewer.triggeraccumulator.params['stack_size'] = 3
viewer.triggeraccumulator.params['left_sweep'] = -.2
viewer.triggeraccumulator.params['right_sweep'] = .5


# Start both nodes
dev.start()
viewer.start()


if __name__ == '__main__':
    import sys
    if sys.flags.interactive == 0:
        app.exec_()
```

6.7 PyAudio oscilloscope (local)

```
"""
PyAudio oscilloscope (local)

Simple demonstration of streaming data from a PyAudio device to a QOscilloscope
viewer.

Both device and viewer nodes are created locally without a manager.
"""

from pyacq.devices.audio_pyaudio import PyAudio
from pyacq.viewers import QOscilloscope
import pyqtgraph as pg


# Start Qt application
app = pg.mkQApp()


# Create PyAudio device node
dev = PyAudio()

# Print a list of available input devices (but ultimately we will just use the
# default device).
default_input = dev.default_input_device()
print("\nAvaliable devices:")
for device in dev.list_device_specs():
    index = device['index']
    star = "*" if index == default_input else " "
    print("  %s %d: %s" % (star, index, device['name']))

# Configure PyAudio device with a single (default) input channel.
dev.configure(nb_channel=1, sample_rate=44100., input_device_index=default_input,
              format='int16', chunksize=1024)
dev.output.configure(protocol='tcp', interface='127.0.0.1', transfermode='plaindata')
dev.initialize()


# Create an oscilloscope to display data.
viewer = QOscilloscope()
viewer.configure(with_user_dialog = True)

# Connect audio stream to oscilloscope
viewer.input.connect(dev.output)

viewer.initialize()
viewer.show()
viewer.params['decimation_method'] = 'min_max'
viewer.by_channel_params['Signal0', 'gain'] = 0.001

# Start both nodes
dev.start()
viewer.start()


if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
    import sys
    if sys.flags.interactive == 0:
        app.exec_()
```

6.8 Stream monitor

```
"""
Stream monitor

A simple node that monitors activity on an input stream and prints details about␣
↪packets
received.

"""
import numpy as np
from pyqtgraph.Qt import QtCore, QtGui

from pyacq.core import Node, register_node_type
from pyacq.core.tools import ThreadPollInput


class StreamMonitor(Node):
    """
    Monitors activity on an input stream and prints details about packets
    received.
    """
    _input_specs = {'signals': {}}

    def __init__(self, **kargs):
        Node.__init__(self, **kargs)

    def _configure(self):
        pass

    def _initialize(self):
        # There are many ways to poll for data from the input stream. In this
        # case, we will use a background thread to monitor the stream and emit
        # a Qt signal whenever data is available.
        self.poller = ThreadPollInput(self.input, return_data=True)
        self.poller.new_data.connect(self.data_received)

    def _start(self):
        self.poller.start()

    def data_received(self, ptr, data):
        print("Data received: %d %s %s" % (ptr, data.shape, data.dtype))



# Not necessary for this example, but registering the node class would make it
# easier for us to instantiate this type of node in a remote process via
# Manager.create_node()
register_node_type(StreamMonitor)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    from pyacq.devices import NumpyDeviceBuffer
    app = QtGui.QApplication([])

    # Create a data source. This will continuously emit chunks from a numpy
    # array.
    data = np.random.randn(2500, 7).astype('float64')
    dev = NumpyDeviceBuffer()
    # Configure the source such that it emits 50-element chunks twice per second.
    dev.configure(nb_channel=7, sample_interval=0.01, chunksize=50, buffer=data)
    dev.output.configure(protocol='tcp', interface='127.0.0.1', transfermode=
↪'plaindata')
    dev.initialize()

    # Create a monitor node
    mon = StreamMonitor()
    mon.configure()
    mon.input.connect(dev.output)
    mon.initialize()

    # start both nodes
    mon.start()
    dev.start()
```

6.9 Custom RPC client

```
"""
Custom RPC client

Demonstrate the most simple use of zmq and json to create a client that
connects to an RPCServer. This provides a basic template for connecting
to pyacq from non-Python platforms.

One important note before we start: pyacq's remote API is not actually different
from its internal Python API. Any function you can call from within Python
can also be invoked remotely by RPC calls. The example below deals entirely
with pyacq's RPC protocol--how translate between the Python API and the raw
packets handled by zeroMQ.
"""

# First we will start a manager in a subprocess to test our client against
from pyacq.core import create_manager
manager = create_manager('rpc')
address = manager._rpc_addr


# --- From here on, we don't use any pyacq code ---
import json, zmq

# Here's how we connect to a new server (we will likely want to connect to
# multiple servers)
def create_socket(address, name):
```

(continues on next page)

(continued from previous page)

```
    """Return a ZeroMQ socket connected to an RPC server.

    Parameters
    ----------
    address : str
        The zmq interface where the server is listening (e.g.
        'tcp://127.0.0.1:5678')
    name : str
        A unique name identifying the client.
    """
    if isinstance(name, str):
        name = name.encode()
    socket = zmq.Context.instance().socket(zmq.DEALER)
    socket.setsockopt(zmq.IDENTITY, name)
    socket.setsockopt(zmq.RCVTIMEO, 5000)  # 5 sec timeout

    # Connect the socket to the server
    if isinstance(address, str):
        address = address.encode()
    socket.connect(address)

    # Ping the server until it responds to make sure we are connected.
    ping(socket)
    print("\nConnected to server @ %s" % address)

    return socket


# Here's how we have to format all request messages that we send to RPC servers
next_req_id = 0
def send(socket, action, opts=None, request_response=True, return_type='auto'):
    """Send a request to an RPC server.

    Parameters
    ----------
    socket : zmq socket
        The ZeroMQ socket that is connected to the server.
    action : str
        Name of action server should perform. See :func:`RPCClient.send()` for
        a list of actions and their associated options.
    opts : dict or None
        An optional dict of options specifying the behavior of the action.
    request_response : bool
        If True, then the server is asked to send a response.
    return_type : str
        'proxy' to force the server to send return values by proxy, or 'auto'
        to allow the server to decide whether to return by proxy or by value.
    """
    global next_req_id

    # If we want the server to send a response, then we must supply a unique ID
    # for the request. Otherwise, send -1 as the request ID to indicate that
    # the server should not send a reply.
    if request_response:
        req_id = next_req_id
        next_req_id += 1
    else:
```

(continues on next page)

(continued from previous page)

```
        req_id = -1

    # Serialize opts if it was specified, otherwise send an empty string.
    if opts is None:
        opts_str = b''
    else:
        opts_str = json.dumps(opts).encode()

    # Tell the server which serializer we are using
    ser_type = b'json'

    # Send the request as a multipart message
    msg = [str(req_id).encode(), action.encode(), return_type.encode(), ser_type,␣
↪opts_str]
    socket.send_multipart(msg)

    # Print so we can see what the final json-encoded message looks like
    msg = '\n'.join(['    ' + m.decode() for m in msg])
    print("\n>>> send to %s:\n%s" % (socket.last_endpoint.decode(), msg))

    # Return the request ID we can use to listen for a response later.
    return req_id


# ..And here is how we receive responses from the server.
def recv(socket):
    # Wait for a response or a timeout.
    try:
        msg = socket.recv().decode()
    except zmq.error.Again:
        raise TimeoutError('Timed out while waiting for server response.')

    # Print so we can see what the json-encoded message looks like
    print("\n<<< recv from %s:\n    %s" % (socket.last_endpoint.decode(), msg))

    # Unserialize the response
    msg = json.loads(msg)

    # Check for error
    if msg.get('error', None) is not None:
        traceback = ''.join(msg['error'][1])
        raise RuntimeError("Exception in remote process:\n%s" % traceback)

    # NOTE: msg also contains the key 'req_id', which should be used to verify
    # that the message received really does correspond to a particular request.
    # We're skipping that here for simplicity.

    return msg['rval']


def get_attr(socket, obj, attr_name):
    """Return an attribute of an object owned by a remote server.

    Parameters
    ----------
    socket : zmq socket
        A socket that is connected to the remote server.
```

(continues on next page)

(continued from previous page)

```
    obj : dict
        A dict that identifies the object owned by the server.
    attr_name : str
        The name of the attribute to return.
    """
    attr = obj.copy()
    attr['attributes'] = (attr_name,)
    send(socket, action='get_obj', opts={'obj': attr})
    return recv(socket)


def call_method(socket, obj, method_name, *args, **kwds):
    """Request that a remote server call a method on an object.

    Parameters
    ----------
    socket : zmq socket
        A socket that is connected to the remote server.
    obj : dict
        A dict that identifies the object owned by the server. This should have
        been returned by a previous request to the server.
    method_name : str
        The name of the method to call.
    args,kwargs :
        All further arguments are passed to the remote method call.
    """
    # modify object reference to point to its method instead.
    # (this is faster than using get_attr as defined above)
    func = obj.copy()
    func['attributes'] = (method_name,)
    send(socket, action='call_obj', opts={'obj': func, 'args': args, 'kwargs': kwds})
    return recv(socket)


def ping(socket):
    """Ping a server until it responds.

    This can be called to check that a functional connection to a server exists
    before making any other requests.
    """
    for i in range(3):
        req_id = send(socket, action='ping')
        try:
            resp = recv(socket)
            assert resp == 'pong'
            break
        except TimeoutError:
            pass
        if i == 2:
            raise RuntimeError("Did not receive any response from server at %s!"
                % socket.last_endpoint)




# Create a zmq socket with a unique name
socket = create_socket(address, 'my_custom_client')
```

(continues on next page)

(continued from previous page)

```
# Request a reference to the manager
send(socket, action='get_item', opts={'name': 'manager'})
manager = recv(socket)

# Ask the manager to create a nodegroup
nodegroup = call_method(socket, manager, 'create_nodegroup', name='my_nodegroup')

# Request from the manager a list of all available nodegroups
ng_list = call_method(socket, manager, 'list_nodegroups')
assert ng_list[0] == nodegroup

# Connect to the newly spawned nodegroup and ask it a question
ng_socket = create_socket(nodegroup['rpc_addr'], 'my_nodegroup_socket')
node_types = call_method(ng_socket, nodegroup, 'list_node_types')
print("\nAvailable node types: %s" % node_types)
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index