py-zmq-pipeline Documentation

Release 0.1.3

Alan Illing

July 30, 2014

Contents

CHAPTER 1

Introduction

Py-zmq-pipeline is as high level pipeline pattern implemented in Python with ZeroMQ.

The ZeroMQ Pipeline pattern is a data and task distribution pattern for distributed and parallel computing. The py-zmq-pipeline implementation abstracts away many details of ZeroMQ while keeping all the performance advantages the framework offers.

CHAPTER 2

Contents

2.1 Installation

2.1.1 Install from PyPI

Run the following from your terminal or GitBash (for windows users) to install from PyPI:

```
pip install py-zmq-pipeline
```

2.1.2 Install from source

Clone the git repository:

```
git clone git@github.com/ailling/py-zmq-pipeline.git
```

Install:

```
cd py-zmq-pipeline
python setup.py install
```

In both cases it's recommended you install into a virtual environment using virtualenv or virtualenvwrapper.

2.1.3 Test installation

You should now be able to import zmqpipeline from your Python interpreter:

```
>>> import zmqpipeline
```

2.1.4 Running tests

py-zmq-pipeline uses pytest for running tests. It is included in the installation of py-zmq-pipeline.

To run the tests:

```
py.test tests
```

Current build status is available on GitHub Repository Homepage

2.2 Overview and features

2.2.1 Introduction

Py-zmq-pipeline is as high level pipeline pattern implemented in **Python_** with **ZeroMQ_**.

The ZeroMQ Pipeline pattern is a data and task distribution pattern for distributed and parallel computing. The py-zmq-pipeline implementation abstracts away many details of ZeroMQ while keeping all the performance advantages the framework offers.

This library focuses more on task distribution than data distribution, since distributing data creates a memory bottleneck and tends not to scale well.

Client workers are responsible for both retrieval and processing of data in this implementation.

2.2.2 Features

A good pipeline pattern implementation will use load balancing to ensure all workers are supplied with roughly the same amount of work. While ZeroMQ provides fair-queuing and uniform request distribution out of the box, it **does not automatically provide load balancing**. Py-zmq-pipeline **does have load balancing** built in automatically.

Additional features include:

- Built-in single and multi-threaded workers
- **Task dependencies**
  - Allows you to construct dependency trees of tasks
- **Templated design pattern**
  - This means you inherit a class, provide your own implementation and invoke a run() method
- High performance with low overhead (benchmarks available in examples)
- **Reliability**
  - workers can die and be restarted without interrupting overall workflow.
  - repeatedly sent tasks are not re-processed once acknowledged
- **Fast serialization**
  - uses msgpack for highly efficient, dense and flexible packing of information, allowing py-zmq-pipeline to have a minimal footprint on the wire.
- Load-balanced
- Built in logging support for easier debugging

2.2.3 Overview

There are 4 components in py-zmq-pipeline:

- A *Distributor class*, responsible for pushing registered tasks to worker clients
- A *Task class*, an encapsulation of work that needs to be done and configures the distributor to do it
- A *Worker class*, a class that consumes computational resources to execute a given task instance
- A *Collector class*, a sink that accepts receipts of completed work and sends ACKs (acknowledgements) back to the distributor

Under the server / client paradigm the distributor, task and collector are server-side entities, while the worker is a client entity.

2.3 Tutorial

This tutorial will walk you through basic concepts of the py-zmq-pipeline library, using code examples of minimalistic implementaions along the way. For more realistic examples see the **examples_** section.

2.3.1 Distributor class

The distributor class does not need to be inherited from. Its behavior is defined at instantiation and through the registration of tasks.

The distributor is instantiated as:

```
Distributor(collector_endpoint, collector_ack_endpoint, [receive_metadata = False, [metadata_endpoint
```

If receive_metadata is True, a metadata endpoint must be provided. In this case the distributor will wait for metadata to be received until it begins processing tasks, and all metadata will be forwarded to the tasks. Tasks can then optionally forward metadata to the workers if needed.

All tasks that need to be distributed must be reigstered with the distributor before invoking run():

```
Distributor.register_task(MyTask)
```

Start the distributor with run():

```
dist = Distributor(...)
dist.run()
```

2.3.2 Task class

The task class is an abstract base class requiring the following implementations:

- **task_type: a valid *Task type***
  - determines the type of task. Task types must be registered with the TaskType class before the task is declared. See the documentation for *Task type* for more.
- **endpoint: a valid *Endpoint address***
  - the endpoint for the worker address. The worker will connect to this endpoint to receive data. See the documentation for *Endpoint address* for more.
- **dependencies: a list of *task-types* instances.**
  - Dependencies are tasks that must be complete before the given task can be executed
- **handle: invoked by the distributor to determine what information to forward to the worker.**
  - Must return either a dictionary or nothing. Other return types, such as list or string, will raise a TypeError.

The signature of the handle() method is:

```
@abstractmethod
def handle(self, data, address, msgtype):
    """
    Handle invocation by the distributor.
    :param data: Meta data, if provided, otherwise an empty dictionary
    :param address: The address of the worker data will be sent to.
    :param msgtype: The message type received from the worker. Typically zmqpipeline.messages.MESSAGE
    :return: A dictionary of data to be sent to the worker, or None, in which case the worker will re
    """
    pass
```

Optionally a task can override initialize() to setup the worker. This is particularly helpful when metadata is supplied.

The default implementation is to store the metadata on the task:

```
def initialize(self, metadata={}):
    """
    Initializes the task. Default implementation is to store metadata on the object instance
    :param metadata: Metadata received from the distributor
    :return:
    """
    self.metadata = metadata
```

A minimal task implementation looks like this:

```
import zmqpipeline
zmqpipeline.TaskType.register_type('MYTSK')

class MyTask(zmqpipeline.Task):
    task_type = zmqpipeline.TaskType('MYTSK)
    endpoint = zmqpipeline.EndpointAddress('ipc://worker.ipc')
    dependencies = []

    def handle(self, data, address, msgtype):
        """
        Simulates some work to be done
        :param data: Data received from distributor
        :param address: The address of the client where task output will be sent
        :param msgtype: the type of message received. Typically zmqpipeline.utils.messages.MESSAGE_T
        :return:
        """
        self.n_count += 1
        if self.n_count >= 100:
            # mark as complete after 100 executions.
            self.is_complete = True

        # return the work to be done on the worker
        return {
            'workload': .01
        }
```

2.3.3 Worker class

The worker is an abstract base class that requires the following to be defined:

- task_type: a valid *Task type*
- **endpoint: a valid *Endpoint address***

  - the worker will connect to this endpoint to receive tasks from the the distributor

- **collector_endpoint: a valid *Endpoint address***
  - the worker will connect to this endpoint to send output to. It should be the address of the collector endpoint
- handle_execution: a method for handling messages from the distributor.

The signature of the handle_execution() method is:

```
@abstractmethod
def handle_execution(self, data, *args, **kwargs):
    """
    Invoked in the worker's main loop. Override in client implementation
    :param data: Data provided as a dictionary from the distributor
    :param args:
    :param kwargs:
    :return: A dictionary of data to be passed to the collector, or None, in which case no data will
    """
    return {}
```

You can also optionally define a method: init_worker. By default it has no implementation:

```
def init_worker(self):
    pass
```

This method will be invoked after the worker advertises its availability for the first time and receives a message back from the distributor. Note that if the worker depends on one or more tasks, it won't receive an acknowledgement from the distributor and hence this method will not be invoked until after those dependent tasks have finished processing.

SingleThreadedWorker class

The single threaded worker is a pure implementation of the *Worker class* documented above.

MultiThreadedWorker class

The multi threaded worker implements the *Worker class* documented above but requires two pieces of information:

- **handle_thread_execution(): method for handling data forwarded by the worker.**
  - this is where data processing should take place in the multi threaded worker
- n_threads: the number of threads to utilize in the worker. Should be a positive integer.

being processed in handle_execution, work is intended to be handled by handle_thread_execution().

You must still implement handle_execution(), but its role is to forward data to the thread, possibly making modificiations or doing pre-processing before hand.

The signature of handle_thread_execution() is:

```
@abstractmethod
def handle_thread_execution(self, data, index):
    """
    Invoked in worker's thread. Override in client implementation
    :return:
    """
    return {}
```

A minimal implementation of the multi threaded worker is:

```
import zmqpipeline
import time
zmqpipeline.TaskType.register_type('MYTSK')

class MyWorker(zmqpipeline.MultiThreadedWorker):
    task_type = zmqpipeline.TaskType('MYTSK')
    endpoint = zmqpipeline.EndpointAddress('ipc://worker.ipc')
    collector_endpoint = zmqpipeline.EndpointAddress('ipc://collector.ipc')

    n_threads = 10
    n_executions = 0

    def handle_execution(self, data, *args, **kwargs):
        """
        Handles execution of the main worker
        :param data:
        :param args:
        :param kwargs:
        :return:
        """
        # forward all received data to the thread
        self.n_executions += 1
        return data


    def handle_thread_execution(self, data, index):
        workload = data['workload']
        time.sleep(workload)

        # returning nothing forwards no extra information to the collector
```

2.3.4 MetaDataWorker Class

Despite its name, MetaDataWorker doesn't inherit from the Worker base class.

It's a stand-alone abstract base class requiring the following implementations:

- **endpoint: a valid *Endpoint address* instance**
  - this is the address of the meta data worker and should be supplied to the distributor at instantiation when using meta data.
- get_metadata(): a method returning a dictionary of meta data

The signature of get_metadata() is:

```
@abstractmethod
def get_metadata(self):
    """
    Retrieves meta data to be sent to tasks and workers.
    :return: A dictionary of meta data
    """
    return {}
```

A typical use case for retrieving meta data is querying a database or introspecting live code.

To start the meta data worker, call the run() method. A minimal implementation of a meta data worker is provided below.

```
import zmqpipeline

class MyMetaData(zmqpipeline.MetaDataWorker):
    endpoint = zmqpipeline.EndpointAddress('ipc://metaworker.ipc')

    def get_metadata(self):
        """
        Returns meta data for illustrative purposes
        :return:
        """
        return {
            'meta_variable': 'my value',
        }

if __name__ == '__main__':
    instance = MyMetaData()
    instance.run()
```

2.3.5 Collector class

The collector is an abstract base class requiring implementations of the following:

- endpoint: a valid *Endpoint address*
- ack_endpoint: a valid *Endpoint address*
- handle_collection: a method to handle messages received by the worker

The signature of handle_collection() is:

```
@abstractmethod
def handle_collection(self, data, task_type, msgtype):
    """
    Invoked by the collector when data is received from a worker.
    :param data: Data supplied by the worker (a dictionary). If the worker doesn't return anything th
    :param task_type: The task type of the worker and corresponding task
    :param msgtype: The message type. Typically zmqpipeline.messages.MESSAGE_TYPE_DATA
    :return:
    """
    pass
```

You can optionally implement handle_finished(), which is invoked when the collector receives a termination signal from the distributor.

The signature of handle_finished() is:

```
def handle_finished(self, data, task_type):
    """
    Invoked by the collector when message
    :param data: Data received from the worker on a termination signal
    :param task_type: The task type of the worker and correspond task
    :return: None
    """
    pass
```

2.3.6 Endpoint address

A string that must be a valid endpoint address, otherwise a type error is thrown.

Endpoint address signature:

```
zmqpipeline.EndpointAddress(string)
```

Addresses must belong to one of the acceptable protocols to be considered valid. Accepted protocols are:

- tpc
- ipc
- inproc

tpc should be used for connecting code across machines. ipc (inter-process-communication) can be used for connecting two apps on the same machine. inproc can only be used for connecting threads to a process. It is significantly faster than tpc or ipc and used by default in the multi threaded worker.

2.3.7 Task type

A string that identifies a task and a worker. Tasks and workers specify a task type in one-to-one fashion. That is, one task type can be associated with one task and one worker. No more, no less. This allows zmqpipeline to coordinate tasks and workers.

Task types can be any valid string but must be registered with zmqpipeline before using them declaratively.

To register a task type:

```
zmqpipeline.TaskType.register_type('MY_TASK_TYPE')
```

You can now use this type as follows:

```
task_type = zmqpipeline.TaskType('MY_TASK_TYPE')
```

2.3.8 Messages

Messages are stand-alone data structures used by zmqpipeline internally for packing additional information along with the data being put on the wire. You shouldn't be interacting with the messages library directly - documentation is provided here for debugging purposes only.

Message type

Message types are defined in zmqpipeline.utils.messages

The available message types are:

```
* MESSAGE_TYPE_ACK: acknowledgement
* MESSAGE_TYPE_SUCCESS: success
* MESSAGE_TYPE_FAILURE: failure
* MESSAGE_TYPE_READY: ready
* MESSAGE_TYPE_END: termination
* MESSAGE_TYPE_DATA: data
* MESSAGE_TYPE_META_DATA: metadata
* MESSAGE_TYPE_EMPTY: empty message
```

Creating messages

Message signatures are defined as follows:

```
def create(data, tasktype, msgtype):
def create_data(task, data):
def create_metadata(metadata):
def create_ack(task = '', data=''):
def create_success(task = '', data=''):
def create_failure(task = '', data=''):
def create_empty(task = '', data=''):
def create_ready(task = '', data=''):
def create_end(task = '', data=''):
```

2.3.9 Logging

py-zmq-pipeline makes use of python's built in logging library to output information about what's going on- for example, connections being made, initializations, data messages being passed around, etc.

More about Python's logging libraries can be found here: Python Logging Facility.

py-zmq-pipeline logs to the following loggers:

- zmqpipeline.distributor. Logs made from the distributor clas
- zmqpipeline.collector. Logs made from the worker class
- zmqpipeline.task. Logs made from the task base class
- zmqpipeline.worker. Logs made from either the single threaded worker or the multi threaded worker class.
- zmqpipeline.metadataworker. Logs made from the meta data worker class

Note that it's recommended you extend this logging naming convention in your own implementations for organization. For example, zmqpipeline.task.FirstTask could refer to logs made from your first task, etc.

```
import zmqpipeline
zmqpipeline.configureLogging({
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        # console logger
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        },
        # a file handler
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': 'output.log',
            'formatter': 'verbose'
        },
    },
    'loggers': {
        'zmqpipeline.distributor': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
```

```
        },
        'zmqpipeline.task': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
        },
        'zmqpipeline.collector': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
        },
        'zmqpipeline.worker': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
        },
    },
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d %(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
})
```

2.4 Examples

These examples will illustrate using py-zmq-pipeline by distributing a workload to one or more workers. We'll sleep 10 milliseconds to simulate some work being done.

The examples covered are:

- *Example 1: Sequential execution* will provide a baseline how quickly a workload can be executed
- *Example 2: Single threaded worker* will illustrate using a basic worker and compare performance to the baseline
- *Example 3: Multi threaded worker* will illustrate using the multi threaded worker and compare performance
- *Example 4: Task dependencies* will illustrate how to construct tasks that depend on each other
- *Example 5: Using metadata* will illustrate how to use metadata to start the distributor
- *Example 6: Services* will cover services and relevant use cases
- *Example 7: Logging* will cover how to configure logging in py-zmq-pipeline

Machine specs are more or less irrelevant for the amount of work being simulated, but for the record these benchmarks were made on a 4 core Intel i7 processor with 8GB of ram, running Linux Mint 16.

2.4.1 Example 1: Sequential execution

This example is used to establish a benchmark for future tests.

In examples/sequential/benchmark.py:

```
import time

def main(n, delay):
    print 'running - iterations: %d - delay: %.3f' % (n, delay)

    start = time.time()

    for i in xrange(n):
        time.sleep(delay)

    m = (time.time() - start) * 1000
    expected = n * delay * 1000
    overhead = m / expected - 1.0
    print 'expected execution time: %.1f' % expected
    print 'sequential execution took: %.1f milliseconds' % m
    print 'overhead: %.1f%%' % (overhead * 100.0)

if __name__ == '__main__':
    main(100, .01)
    main(1000, .01)
    main(5000, .01)
```

Produces the following output:

```
running - iterations: 100 - delay: 0.010
expected execution time: 1000.0
sequential execution took: 1011.5 milliseconds
overhead: 1.2%
running - iterations: 1000 - delay: 0.010
expected execution time: 10000.0
sequential execution took: 10112.4 milliseconds
overhead: 1.1%
running - iterations: 5000 - delay: 0.010
expected execution time: 50000.0
sequential execution took: 50522.6 milliseconds
overhead: 1.0%
```

This benchmark establishes an overhead of about 1% for seqeuntially looping over a piece of work. The overhead is due to looping the for loop, maintaining an index, calling the generator, as well as measurement errors in the time() method.

If we observe, say a 5% overhead in future benchmarks, we're really taking about a 4% difference from the benchmark.

2.4.2 Example 2: Single threaded worker

As explained in the overview, there are 4 components in py-zmq-pipeline:

- Distributor, responsible for pushing registered tasks to worker clients
- Task, an encapsulation of work that needs to be done and configures the distributor to do it
- Worker, a class that consumes computational resources to execute a given task instance
- Collector, a sink that accepts receipts of completed work and sends ACKs (acknowledgements) back to the distributor

Under the server / client paradigm the distributor, task and collector are server-side entities, while the worker is a client entity.

First let's setup some settings for the app. In examples/singlethread/settings.py:

```
################################################
# PUT ZMQPIPELINE LIBRARY ON SYSTEM PATH
################################################
import os, sys

app_path = os.path.dirname(os.path.abspath(__file__))
examples_path = os.path.join(app_path, '..')
root_path = os.path.join(examples_path, '..')

if root_path not in sys.path:
    sys.path.insert(0, root_path)

################################################
# APP SETTINGS
################################################
import zmqpipeline

TASK_TYPE_CALC = 'C'
zmqpipeline.TaskType.register_type(TASK_TYPE_CALC)

COLLECTOR_ENDPOINT = 'tcp://localhost:5558'
COLLECTOR_ACK_ENDPOINT = 'tcp://localhost:5551'
WORKER_ENDPOINT = 'ipc://worker.ipc'
```

The first part of this file is just adding py-zmq-pipeline to the command line in case you decided to clone the project and you're running it from within the examples directory. It will be common to all settings files in subsequent examples.

In the app settings section we're defining a task type and registering it with the library. Tasks are associated with task types in one-to-one fashion and should represent a unit of isolated work to be done. Some task types may depend on one or more other types; we'll cover that in example 4.

Let's write the task to issue some work. In examples/singlethread/tasks.py:

```
import settings
import zmqpipeline

class CalculationTask(zmqpipeline.Task):
    task_type = zmqpipeline.TaskType(settings.TASK_TYPE_CALC)
    endpoint = zmqpipeline.EndpointAddress(settings.WORKER_ENDPOINT)
    dependencies = []
    n_count = 0

    def handle(self, data, address, msgtype):
        """
        Simulates some work to be done
        :param data: Data received from distributor
        :param address: The address of the client where task output will be sent
        :param msgtype: the type of message received. Typically zmqpipeline.utils.messages.MESSAGE_T
        :return:
        """
        self.n_count += 1
        if self.n_count >= 100:
            # mark as complete after 1000 executions. Should take a total of 10 seconds to run sequer
            self.is_complete = True

        # return the work to be done on the worker
        return {
            'workload': .01
        }
```

The distributor invokes a method on the task called handle(). This method should supply details about the work to be done and return it as a dictionary. That dictionary will be forwarded to the worker by the distributor.

Workers receive work by advertising their availability to the distributor. At that time the worker provides its address and message type. Message types are available in the API reference. The data parameter will typically be an empty dictionary; it will likely be used in future versions.

Finally, the task controls how much work to send, in this case 100 messages.

Here's an implementation of the worker, in examples/singlethread/worker.py:

```
import settings
import zmqpipeline
from zmqpipeline.utils import shutdown
import time

class MyWorker(zmqpipeline.SingleThreadedWorker):
    task_type = zmqpipeline.TaskType(settings.TASK_TYPE_CALC)
    endpoint = zmqpipeline.EndpointAddress(settings.WORKER_ENDPOINT)
    collector_endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ENDPOINT)

    n_executions = 0

    def handle_execution(self, data, *args, **kwargs):
        self.n_executions += 1

        workload = data['workload']
        time.sleep(workload)

        # returning nothing forwards no extra information to the collector

if __name__ == '__main__':
    worker = MyWorker()
    print 'worker running'
    worker.run()

    print 'finished - handled %d executions' % worker.n_executions
    shutdown.kill_current_process()
```

The single threaded worker must provide an implementation of handle_execution(). This method is invoked whenever data is received from the distributor. The contents of data is determined by the task corresponding to the worker's task_type.

In this worker implementation we keep track of the number of executions made by the worker. py-zmq-pipeline uses implements a load balancing pattern, so if there are M total tasks and N workers are started, each worker should be executed approximately M/N times.

Note that the worker's job to receive input and deliver output. It receives input from the distributor, the connection to which is listed as WORKER_ENDPOINT in the settings module. It delivers output to the collector, connected to by the collector_endpoint address.

Next we have to implement the collector. In examples/singlethread/collector.py:

```
import settings
import zmqpipeline
import time

class StopWatchCollector(zmqpipeline.Collector):
    endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ENDPOINT)
    ack_endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ACK_ENDPOINT)
```

```
    start_time = None
    end_time = None

    @property
    def total_msec(self):
        if not self.end_time or not self.start_time:
            return 0

        return (self.end_time - self.start_time) * 1000

    def handle_collection(self, data, task_type, msgtype):
        if not self.start_time:
            self.start_time = time.time()


    def handle_finished(self, data, task_type, msgtype):
        self.end_time = time.time()
        print 'collection took %.1f milliseconds' % self.total_msec


if __name__ == '__main__':
    collector = StopWatchCollector()
    print 'collector running'
    collector.run()

    print 'finished'
```

This collector acts as a simple stopwatch in order to assess the performance of the worker. handle_collection() is invoked whenever the collector receives data from a worker, and handle_finished() is invoked whenever the distributor sends a termination message. The collector automatically sends ACKs (acknowledgements) back to the distributor, but needs to be explicitly setup with the ack_endpoint address. The endpoint address is used to communicate with workers.

Note that every message from a worker is sent back to the distributor as an ACK. Due to the frequency of messages traveling from collector to distributor it's best to put the collector and distributor on the same machine, posssibly connected through the ipc protocol instead of tcp.

Finally setting up and running the distributor is simple. All tasks need to be registered with the distributor before instantiating it and collector endpoint addresses are provided to the constructor. In examples/singlethread/run_distributor.py:

```
import settings
import zmqpipeline
import tasks

if __name__ == '__main__':
    zmqpipeline.Distributor.register_task(tasks.CalculationTask)

    dist = zmqpipeline.Distributor(
        collector_endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ENDPOINT),
        collector_ack_endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ACK_ENDPOINT)
    )
    dist.run()
    print 'finished'
```

Normally the distributor, collector and workers can be started in any order. For this example, make sure to start the collector first otherwise the output of the stopwatch might not make sense.

Single threaded worker benchmarks

Running the above example yields the following.

- **One worker**
  - 100 tasks: 1060 milliseconds (6% overhead)
  - 1000 tasks: 10850 milliseconds (8.5% overhead)
  - 5000 tasks: 53940 milliseconds (7.8% overhead)
- **Two workers**
  - 100 tasks: 527 milliseconds (5.4% overhead)
  - 1000 tasks: 550 milliseconds (10% overhead)
  - 5000 tasks: 26900 milliseconds (7.6% overhead)

As expected, the overhead is slightly higher for more workers since there's now a greater coordination burden by the distributor. However, while doubling the number of workers reduces the total processing time by a 2X order of maginitude, the overhead doesn't change much. The load balancing implementation is well worth the expense.

2.4.3 Example 3: Multi threaded worker

Switching from a single threaded to a multi threaded worker is a matter of inherint from a different subclass:

```
import settings
import zmqpipeline
from zmqpipeline.utils import shutdown
import time

class MyWorker(zmqpipeline.MultiThreadedWorker):
    task_type = zmqpipeline.TaskType(settings.TASK_TYPE_CALC)
    endpoint = zmqpipeline.EndpointAddress(settings.WORKER_ENDPOINT)
    collector_endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ENDPOINT)

    n_threads = 10
    n_executions = 0

    def handle_execution(self, data, *args, **kwargs):
        """
        Handles execution of the main worker
        :param data:
        :param args:
        :param kwargs:
        :return:
        """
        # forward all received data to the thread
        self.n_executions += 1
        return data


    def handle_thread_execution(self, data, index):
        workload = data['workload']
        time.sleep(workload)

        # returning nothing forwards no extra information to the collector
```

```
if __name__ == '__main__':
    worker = MyWorker()
    print 'worker running'
    worker.run()

    print 'finished - handled %d executions' % worker.n_executions
    shutdown.kill_current_process()
```

A multithreaded worker requires you to implement two methods: handle_execution() and handle_thread_execution(). The former forwards data to the thread executor. In this example, we're not adding any data to what's received by the worker and simply making a note that the worker was executed. This time, the thread execution simulates the work as before.

Multi threaded worker benchmarks

- **One worker, 10 threads**
  - 100 tasks: 93 milliseconds (7% **gain**)
  - 1000 tasks: 1070 milliseconds (7% overhead)
  - 5000 tasks: 5430 milliseconds (8.6% overhead)
- **Two workers, 10 threads per worker**
  - 100 tasks: 51 milliseconds (2% overhead)
  - 1000 tasks: 560 milliseconds (12% overhead)
  - 5000 tasks: 2818 milliseconds (12.7% overhead)

Notice that with 100 tasks that take 10 milliseconds each running on 10 parallel threads would expect to take 100 total milliseconds to run. The benchmark with a single worker actually shows a gain over the expected processing time. This means the time it takes it pull 100 messages off the wire and relay it to the thread is less than 10 milliseconds, even though the threads themselves are load balanced. It works only for the low-volume case because the worker is able to pull a relatively large percentage of the workload (10%) at one time.

2.4.4 Example 4: Task dependencies

This example is similar to the single threaded worker example, except we now have two tasks: FirstTask and SecondTask. We require that FirstTask be executed before SecondTask.

The tasks are defined in examples/dependencies/tasks.py:

```
import settings
import zmqpipeline

class FirstTask(zmqpipeline.Task):
    task_type = zmqpipeline.TaskType(settings.TASK_TYPE_FIRST)
    endpoint = zmqpipeline.EndpointAddress(settings.FIRST_WORKER_ENDPOINT)
    dependencies = []
    n_count = 0

    def handle(self, data, address, msgtype):
        """
        Simulates some work to be done
        :param data:
        :param address:
        :param msgtype:
```

```
        :return:
        """
        self.n_count += 1
        if self.n_count >= 500:
            # mark as complete after 1000 executions. Should take a total of 10 seconds to run sequen
            self.is_complete = True

        # return the work to be done on the worker
        return {
            'workload': .01
        }


class SecondTask(zmqpipeline.Task):
    task_type = zmqpipeline.TaskType(settings.TASK_TYPE_SECOND)
    endpoint = zmqpipeline.EndpointAddress(settings.SECOND_WORKER_ENDPOINT)
    dependencies = [zmqpipeline.TaskType(settings.TASK_TYPE_FIRST)]
    n_count = 0

    def handle(self, data, address, msgtype):
        """
        Simulates some work to be done
        :param data:
        :param address:
        :param msgtype:
        :return:
        """
        self.n_count += 1
        if self.n_count >= 500:
            # mark as complete after 1000 executions. Should take a total of 10 seconds to run sequen
            self.is_complete = True

        # return the work to be done on the worker
        return {
            'workload': .01
        }
```

Notice the dependencies variable is provided as a list of task types, each type corresponding to a particular worker. This means we'll need two different workers to handle the task types.

The first worker is defined in examples/dependencies/first_worker.py:

```
import settings
import zmqpipeline
from zmqpipeline.utils import shutdown
import time

class FirstWorker(zmqpipeline.SingleThreadedWorker):
    task_type = zmqpipeline.TaskType(settings.TASK_TYPE_FIRST)
    endpoint = zmqpipeline.EndpointAddress(settings.FIRST_WORKER_ENDPOINT)
    collector_endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ENDPOINT)

    n_executions = 0

    def handle_execution(self, data, *args, **kwargs):
        self.n_executions += 1

        workload = data['workload']
        print 'first worker - working for %f seconds' % workload
```

```
        time.sleep(workload)

        # returning nothing forwards no extra information to the collector

if __name__ == '__main__':
    worker = FirstWorker()
    print 'worker running'
    worker.run()

    print 'finished - handled %d executions' % worker.n_executions
    shutdown.kill_current_process()
```

The second worker is defined in examples/dependencies/second_worker.py:

```
import settings
import zmqpipeline
from zmqpipeline.utils import shutdown
import time

class SecondWorker(zmqpipeline.SingleThreadedWorker):
    task_type = zmqpipeline.TaskType(settings.TASK_TYPE_SECOND)
    endpoint = zmqpipeline.EndpointAddress(settings.SECOND_WORKER_ENDPOINT)
    collector_endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ENDPOINT)

    n_executions = 0

    def handle_execution(self, data, *args, **kwargs):
        self.n_executions += 1

        workload = data['workload']
        print 'second worker - working for %f seconds' % workload
        time.sleep(workload)

        # returning nothing forwards no extra information to the collector

if __name__ == '__main__':
    worker = SecondWorker()
    print 'worker running'
    worker.run()

    print 'finished - handled %d executions' % worker.n_executions
    shutdown.kill_current_process()
```

Running this example you'll see the output of the second worker commence only when the first worker is finished processing all of its tasks.

2.4.5 Example 5: Using metadata

Up to this point the distributor has been statically configured by registering tasks to determine execution behavior. What about altering behavior or providing information dynamically?

For this use case py-zmq-pipeline provides the ability to send metadata to the distributor, which is automatically distributed to tasks and optionally forwarded to workers and collectors.

To use it, two new components are required:

- Subclass and provide implementation for zmqpipeline.worker.MetaDataWorker
- Construct the distributor with additional parameters describing the meta worker

Here's the meta worker from examples/metadata/meta_worker.py:

```
import settings
import zmqpipeline
from zmqpipeline.utils.shutdown import kill_current_process

class MyMetaData(zmqpipeline.MetaDataWorker):
    endpoint = zmqpipeline.EndpointAddress(settings.METADATA_ENDPOINT)

    def get_metadata(self):
        """
        Returns meta data for illustrative purposes
        :return:
        """
        return {
            'meta_variable': 'my value',
        }

if __name__ == '__main__':
    instance = MyMetaData()
    instance.run()

    print 'worker is finished'
    kill_current_process()
```

The meta worker requires an *Endpoint address* instance and an implementation of the get_metadata() method.

get_metadata() takes no arguments and should return parameters or data as a dictionary. Typical use cases for retrieving data in this method include querying a database or introspecting live code.

Next the distributor needs to be told to wait for the meta data to be received before starting. To do this, set a boolean flag and supply the meta data endpoint as follows:

```
import settings
import zmqpipeline
import tasks

if __name__ == '__main__':
    zmqpipeline.Distributor.register_task(tasks.MyTask)

    dist = zmqpipeline.Distributor(
        collector_endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ENDPOINT),
        collector_ack_endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ACK_ENDPOINT),
        receive_metadata = True,
        metadata_endpoint = zmqpipeline.EndpointAddress(settings.METADATA_ENDPOINT)
    )
    dist.run()

    print 'finished'
```

The other components are straightforward. The task will receive the metadata in its data paramter. The following code prints out the value of meta_variable at the task level.

```
import settings
import zmqpipeline

class MyTask(zmqpipeline.Task):
    task_type = zmqpipeline.TaskType(settings.TASK_TYPE_MY_TASK)
    endpoint = zmqpipeline.EndpointAddress(settings.WORKER_ENDPOINT)
    dependencies = []
```

```
    def handle(self, data, address, msgtype):
        """
        Sends one message and prints the contents of the meta variable.
        Meta data is forwarded to the worker.
        :param data: Data received from distributor
        :param address: The address of the client where task output will be sent
        :param msgtype: the type of message received. Typically zmqpipeline.utils.messages.MESSAGE_T
        :return:
        """
        self.is_complete = True

        meta_variable = data['meta_variable']
        print 'MyTask: meta_variable is: ', meta_variable

        # forward data to the worker
        return data
```

This task forwards meta data to the worker. This following worker shows the metadata is then available to the worker, alongside additional data generated by the task.

```
import settings
import zmqpipeline
from zmqpipeline.utils import shutdown

class MyWorker(zmqpipeline.SingleThreadedWorker):
    task_type = zmqpipeline.TaskType(settings.TASK_TYPE_MY_TASK)
    endpoint = zmqpipeline.EndpointAddress(settings.WORKER_ENDPOINT)
    collector_endpoint = zmqpipeline.EndpointAddress(settings.COLLECTOR_ENDPOINT)

    def handle_execution(self, data, *args, **kwargs):
        meta_variable = data['meta_variable']
        print 'MyWorker: meta variable is %s' % meta_variable


if __name__ == '__main__':
    worker = MyWorker()
    print 'worker running'
    worker.run()

    print 'worker finished'
    shutdown.kill_current_process()
```

2.4.6 Example 6: Services

The use case for a distributor / collector pair is offline batch processing. That is, a fixed amount of data that needs to be processed in some period of time and results persisted to a database, warehouse, file, etc.

Services are long-running instances that route client requests to backend workers in load-balanced fashion.

Example code is available in /examples/service, further example documentation is forthcoming.

2.4.7 Example 7: Logging

py-zmq-pipeline makes use of python's built in logging library to output information about what's going on- for example, connections being made, initializations, data messages being passed around, etc.

More about Python's logging libraries can be found here: Python Logging Facility.

Example logging configuration can be found in examples/logging/settings.py.

```
import zmqpipeline
zmqpipeline.configureLogging({
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        # console logger
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': 'simple'
        },
        # a file handler
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': 'output.log',
            'formatter': 'verbose'
        },
    },
    'loggers': {
        'zmqpipeline.distributor': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
        },
        'zmqpipeline.task': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
        },
        'zmqpipeline.collector': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
        },
        'zmqpipeline.worker': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
        },
    },
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d %(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
})
```

2.5 Reference

Reference information for each class and method accessible by developer using this library.

2.5.1 Distribution and Collection

Collector

class zmqpipeline.**Collector**
Collects results from the workers and sends ACKs (acknowledgements) back to the distributor

ack_endpoint
Endpoint for sending ACKs (acknowledgements) back to the distributor. The distributor should connect to this same endpoint for receiving ACKs.

This property must be defined by the sublcassed implementation.

Returns An EndpointAdress instance

endpoint
Endpoint to bind the collector to for receiving messages from workers. Workers should connect to this same endpoint for sending data.

This property must be defined by the subclassed implementation.

Returns An EndpointAddress instance

get_ack_data()
Optionally override this method to attach data to the ACK that will be sent back to the distributor.

This method is typically used when deciding to override Task.is_available_for_handling(), since the data received by the distributor will be forwarded to the Task via is_available_for_handling() for determining whether the task can be executed or not.

The use case of implementing these methods in conjuction is for the task to wait on the receipt of a particular ACK - that is, if the currently processing ACK requires data from a previously processed result. These types of tasks are time and order-dependent.

Returns A dictionary of data

handle_collection(*data*, *task_type*, *msgtype*)
Invoked by the collector when data is received from a worker.

Parameters

- **data** (*dict*) – Data supplied by the worker (a dictionary). If the worker doesn't return anything this will be an empty dict
- **task_type** (*str*) – The task type of the worker and corresponding task
- **msgtype** (*str*) – The message type. Typically zmqpipeline.messages.MESSAGE_TYPE_DATA

Returns None

handle_finished(*data*, *task_type*)
Invoked by the collector when message

Parameters

- **data** (*dict*) – Data received from the worker on a termination signal

- **task_type** (*string*) – The task type of the worker and correspond task

Return dict A dictionary of data to be sent to the distributor along with the ACK, or None to send nothing back to the distributor

run()
Runs the collector. Invoke this method to start the collector

Returns None

Distributor

class zmqpipeline.**Distributor**(*collector_endpoint*, *collector_ack_endpoint*, *receive_metadata=False*, *metadata_endpoint=None*)
Responsible for distributing (pushing) tasks to workers. What gets distributed is determined by Tasks, which are user-implementations that configure how the distributor works.

The pipeline pattern assumes there is only one distributor with one or more registered tasks.

run()
Runs the distributor and blocks. Call this immediately after instantiating the distributor.

Since this method will block the calling program, it should be the last thing the calling code does before exiting. Run() will automatically shutdown the distributor gracefully when finished.

Returns None

shutdown()
Shuts down the distributor. This is automatically called when run() is complete and the distributor exits gracefully. Client code should only invoke this method directly on exiting prematurely, for example on a KeyboardInterruptException

Returns None

Task

class zmqpipeline.**Task**
Tasks define what information the workers receive at each invocation as well as how many items to be processed.

Tasks dynamically configure the Distributor, which looks up and invokes registered tasks.

dependencies
Zero or more tasks, identified by task type, that must be executed in full before this task can begin processing.

Returns A list of TaskType instances

endpoint
A valid EndpointAddress used to push data to worker clients.

Returns An EndpointAddress instance

handle(*data*, *address*, *msgtype*, *ack_data={}*)
Handle invocation by the distributor.

Parameters

- **data** (*dict*) – Meta data, if provided, otherwise an empty dictionary
- **address** (*EndpointAddress*) – The address of the worker data will be sent to.

- **msgtype** (*str*) – The message type received from the worker. Typically zmqpipeline.messages.MESSAGE_TYPE_READY
- **ack_data** (*dict*) – Data received in the most recent ACK from collector

Return dict A dictionary of data to be sent to the worker, or None, in which case the worker will receive no information

initialize(*metadata={}*)
Initializes the task. Default implementation is to store metadata on the object instance :param metadata: Metadata received from the distributor :return:

is_available_for_handling(*last_ack_data*)
Optionally override this if the task requires the ACK of a previously sent task.

If this method is overriden, get_ack_data() should be overrridden in your custom collector. The default is to always assume the task is available to process the next task. This is the fastest behavior, but also doesn't guarantee the last message sent by the task (and then the corresponding worker) has been ACKed by the distributor.

Parameters **last_ack_data** (*dict*) – A dictionary of data received in the last ACK. All instances include _task_type and _id, a globally incrementing counter for the latest ACK. This dictionary will include whatever you attach to get_ack_data() in the collector. If you override this method, you should override get_col

Returns True if the task is available for handling the next received message; false if otherwise

task_type
The type of task being defined. This type must be registered with TaskType before definition, otherwise an exception will be thrown

Returns A TaskType instance

Single-threaded Worker

class zmqpipeline.**SingleThreadedWorker**(**args*, ***kwargs*)
A worker that processes data on a single thread

handle_execution(*data*, **args*, ***kwargs*)
Invoked in the worker's main loop whenever a task is received from the distributor. This is where client implemntations should process data and forward results to the collector.

Parameters

- **data** (*dict*) – Data provided as a dictionary from the distributor
- **args** – A list of additional positional arguments
- **kwargs** – A list of additional keyword arguments

Returns A dictionary of data to be passed to the collector, or None, in which case no data will be forwarded to the collector

Multi-threaded Worker

class zmqpipeline.**MultiThreadedWorker**(**args*, ***kwargs*)
A worker that processes data on multiple threads.

Threads are instantiated and pooled at initialization time to minimize the cost of context switching.

Morever, data is forwarded from the worker to each thread over the inproc protocol by default, which is significantly faster than tcp or ipc.

handle_execution (*data*, **args*, ***kwargs*)
This method is invoked when the worker's main loop is executed. Client implementions of this method should, unlike the SingleThreadedWorker, not process data but instead forward the relevant data to the thread by returning a dictionary of information.

Parameters

- **data** (*dict*) – A dictionary of data received by the worker
- **args** – Additional arguments
- **kwargs** – Additional keyword arguments

Return dict Data to be forwarded to the worker thread

handle_thread_execution (*data*, *index*)
This method is invoked in the working thread. This is where data processing should be handled.

Parameters

- **data** (*dict*) – A dictionary of data provided by the worker
- **index** (*int*) – The index number of the thread that's been invoked

Return dict A dictionary of information to be forwarded to the collector

n_threads
The number of threads used.

Return int A positive integer

Worker

Meta data Worker

class zmqpipeline.**MetaDataWorker**
Transmits meta information to the distributor for dynamic configuration at runtime. When using a meta data worker, the Distributor should be instantiated with receive_metadata boolean turned on.

get_metadata()
Retrieves meta data to be sent to tasks and workers. :return: A dictionary of meta data

run()
Runs the meta worker, sending meta data to the distributor. :return:

2.5.2 Services

Service

class zmqpipeline.**Service** (*frontend_endpoint*, *backend_endpoint*)
A service is a load-balanced router, accepting multiple client requests asynchronously and distributing the work to multiple workers.

run()
Runs the service. This is a blocking call.

Since this method will block the calling program, it should be the last thing the calling code does before exiting. run() will automatically shutdown the service gracefully when finished.

Returns None

shutdown()
Shuts down the service. This is automatically called when run() is complete and the distributor exits gracefully. Client code should only invoke this method directly on exiting prematurely, for example on a KeyboardInterruptException

Returns None

Service Worker

class zmqpipeline.**ServiceWorker**(*id_prefix='worker'*)
A worker that plugs into a service. Regular workers plug into a distributor / collector pair

endpoint
The address of the broker's backend

Return EndpointAddress A valid EndpointAddress instance

handle_message(*data*, *task_type*, *msgtype*)
Overridden by subclassed ServiceWorkers.

Parameters

- **data** – Data the client has submitted for processing, typically in a dictionary
- **task_type** (*TaskType*) – A registered TaskType enumerated value
- **msgtype** (*str*) – A message type enumerated value

Returns Data to be sent back to the requesting client, typically a dictionary

run()
Run the service worker. This call blocks until an END message is received from the broker

Returns None

task_type
A registered task type, or a blank string.

Return TaskType A properly registered task type or an empty string

Service Client

class zmqpipeline.**ServiceClient**(*endpoint_address*, *task_type=''*, *id_prefix='client'*)

2.5.3 Enumerated typed / helpers

EndpointAddress

class zmqpipeline.**EndpointAddress**(*address*)
A valid network address for one of the components of the pipeline pattern to either bind to or connect to.

The supported protocols are:

- tcp for cross-machine communication
- ipc for cross-process communication
- inproc for cross-thread communication

Note that on unix systems cross-process sockets are files, as it recommended to specify the .ipc file extension when naming ipc socket addresses.

TaskType

class zmqpipeline.**TaskType**(*v*)

Represents a task type.

A task type is any valid string that has been previously registered with the TaskType class. Unregistered task types will raise an exception.

2.6 License

2.7 Additional Help

If you're having trouble, please email alan@startupallies.com or tweet @alanilling

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*