pylm Documentation

Release 0.14.4

NFQ Solutions

Jul 26, 2017

Contents

1 Introduction **3**

2 High level API **5**
- 2.1 Servers . . . 5
  - 2.1.1 Cache . . . 7
  - 2.1.2 Scatter messages from the master to the workers . . . 8
  - 2.1.3 Gather messages from the workers . . . 9
- 2.2 The PALM message . . . 10
- 2.3 Server features . . . 11
  - 2.3.1 Errors . . . 11
  - 2.3.2 Logging . . . 12
  - 2.3.3 Playing with the stream of messages . . . 13
- 2.4 Clients . . . 13
- 2.5 Workers . . . 14
- 2.6 The Pipeline component . . . 14
  - 2.6.1 Controlling the messages down the pipeline . . . 15
- 2.7 The Sink component . . . 16
- 2.8 The Hub server . . . 16

3 Low level API **19**
- 3.1 Building components from separate parts . . . 19
  - 3.1.1 The router . . . 19
  - 3.1.2 The parts . . . 20
  - 3.1.3 Services and connections . . . 20
  - 3.1.4 Bypass parts . . . 22
- 3.2 Using HTTP . . . 22
- 3.3 Turning a PALM master into a microservice . . . 23

4 High level API documentation **27**
- 4.1 Servers . . . 27
- 4.2 Clients . . . 34

5 Low level API documentation **37**
- 5.1 The router and the parts . . . 37
- 5.2 The server templates . . . 39
- 5.3 Services . . . 41
- 5.4 Gateways . . . 44
- 5.5 Connections . . . 46

6 Examples **51**
- 6.1 Simple server and client communication . . . 51

6.2 Simple parallel server and client communication . . . 52
6.3 Cache operation for the standalone parallel version . . . 53
6.4 Usage of the scatter function . . . 53
6.5 Usage of the gather function . . . 54
6.6 A pipelined message stream . . . 55
6.7 A pipelined message stream forming a tee . . . 56
6.8 A pipelined message stream forming a tee and controls the stream of messages . . . 58
6.9 A pipelined message stream forming a tee and controls the stream of messages with a sink . . . 60
6.10 Connecting a pipeline to a master . . . 62
6.11 Connecting a hub to a server . . . 63
6.12 Building a master server from its components . . . 65
6.13 Turning a master into a web server with the HTTP gateway . . . 66
6.14 Using server-less infrastructure as workers via the HTTP protocol . . . 67

7 Adapting your components to the pylm registry **69**

8 Beyond Python **73**
8.1 A Simple worker in C++ . . . 73

9 Indices and tables **75**

Python Module Index **77**

Pylm is the Python implementation of PALM, a framework to build clusters of high performance backend components. It is presented in two different levels of abstraction. In the high level API you will find servers and clients that are functional *out of the box*. Use the high level API if you are interested in simple communication patterns like client-server, master-slave or a streaming pipeline. In the low level API there are a variety of small components that, once combined, they can be used to implement almost any kind of component. It's what the high level API uses under the hood. Choose the low level API if you are interested in creating your custom component and your custom communication pattern.

Important: Pylm requires a version of Python equal or higher than 3.4, and it is more thoroughly tested with Python 3.5.

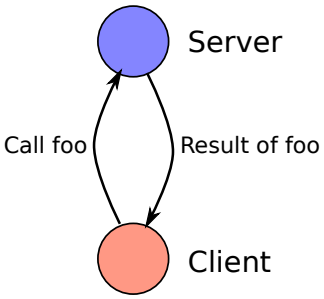
Pylm is a project developed by Guillem Borrell for NFQ Solutions.

CHAPTER 1

Introduction

This is a short introduction of some of the key aspects of pylm, a framework to implement high performance micro-services from reusable components.

But let's begin with something basic, a server and a client able to call one of the server's methods. Much in the fashion of a RPC server.



With pylm, the first step is to create the server by subclassing one of the available templates in the high-level API:

```
from pylm.servers import Server
import logging


class MyServer(Server):
    def foo(self, message):
        self.logger.warning('Got a message')
        return b'you sent me ' + message


if __name__ == '__main__':
    server = MyServer('my_server',
                      db_address='tcp://127.0.0.1:5555',
                      pull_address='tcp://127.0.0.1:5556',
                      pub_address='tcp://127.0.0.1:5557',
                      log_level=logging.DEBUG
                      )
    server.start()
```

Secondly, we create the client that connects to the server and calls the `foo` function from the server.

```
1 from pylm.clients import Client
2
3 client = Client('my_server', 'tcp://127.0.0.1:5555')
4
5 if __name__ == '__main__':
6     result = client.eval('my_server.foo', b'a message')
7     print('Client got: ', result)
```

It does not care in which order you start the client and the server, pylm uses ZeroMQ sockets for all the connections, that deal with all the details. Pylm uses ZeroMQ extensively, and somehow, it also follows its philosophy.

This is what we get when we start the server:

```
$> python server.py
2016-08-09 07:34:53,205 - my_server - WARNING - Got a message
```

And the client:

```
$> python client.py
Client got: b'you sent me a message'
```

Which is what we expected. The function `foo` only picks what is sent from the client, and adds it to `you sent me`. As simple as it seems. However, this basic example allows us to discover some important aspects of pylm.

- All messages are binary, represented as a sequence of bytes. This means that you decide how to serialize and deserialize the data you send to the server. If I decided to send a number instead of a sequence of bytes (replacing line 6 for `result = client.job('foo', 1)`), the client would crash with a *TypeError*.
- The server inherits from *`pylm.servers.Server`*. This parent class includes some interesting capabilities so you don't have to deal with health monitoring, logging, performance analysis and so on. Maybe you don't need all these things with a single server, but they become really handy when you have to monitor hundreds of microservers.
- The philisopy of pylm has two main principles:
  1. Simple things must be simple. If you don't need something, you just ignore it and the whole system will react accordingly.
  2. Pylm is a framework, and does not come with any imposition that is not strictly necessary. Use the deployment system of your choice, the infrastructure you want... Pylm gives you the pieces to create the cluster, and you are in charge of the rest.

Important: At this point you are maybe wondering where to start, and you are afraid that you may have to read tons of documentation to start using pylm. Well, despite we'd love if you carefully read the documentation, it is probable that you find a template that works for you in the *Examples* section. This way you can start from code that it already works.

The example presented in this section does not honor of the capabilities of pylm. Particularly the patterns that support parallel execution of jobs. To learn what are the capabilities of the different servers that are already implemented in pylm, visit the section about the *High level API*.

If you want to understand the underlying principles and algorithms of the small components that are used to implement a palm micro-service, visit the section about the *Low level API*.

CHAPTER 2

High level API

The High level API of pylm exposes a series of servers and clients that you can inherit to implement different communication and execution models. A simple example of a standalone server and its communication with the corresponding client can be found in the *Introduction*.

In each of the flavors, a *single server* refers to a unique server that connects to a client, while a *parallel server* refers to the combination of a *master server* and a series of *workers*. A parallel server is able to distribute the workload among the available workers, in other words, the master is in charge of the management and the workers do the actual work. You will find a thorough description of each flavor and variant in the following sections.

All servers, regardless of their flavor, have a set of useful tools, documented in *Server features*. You can also visit the section devoted to *Workers* if yow want to know the details of those simpler pieces that do the actual work.

Servers

An example of the simplest server was presented in the *Introduction*. A single server running in a single process may be useful, but there are a million alternatives to pylm for that. The real usefulness of pylm arrives when the workload is so large that a single server is not capable of handling it. Here we introduce parallelism for the first time with the parallel standalone server.

A simple picture of the architecture of a parallel server is presented in the next figure. The client connects to a master process that manages an arbitrary number of workers. The workers act as slaves, and connect only to the master.

The following is a simple example on how to configure and run a parallel server. Since the parallel server is designed to handle a large workload, the job method of the client expects a generator that creates a series of binary messages.

```
from pylm.servers import Master


server = Master(name='server',
                pull_address='tcp://127.0.0.1:5555',
                pub_address='tcp://127.0.0.1:5556',
                worker_pull_address='tcp://127.0.0.1:5557',
                worker_push_address='tcp://127.0.0.1:5558',
                db_address='tcp://127.0.0.1:5559')

```

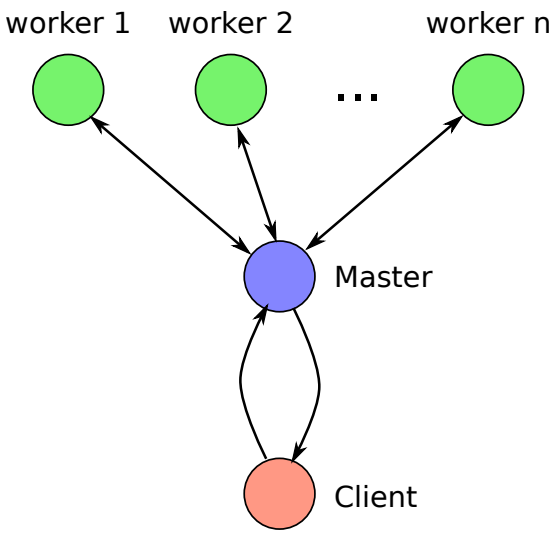


Fig. 2.1: Example of a pair client-master with workers for load balancing

```
if __name__ == '__main__':
    server.start()
```

```
from pylm.servers import Worker
from uuid import uuid4
import sys


class MyWorker(Worker):
    def foo(self, message):
        return self.name.encode('utf-8') + b' processed ' + message

server = MyWorker(str(uuid4()), 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()

```

```
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    for response in client.job('server.foo',
                               repeat(b'a message', 10),
                               messages=10):
        print(response)
```

The master can be run as follows:

```
$> python master.py
```

And we can launch two workers as follows:

```
$> python worker.py worker1
$> python worker.py worker2
```

Finally, here's how to run the client and its output:

```
$> python client.py
b'worker2 processed a message'
b'worker1 processed a message'
b'worker2 processed a message'
b'worker1 processed a message'
b'worker2 processed a message'
b'worker1 processed a message'
b'worker2 processed a message'
b'worker1 processed a message'
b'worker2 processed a message'
b'worker1 processed a message'
```

The communication between the master and the workers is a PUSH-PULL queue of ZeroMQ. This means that the most likely distribution pattern between the master and the workers follows a round-robin scheduling.

Again, this simple example shows very little of the capabilities of this pattern in pylm. We'll introduce features step by step creating a manager with more and more capabilities.

Cache

One of the services that the master offers is a small key-value database that **can be seen by all the workers**. You can use that database with RPC-style using *pylm.clients.Client.set()*, *pylm.clients.Client.get()*, and *pylm.clients.Client.delete()* methods. Like the messages, the data to be stored in the database must be binary.

Note: Note that the calling convention of *pylm.clients.Client.set()* is not that conventional. Remember to pass first the value, and then the key if you want to use your own.

Important: The master stores the data in memory. Have that in mind if you plan to send lots of data to the master.

The following example is a little modification from the previous example. The client, previously to sending the job, it sets a value in the temporary cache of the master server. The workers, where the value of the cached variable is hardcoded within the function that is executed, get the value and they use it to build the response. The variations respect to the previous examples have been empasized.

```
1  from pylm.servers import Master
2
3  server = Master(name='server',
4                  pull_address='tcp://127.0.0.1:5555',
5                  pub_address='tcp://127.0.0.1:5556',
6                  worker_pull_address='tcp://127.0.0.1:5557',
7                  worker_push_address='tcp://127.0.0.1:5558',
8                  db_address='tcp://127.0.0.1:5559')
9
10 if __name__ == '__main__':
11     server.start()
```

```
1  from pylm.servers import Worker
2  import sys
3
4
5  class MyWorker(Worker):
6      def foo(self, message):
7          data = self.get('cached')
8          return self.name.encode('utf-8') + data + message
9
```

```
server = MyWorker(sys.argv[1],
                  db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    client.set(b' cached data ', 'cached')
    print(client.get('cached'))

    for response in client.job('server.foo', repeat(b'a message', 10), messages=10):
        print(response)
```

And the output is the following:

```
$> python client.py
b' cached data '
b'worker1 cached data a message'
b'worker2 cached data a message'
b'worker1 cached data a message'
b'worker2 cached data a message'
b'worker1 cached data a message'
b'worker2 cached data a message'
b'worker1 cached data a message'
b'worker2 cached data a message'
b'worker1 cached data a message'
b'worker2 cached data a message'
```

Scatter messages from the master to the workers

Master server has a useful method called *pylm.servers.Master.scatter()*, that is in fact a generator. For each message that the master gets from the inbound socket, this generator is executed. It is useful to modify the message stream in any conceivable way. In the following example, right at the highlighted lines, a new master server overrides this `scatter` generator with a new one that sends the message it gets three times.

```
from pylm.servers import Master


class MyMaster(Master):
    def scatter(self, message):
        for i in range(3):
            yield message

server = MyMaster(name='server',
                  pull_address='tcp://127.0.0.1:5555',
                  pub_address='tcp://127.0.0.1:5556',
                  worker_pull_address='tcp://127.0.0.1:5557',
                  worker_push_address='tcp://127.0.0.1:5558',
                  db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

The workers are identical than in the previous example. Since each message that the client sends to the master is

repeated three times, the client expects 30 messages instead of 10.

```
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    client.set(b' cached data ', 'cached')
    print(client.get('cached'))

    for response in client.job('server.foo', repeat(b'a message', 10),␣
↪messages=30):
        print(response)
```

This is the (partially omitted) output of the client:

```
$> python client.py
b' cached data '
b'worker1 cached data a message'
b'worker1 cached data a message'
b'worker2 cached data a message'

...

b'worker1 cached data a message'
b'worker2 cached data a message'
b'worker1 cached data a message'
b'worker2 cached data a message'
```

Gather messages from the workers

You can also alter the message stream after the workers have done their job. The master server also includes a *pylm.servers.Master.gather()* method, that is a generator too, that is executed for each message. Being a generator, this means that gather has to yield, and that the result can be either no message, or an arbitrary amount of messages. To make this example a little more interesting, we will also disassemble one of these messages that were apparently just a bunch of bytes.

We will define a gather generator that counts the amount of messages, and when the message number 30 arrives, the final message, its payload is changed with a different binary string. This means that we need to add an attribute to the server for the counter, and we have to modify a message with *pylm.servers.Master.change_payload()*.

See that this example is incremental respect to the previous one, and in consequence it uses the cache service and the scatter and the gather generators.

```
from pylm.servers import Master


class MyMaster(Master):
    def __init__(self, *args, **kwargs):
        self.counter = 0
        super(MyMaster, self).__init__(*args, **kwargs)

    def scatter(self, message):
        for i in range(3):
            yield message

    def gather(self, message):
        self.counter += 1

```

```
        if self.counter == 30:
            yield self.change_payload(message, b'final message')
        else:
            yield message

server = MyMaster(name='server',
                  pull_address='tcp://127.0.0.1:5555',
                  pub_address='tcp://127.0.0.1:5556',
                  worker_pull_address='tcp://127.0.0.1:5557',
                  worker_push_address='tcp://127.0.0.1:5558',
                  db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
$> python client.py
b' cached data '
b'worker1 cached data a message'
b'worker2 cached data a message'
b'worker2 cached data a message'

...

b'worker2 cached data a message'
b'worker1 cached data a message'
b'worker2 cached data a message'
b'worker1 cached data a message'
b'Final message'
```

The PALM message

Every single server and process in PALM, and of course pylm, uses the following Google's protocol buffers message. message.

```
syntax = "proto3";

message PalmMessage {
  string pipeline = 1;
  string client   = 2;
  int64  stage    = 3;
  string function = 4;
  string cache    = 5;
  bytes  payload  = 6;
}
```

If you don't want to dive within the internals of the servers, it is likely that you don't have to even know about it, but it is relevant if you want to understand how servers (and their parts) communicate with each other. As you see, the server has a set of fields, just like a tuple. Each one is used for a different purpose:

pipeline This is an unique identifier of the stream of messages that is sent from the client. It is necessary for the servers to be able to handle multiple streams of messages at the same time.

client An unique ID of the client that initiated the stream.

stage Counter of the step within the stream that the message is going through. Every client initiates this value to 0. Every time that the message goes through a server, this counter is incremented.

function A string with a server.method identifier, or a series of them separated by spaces. These are the functions that have to be called at each step of the pipeline. Of course, this variable needs *stage* to be useful if there are more than one steps to go through.

cache This is an utility variable used for various purposes. When the client communicates with the cache of the servers, this variable brings the key for the key-value store. It is also used internally by some servers to keep track of messages that are being circulated by their internal parts. You can mostly ignore this field, and use it only when you know what you are doing.

payload The actual data carried by the message. It is usually a bunch of bits that you have to deserialize.

Again, if you use the simplest parts of the high-level API, you can probably ignore all of this, but if you want to play with the stream of messages, or you want to play with the internal of the servers, you need to get comfortable with all those fields.

Server features

All servers have built-in features that are useful to build a manageable cluster. This section explains how to use and to configure them. It builds upon the examples of *Servers*.

Errors

You are probably wondering what happens if there is a bug in any of your functions. Of course, your server will not crash. You must try really hard to force one exception in one of the servers and crash it completely. The *user* part of the server runs within an exception handler, that outputs the full exception traceback without side effects.

For instance, take the simplest of the examples, the one in the introduction, and add an obvious bug in the `foo` function.

```
from pylm.servers import Server


class MyServer(Server):
    def foo(self, message):
        self.logger.warning('Got a message')
        print(x)
        return b'you sent me ' + message


if __name__ == '__main__':
    server = MyServer('my_server', 'tcp://127.0.0.1:5555',
                      'tcp://127.0.0.1:5556', 'tcp://127.0.0.1:5557')
    server.start()
```

Of course, this triggers a `NameError`, because the variable `x` was not defined within the user function. The result is that the server logs the error:

```
$> python server.py
2016-08-26 09:41:59,053 - my_server - WARNING - Got a message
2016-08-26 09:41:59,053 - my_server - ERROR - User function gave an error
2016-08-26 09:41:59,054 - my_server - ERROR - Traceback (most recent call last):
Traceback (most recent call last):
  File "/usr/lib/python3.5/site-packages/pylm/servers.py", line 117, in start
    result = user_function(message.payload)
  File "server.py", line 7, in foo
    print(x)
NameError: name 'x' is not defined

...
```

After the error has been logged as such, the server keeps on running and waiting for more input.

Logging

Each server, independently on its variant, has a built-in logger with the usual Python's logging levels. You can find them in the `logging` module of the standard library. The following example, that builds upon the previous one illustrates how to use the logging capabilities.

```
from pylm.servers import Master
import logging


class MyMaster(Master):
    def __init__(self, *args, **kwargs):
        self.counter = 0
        super(MyMaster, self).__init__(*args, **kwargs)

    def scatter(self, message):
        for i in range(3):
            yield message

    def gather(self, message):
        self.counter += 1

        if self.counter == 30:
            self.logger.critical('Changing the payload of the message')
            yield self.change_payload(message, b'final message')
        else:
            yield message

server = MyMaster(name='server',
                  db_address='tcp://127.0.0.1:5559',
                  pull_address='tcp://127.0.0.1:5555',
                  pub_address='tcp://127.0.0.1:5556',
                  worker_pull_address='tcp://127.0.0.1:5557',
                  worker_push_address='tcp://127.0.0.1:5558',
                  log_level=logging.WARNING)

if __name__ == '__main__':
    server.start()
```

The server sets the `WARNING` logging level, and then logs as critical when it changes the payload of the last message.

```
from pylm.servers import Worker
import sys


class MyWorker(Worker):
    def __init__(self, *args, **kwargs):
        self.ncalls = 0
        super(MyWorker, self).__init__(*args, **kwargs)

    def foo(self, message):
        self.ncalls += 1
        data = self.get('cached')

        if self.ncalls%10 == 0:
            self.logger.info('Processed {} messages'.format(self.ncalls))

        return self.name.encode('utf-8') + data + message

server = MyWorker(sys.argv[1], db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
```

```
22      server.start()
```

The worker server implementation just adds a counter, and each time it processes ten messages, it logs as *info* the number of messages processed. The output of the master is then:

```
$> python master.py
2016-08-26 08:08:38,679 - server - CRITICAL - Changing the payload of the message
```

And the output of any of the workers (the two workers are probably doing exactly the same amount of work) is:

```
$> python worker.py worker1
2016-08-26 08:08:38,672 - worker1 - INFO - Processed 10 messages
```

Playing with the stream of messages

Servers have an interesting feature that can be used to seriously tune how the message stream moves through the cluster. In a couple of sections you will learn about pipelines (*The Pipeline component*) and hubs (*The Hub server*), and how several steps can be connected forming a complete streaming pipeline of messages.

Clients

A client has the mission of being the source, and maybe the sink, of the message pipeline. The capabilities of the clients are related on how the stream of messages is handled, so this section is a short review of all of them. You can find the clients in *pylm.clients*, like *pylm.clients.Client*, and they all work in a very similar way.

The simplest configuration is to connect the client to the `db_address` of a *pylm.servers.Server* or a *pylm.servers.Master*, with its corresponding `server_name`. If no other arguments are specified, the client will assume that you want to send all the messages to this server, and also to receive all its output back. There are several examples on how these sockets are managed manually to control the flow of data around the cluster, particularly when there are *pylm.servers.Pipeline* and *pylm.servers.Hub*. You can see some of the cases in the *Examples* section, in which the `sub_address` argument is set as the address of the last server in the pipeline.

Another relevant argument is the `session`, a tag that is added to all the messages in case you need to label them somehow. It is relevant when using the cache of the servers, and it may be useful in some applications.

Clients connect to the key-value store of the server they send the message stream. This feature can be used to implement all sorts of algorithms, and it comes handy when one has to communicate the client with the workers with more information that the one that is sent through the messages. A client has therefore the usual functions for the management of key-value stores: *pylm.clients.Client.get()*, *pylm.clients.Client.set()* and *pylm.clients.Client.delete()*.

Note: The set function reverses the argument order from the usual. The first argument is the value and the second is the key. The reason for that is that you can set a value without the key, and the client generates a random key for you.

The two methods that are used to start the execution of a pipeline are *pylm.clients.Client.eval()* and *pylm.clients.Client.job()*. The former sends only one message, while the latter loops over a generator that produces the message stream. In any case, the message is of the type `bytes`.

The first argument of these two methods is the function to be called in the server or the succession of servers forming a pipeline. The argument can be therefore a string or a list of string, always formatted as two words: the name of the server and the function, separated by a dot. For instance, if we have a server called *first* connected to a pipeline called *second*, and we want to call the *firstfunction* of the former, and the *secondfunction* of the latter, the first argument will be the following list:

```
['first.firstfunction', 'second.secondfunction']
```

The second argument is the palyoad described previously, while the third argument `messages` refers to the number of messages the client has to receive before it exits. If this value is not set, it just stays alive forever waiting for a practically inifinite number of messages.

The last argument `cache` sets the *cache* field of the message, and it is intended for advanced uses.

Workers

Workers are the simplest, but not least important, piece of the parallel servers in pylm. They are in charge of doing the actual work, since the master server deals with worker management, storing data, and other smaller details.

A worker subclasses *`pylm.servers.Worker`*, and defines a set of additional methods that are exposed to the cluster. To configure the worker, only the `db_address` parameter is needed, since the other sockets that connect to the master are usually configured automatically. However, those arguments are present in case you really need to set them manually.

Another feature of the workers is that they are connected to the key-value store of the Master or the Hub, along with the client. This means that the key-value store can be used to communicate each worker with the client and with the other workers too. The methods to interact with the key-value store are the same as the client's.

The Pipeline component

The serial server presented in the previous section is intended to receive messages from a client. The pipeline server is just like a serial server, but it is designed to receive a stream of messages from another server, forming a pipeline. It can then redirect the messages to another server or it can route the messages back to the client to close the message loop.

The simplest architecture where a Pipeline server is useful is adding an additional step to a client-server call like the one presented in the following figure.

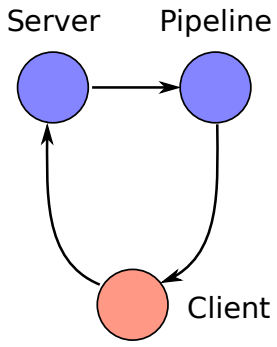


Fig. 2.2: Diagram of components for a simple use of the pipeline server

You can see an example how the Pipeline server to create a pipeline started by a Server in the examples section (*A pipelined message stream*). The big picture is simple. A Server starts a pipeline, and the pipeline servers are the steps of it.

One important detail in this first example is that the client gets a sequence of method calls, the server name and the method of each step, in a list. This of course means that the first argument of the *`pylm.clients.Client.eval()`* and *`pylm.clients.Client.job()`* methods in may be either a string or a list of strings.

Pipeline servers can be attached to Master servers too to attach a parallel-processing step to a serial-processing step

You can find the full example in *Connecting a pipeline to a master*

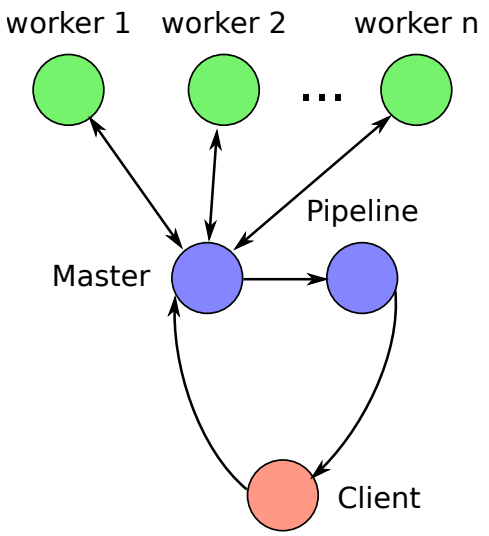


Fig. 2.3: Sketch of a pipeline server processing the output of a master.

Controlling the messages down the pipeline

One important feature of the pipelined message stream is that it can be controlled and diverted. If one connects multiple pipeline servers to a single server, the default behavior is to send all messages to all the connected pipelines.

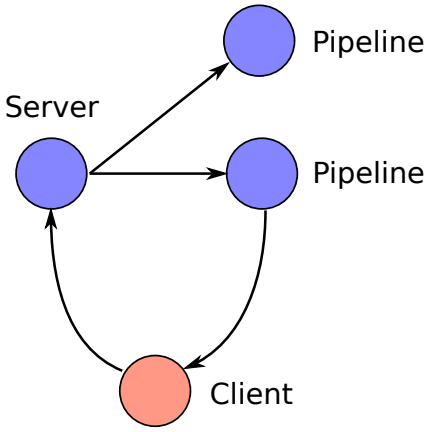


Fig. 2.4: Example of two pipeline components fetching the output of a server. The default behavior of the que is to send the same data to both pipelines.

If you take a look at the full example (*A pipelined message stream forming a tee*), you can see that the Pipeline needs an extra argument, which is the name of the server or the pipeline at the previous step. At the same time, one must tell the servers at its creation that the stream of messages will be sent to a Pipeline, and not sent back to the client.

If you want a finer-grain control over where each message is sent down the pipeline you can use the handle_stream method to manage the stream. This can be used in combination with the `previous` option to fully manage the routing of the messages on each step.

You can see the full example here (*A pipelined message stream forming a tee and controls the stream of messages*).

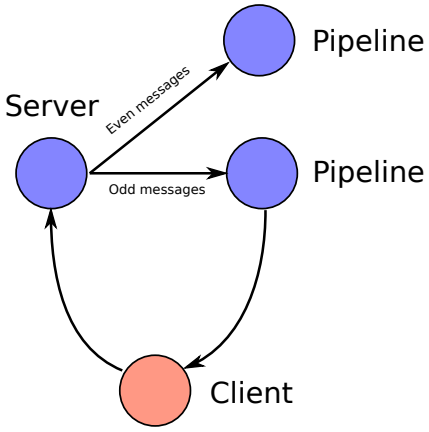


Fig. 2.5: The flow of messages from the server to the pipeline can be controlled in many different ways. In this example, the odd messages are sent to one component, while the even are sent to a different one.

The Sink component

We have seen how to fan-out the stream of messages with *The Pipeline component*. The next step is to learn how to fan-in a series of streams and join the output. This can be done via the `pylm.servers.Sink` server.

A Sink server can subscribe to one or many components of type `pylm.servers.Server` or `pylm.servers.Pipeline`, and fetch all the message every previous step releases. The configuration is similar to a Pipeline component, only the `sub_addresses` and the `previous` parameters require further comment. Since the component must connect to multiple components upstream, these parameters are of type `list`, `sub_addresses` are the list of addresses the component has to connect to, and `previous` are the topics for subscription. The values of these two parameters are zipped, so the order of the elements matter.

You can see a complete example of the use of a `pylm.servers.Sink` in *A pipelined message stream forming a tee and controls the stream of messages with a sink*.

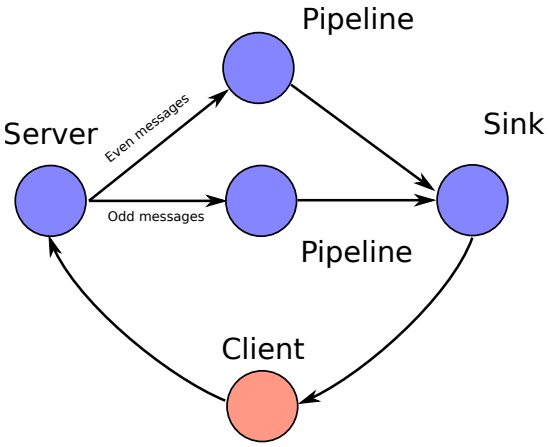


Fig. 2.6: In this sketch, the sink is attached to two pipeline servers that process a divided stream of messages. One of the possible uses of sink components is to synchronize the stream of messages or to check for completion.

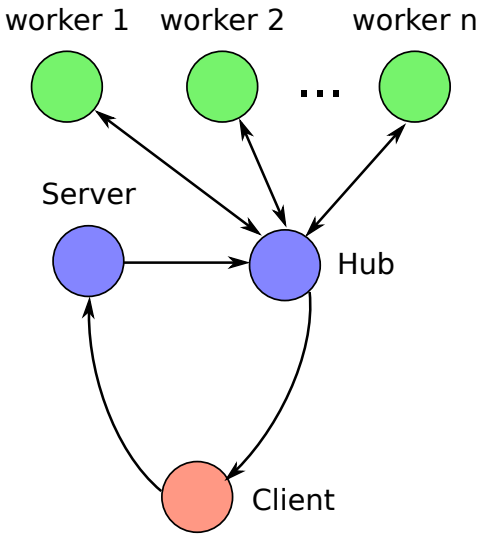
The Hub server

The hub server is a master that can be connected like a pipeline. It therefore needs some more information to be added to a cluster. Instead of pulling from clients, it subscribes to a stream of messages coming from a master or

a server. This is the reason why you have a sub connection instead of a pull service, and you have to take into account when configuring it.

There is yet another change respect to a master server, the *previous* parameter. If you don't want to play dirty tricks to the message stream, i.e. routing a message to a particular subscribed server, it's just the name of the previous server the hub is subscribed to.

Maybe the simplest stream of messages involving a hub is the following.



You can see an example how the output of a server can be pipelined to a hub in the example *Connecting a hub to a server*.

CHAPTER 3

Low level API

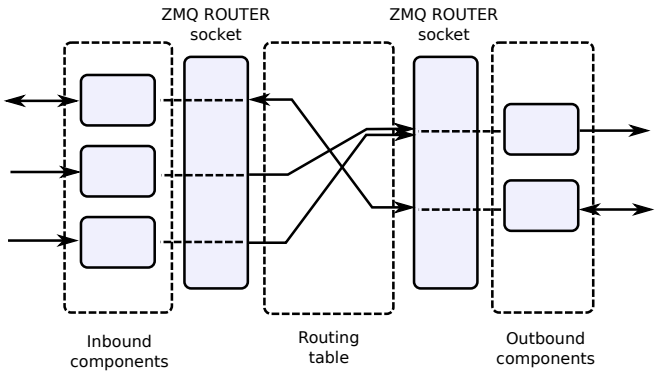
If you are proficient in distributed computing, some of the key aspects of pylm may sound like the actor model. We are aware of this similarity, but we would rather use the term component, because pylm does not expose a programming paradigm. It's just a framework with pieces to implement distributed systems.

Some concepts in this section may be hard, particularly if you don't know how message queues work, ZeroMQ in particular. Before reading this section, it may be a good idea to read the ZeroMQ Guide.

Building components from separate parts

The router

At the very core of most Pylm servers, there is a router, and its architecture is the only profound idea in the whole pylm architecture. The goal is to manage the communication between the servers in a way as much similar to an actual network architecture as possible.



The mission of this router sockets is to connect the parts that receive inbound messages with the parts that deal with outbound messages. The two tall blocks at each side of the table is a representation with such connection. If you know how an actual router works, a part would be a NIC, while the ROUTER socket and the routing table would be the switch. The router is documented in `pylm.parts.core.Router`.

The parts are also related to the router by the fact that they are all threads that run within the same process. In consequence, a pylm server could be described as a router and a series of parts that run in the same process.

The parts

There is a decent number of parts, each one covering some functionality within the PALM ecosystem. What follows is a classification of the several parts that are already available according to their characteristics.

First of all, parts can be services or connections. A service is a part that *binds* to a socket, which is an important detail when you design a cluster. A bind socket blocks waiting for a connection from a different thread or process. Therefore, a service is used to define the communication endpoint. All the available services are present in the module `pylm.parts.services`.

Connections are the complementary of servers, they are used in the *client* side of the communication, and are present in the module `pylm.parts.connections`.

On the second hand, parts can be standard or *bypass*. The former connects to the router, while the latter ignores the router completely. Bypass components inherit from `pylm.parts.core.BypassInbound` or from `pylm.parts.core.BypassOutbound` and also use the word *bypass* in its name, while standard components that connect to the router inherit from `pylm.parts.core.Inbound` and `pylm.parts.core.Outbound`. As an example, the part `pylm.parts.services.CacheService`, regardless of not being named as a bypass name, it exposes the internal cache of a server to workers and clients and does no communicate to the router in any case.

On the third hand, and related to the previous classification, parts can be inbound or outbound according to the direction of the *first* message respect to the router. Inbound services and components inherit from `pylm.parts.core.Inbound` and `pylm.parts.core.BypassInbound`, while outbound inherit from `pylm.parts.core.Outbound` and `pylm.parts.core.BypassOutbound`.

On the fourth hand, components may block or not depending on whether they expect the pair to send some message back. This behavior depends on the kind of ZeroMQ socket in use.

> **Warning:** There is such a thing as a blocking outbound service. This means that the whole server is expecting some pair of an outbound service to send a message back. As you can imagine, these kind of parts must be handled with extreme care.

This classification may seem a little confusing, so we will offer plenty of examples covering most of the services and connections avialiable at the present version.

Services and connections

It's time to build a component from a router and some services and parts that are already available. This way you will have a rough idea of how the high level API of pylm is built. Some of the details of the implementation are not described yet, but this example is a nice prologue about the things you need to know to master the low level API.

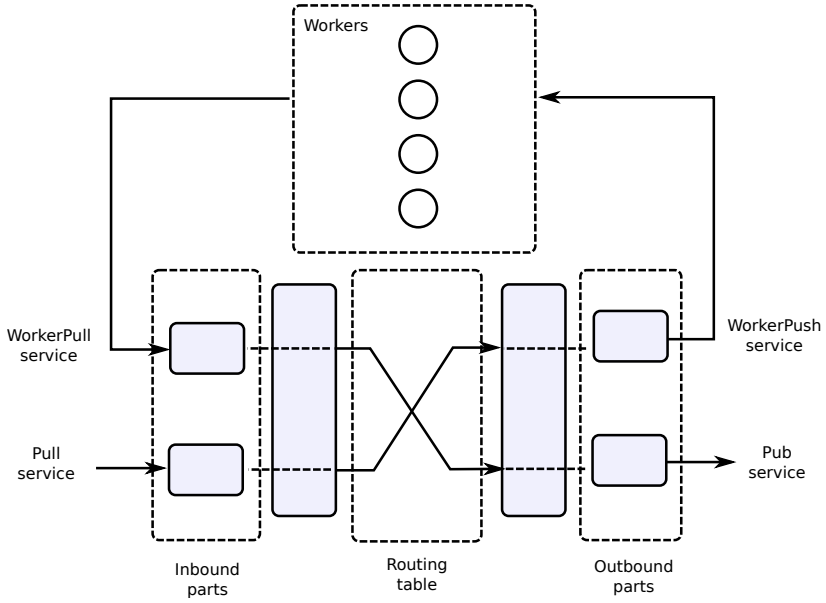
In this section, we have seen that the router is a crucial part of any server in pylm. The helper class `pylm.parts.servers.ServerTemplate` is designed to easily attach the parts to a router. The internal design of a master server can be seen in the following sketch.

A master server like the one used in the examples needs the router and four service parts.

- A *Pull* part that receives the messages from the client
- A *Push* part that sends the messages to the workers
- A *Pull* part that gets the result from the workers
- A *Pub* part that sends the results down the message pipeline or back to the client.

All parts are non-blocking, and the message stream is never interrupted. All the parts are *services*, meaning that the workers and the client connect to the respective sockets, since *service parts* bind to its respective outwards-facing socket.

The part library has a part for each one of the needs depicted above. There is a `pylm.parts.services.PullService` that binds a ZeroMQ Pull socket to the exterior, and sends the messages to the router.



There is a *pylm.parts.services.PubService* that works exactly the other way around. It listens to the router, and forwards the messages to a ZeroMQ Push socket. There are also specific services to connect to worker servers, *pylm.parts.services.WorkerPullService* and *pylm.parts.services.WorkerPushService*, that are very similar to the two previously described services. With those pieces, we are ready to build a master server as follows

```
from pylm.parts.servers import ServerTemplate
from pylm.parts.services import PullService, PubService, WorkerPullService, WorkerPushService, \
    CacheService

server = ServerTemplate()

db_address = 'tcp://127.0.0.1:5559'
pull_address = 'tcp://127.0.0.1:5555'
pub_address = 'tcp://127.0.0.1:5556'
worker_pull_address = 'tcp://127.0.0.1:5557'
worker_push_address = 'tcp://127.0.0.1:5558'

server.register_inbound(PullService, 'Pull', pull_address, route='WorkerPush')
server.register_inbound(WorkerPullService, 'WorkerPull', worker_pull_address, route='Pub')
server.register_outbound(WorkerPushService, 'WorkerPush', worker_push_address)
server.register_outbound(PubService, 'Pub', pub_address)
server.register_bypass(CacheService, 'Cache', db_address)
server.preset_cache(name='server',
                    db_address=db_address,
                    pull_address=pull_address,
                    pub_address=pub_address,
                    worker_pull_address=worker_pull_address,
                    worker_push_address=worker_push_address)

if __name__ == '__main__':
    server.start()
```

Note: There is an additional type of service called *bypass* in this implementation, that will be described at the end of this section.

This server is functionally identical to the master server used in the first example of the section describing *Servers*. You can test it using the same client and workers.

Bypass parts

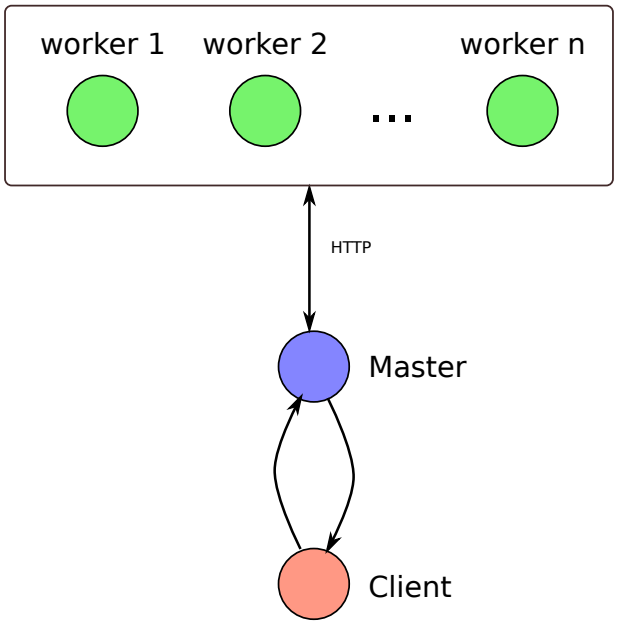
In the previous example one `pylm.parts.services.CacheService` was registered as a *bypass* part. These kind of parts also run in the same process as the router in a separate thread, but they do not interact with the router at all. The *CacheService* is a good example of that. It is the key-value store of the Master and Hub server, and it is one of those nice goodies of the high-level API. It has to be there, but it never waits for a message coming from the router.

Another part that is registered as bypass is the `pylm.parts.gateways.HttpGateway`.

Using HTTP

The default transport to connect the different parts of PALM is ZeroMQ over tcp. Some organizations may not find that suitable for all cases. For instance, it may be necessary to secure servers with encrypted connections, or some servers may have to run behind traffic-sniffing firewalls, or you want to exploit a server-less architecture for the workers... You name it.

For this reason, pylm includes two parts to communicate with workers with the HTTP protocol to create a pipeline with that combines ZMQ sockets over TCP and HTTP.




```
from pylm.parts.servers import ServerTemplate
from pylm.parts.services import PullService, PubService, CacheService
from pylm.parts.connections import HttpConnection

server = ServerTemplate()

db_address = 'tcp://127.0.0.1:5559'
pull_address = 'tcp://127.0.0.1:5555'
pub_address = 'tcp://127.0.0.1:5556'

server.register_inbound(PullService, 'Pull', pull_address,
                        route='HttpConnection')
server.register_outbound(HttpConnection, 'HttpConnection',
```

```
                          'http://localhost:8888', route='Pub', max_workers=1)
server.register_outbound(PubService, 'Pub', pub_address)
server.register_bypass(CacheService, 'Cache', db_address)
server.preset_cache(name='server',
                    db_address=db_address,
                    pull_address=pull_address,
                    pub_address=pub_address)

if __name__ == '__main__':
    server.start()
```

Pylm provides a way to implement a worker in a very similar fashion to the previous workers that were shown, and to obtain a WSGI application from it. If you are not familiar with WSGI, it is a standarised way in which Python applications are able to talk with web servers.

```
from pylm.remote.server import RequestHandler, DebugServer, WSGIApplication


class MyHandler(RequestHandler):
    def foo(self, payload):
        return payload + b' processed online'

app = WSGIApplication(MyHandler)

if __name__ == '__main__':
    server = DebugServer('localhost', 8888, MyHandler)
    server.serve_forever()
```

```
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    for response in client.job('server.foo', repeat(b'a message', 10),␣
↪messages=10):
        print(response)
```

Since the worker is now a WSGI application, you can run it with the web server of your choice.

```
$> gunicorn -w 4 -b 127.0.0.1:8888 web_worker:app
```

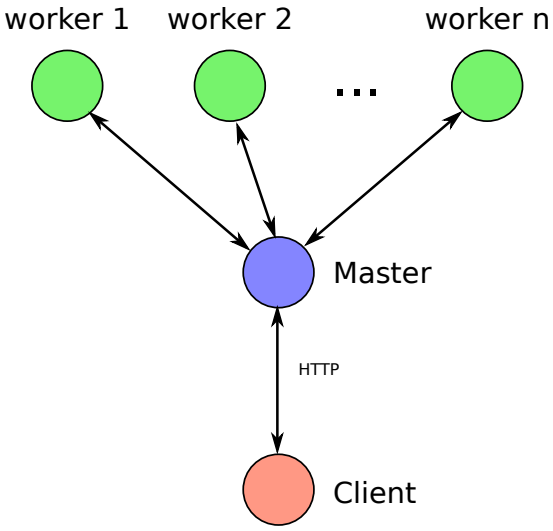
Turning a PALM master into a microservice

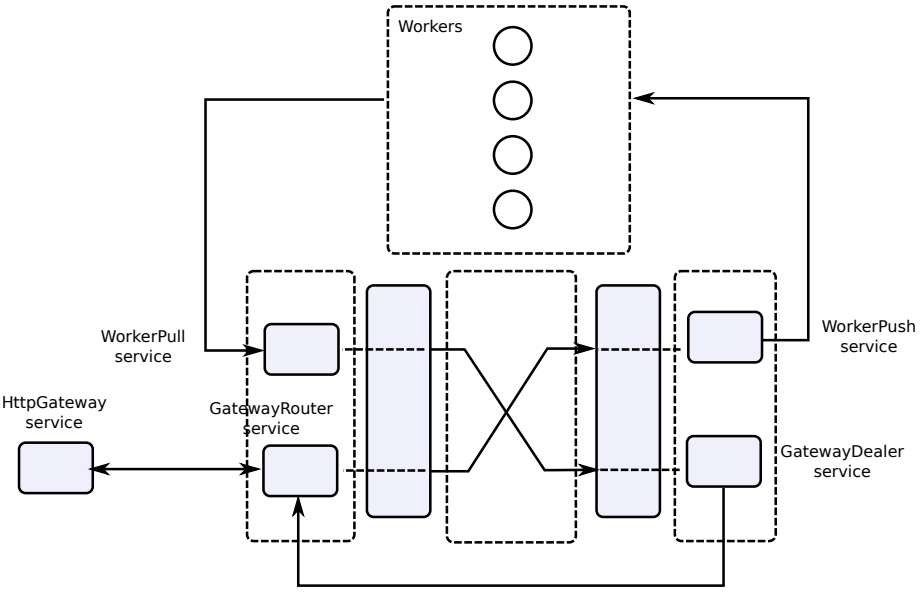
The low level API also includes parts that can be used to turn a master server into a more classical microserver in the form of an HTTP server. The goal would be to offer a gateway to a PALM cluster with the HTTP protocol, The master (and now microservice too) can connect to as many workers as it is needed, just like a Master or a Hub, while serving to several HTTP clients.

Note: One caveat. The HttpGateway part spawns a thread for every client connection so don't rely on it for dealing with thousands of concurrent connections.

The components are the *pylm.parts.gateways.GatewayRouter*, *pylm.parts.gateways.GatewayDealer* and *pylm.parts.gateways.HttpGateway*. They can be used in the following fashion to wire a master to listen to an HTTP connection, that is served from the HttpGateway part.

The whole example can be implemented as follows.


worker 1
worker 2
...
worker n
Master
HTTP
Client


Workers
WorkerPull
service
HttpGateway
service
GatewayRouter
service
WorkerPush
service
GatewayDealer
service

```
from pylm.parts.servers import ServerTemplate
from pylm.parts.services import WorkerPullService, WorkerPushService, \
    CacheService
from pylm.parts.gateways import GatewayDealer, GatewayRouter, HttpGateway

server = ServerTemplate()

worker_pull_address = 'tcp://127.0.0.1:5557'
worker_push_address = 'tcp://127.0.0.1:5558'
db_address = 'tcp://127.0.0.1:5559'

server.register_inbound(GatewayRouter,
                        'gateway_router',
                        'inproc://gateway_router',
                        route='WorkerPush')
server.register_outbound(GatewayDealer,
                         'gateway_dealer',
                         listen_address='inproc://gateway_router')
server.register_bypass(HttpGateway,
                       name='HttpGateway',
                       listen_address='inproc://gateway_router',
                       hostname='localhost',
                       port=8888)
server.register_inbound(WorkerPullService, 'WorkerPull', worker_pull_address,
                        route='gateway_dealer')
server.register_outbound(WorkerPushService, 'WorkerPush', worker_push_address)
server.register_bypass(CacheService, 'Cache', db_address)

server.preset_cache(name='server',
                    db_address=db_address,
                    worker_pull_address=worker_pull_address,
                    worker_push_address=worker_push_address)

if __name__ == '__main__':
    server.start()
```

```
import sys

from pylm.servers import Worker


class MyWorker(Worker):
    def function(self, message):
        return b'acknowledged'

server = MyWorker(sys.argv[1], db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
import requests

print(requests.get('http://localhost:8888/function').content)
```

Note that the client is calling the path `/function` of the server, that is mapped to the `function` method of the worker. This means that the body of the HTTP message is precisely the message you wan to send down the pipeline.

In this example, the GatewayDealer only pipes the output of the workers back to the GatewayRouter and the HttpGateway, but remember that every outbound component has a `route` argument that allows you to multiplex the output stream.

CHAPTER 4

High level API documentation

Servers

class pylm.servers.**Hub**(*name: str*, *sub_address: str*, *pub_address: str*, *worker_pull_address: str*, *worker_push_address: str*, *db_address: str*, *previous: str*, *pipelined: bool = False*, *cache: object = <pylm.persistence.kv.DictDB object>*, *log_level: int = 20*)

A Hub is a pipelined Master.

Parameters

- **name** – Name of the server
- **sub_address** – Valid address for the sub service
- **pub_address** – Valid address for the pub service
- **worker_pull_address** – Valid address for the pull-from-workers service
- **worker_push_address** – Valid address for the push-to-workers service
- **db_address** – Valid address to bind the Cache service
- **previous** – Name of the previous server to subscribe to the queue.
- **pipelined** – The stream is pipelined to another server.
- **cache** – Key-value embeddable database. Pick from one of the supported ones
- **log_level** – Logging level

change_payload(*message: messages_pb2.PalmMessage*, *new_payload: bytes*) → messages_pb2.PalmMessage

Change the payload of the message

Parameters

- **message** – The binary message to be processed
- **new_payload** – The new binary payload

Returns Serialized message with the new payload

gather(*message: messages_pb2.PalmMessage*)

Gather function for outbound messages

Parameters **message** – Binary message

Returns Yield none, one or multiple binary messages

handle_stream(*message: messages_pb2.PalmMessage*)
Handle the stream of messages.

Parameters **message** – The message about to be sent to the next step in the cluster

Returns topic (str) and message (PalmMessage)

The default behaviour is the following. If you leave this function unchanged and pipeline is set to False, the topic is the ID of the client, which makes the message return to the client. If the pipeline parameter is set to True, the topic is set as the name of the server and the step of the message is incremented by one.

You can alter this default behaviour by overriding this function. Take into account that the message is also available in this function, and you can change other parameters like the stage or the function.

preset_cache(***kwargs*)
Send the following keyword arguments as cache variables. Useful for configuration variables that the workers or the clients fetch straight from the cache.

Parameters **kwargs** –

register_bypass(*part*, *name=''*, *listen_address=''*, ***kwargs*)
Register a bypass part to this server

Parameters

- **part** – part class
- **name** – part name
- **listen_address** – Valid ZeroMQ address listening to the exterior
- **kwargs** – Additional keyword arguments to pass to the part

register_inbound(*part*, *name=''*, *listen_address=''*, *route=''*, *block=False*, *log=''*, ***kwargs*)
Register inbound part to this server.

Parameters

- **part** – part class
- **name** – Name of the part
- **listen_address** – Valid ZeroMQ address listening to the exterior
- **route** – Outbound part it routes to
- **block** – True if the part blocks waiting for a response
- **log** – Log message in DEBUG level for each message processed.
- **kwargs** – Additional keyword arguments to pass to the part

register_outbound(*part*, *name=''*, *listen_address=''*, *route=''*, *log=''*, ***kwargs*)
Register outbound part to this server

Parameters

- **part** – part class
- **name** – Name of the part
- **listen_address** – Valid ZeroMQ address listening to the exterior
- **route** – Outbound part it routes the response (if there is) to
- **log** – Log message in DEBUG level for each message processed
- **kwargs** – Additional keyword arguments to pass to the part

scatter(*message: messages_pb2.PalmMessage*)
Scatter function for inbound messages

Parameters **message** – Binary message

Returns Yield none, one or multiple binary messages

start()
Start the server with all its parts.

class pylm.servers.**Master**(*name: str*, *pull_address: str*, *pub_address: str*, *worker_pull_address: str*, *worker_push_address: str*, *db_address: str*, *pipelined: bool = False*, *cache: object = <pylm.persistence.kv.DictDB object>*, *log_level: int = 20*)
Standalone master server, intended to send workload to workers.

Parameters

- **name** – Name of the server
- **pull_address** – Valid address for the pull service
- **pub_address** – Valid address for the pub service
- **worker_pull_address** – Valid address for the pull-from-workers service
- **worker_push_address** – Valid address for the push-to-workers service
- **db_address** – Valid address to bind the Cache service
- **pipelined** – The output connects to a Pipeline or a Hub.
- **cache** – Key-value embeddable database. Pick from one of the supported ones
- **log_level** – Logging level

change_payload(*message: messages_pb2.PalmMessage*, *new_payload: bytes*) → messages_pb2.PalmMessage
Change the payload of the message

Parameters

- **message** – The binary message to be processed
- **new_payload** – The new binary payload

Returns Serialized message with the new payload

gather(*message: messages_pb2.PalmMessage*)
Gather function for outbound messages

Parameters **message** – Binary message

Returns Yield none, one or multiple binary messages

handle_stream(*message: messages_pb2.PalmMessage*)
Handle the stream of messages.

Parameters **message** – The message about to be sent to the next step in the cluster

Returns topic (str) and message (PalmMessage)

The default behaviour is the following. If you leave this function unchanged and pipeline is set to False, the topic is the ID of the client, which makes the message return to the client. If the pipeline parameter is set to True, the topic is set as the name of the server and the step of the message is incremented by one.

You can alter this default behaviour by overriding this function. Take into account that the message is also available in this function, and you can change other parameters like the stage or the function.

preset_cache(***kwargs*)
Send the following keyword arguments as cache variables. Useful for configuration variables that the workers or the clients fetch straight from the cache.

Parameters **kwargs** –

register_bypass(*part*, *name=''*, *listen_address=''*, ***kwargs*)
Register a bypass part to this server

Parameters

- **part** – part class
- **name** – part name
- **listen_address** – Valid ZeroMQ address listening to the exterior
- **kwargs** – Additional keyword arguments to pass to the part

register_inbound(*part*, *name=''*, *listen_address=''*, *route=''*, *block=False*, *log=''*, ***kwargs*)
Register inbound part to this server.

Parameters

- **part** – part class
- **name** – Name of the part
- **listen_address** – Valid ZeroMQ address listening to the exterior
- **route** – Outbound part it routes to
- **block** – True if the part blocks waiting for a response
- **log** – Log message in DEBUG level for each message processed.
- **kwargs** – Additional keyword arguments to pass to the part

register_outbound(*part*, *name=''*, *listen_address=''*, *route=''*, *log=''*, ***kwargs*)
Register outbound part to this server

Parameters

- **part** – part class
- **name** – Name of the part
- **listen_address** – Valid ZeroMQ address listening to the exterior
- **route** – Outbound part it routes the response (if there is) to
- **log** – Log message in DEBUG level for each message processed
- **kwargs** – Additional keyword arguments to pass to the part

scatter(*message: messages_pb2.PalmMessage*)
Scatter function for inbound messages

Parameters **message** – Binary message

Returns Yield none, one or multiple binary messages

start()
Start the server with all its parts.

class pylm.servers.**MuxWorker**(*name=''*, *db_address=''*, *push_address=None*, *pull_address=None*, *log_level=20*, *messages=9223372036854775807*)
Standalone worker for the standalone master which allow that user function returns an iterator a therefore the gather function of the Master recieve more messages.

Parameters

- **name** – Name assigned to this worker server
- **db_address** – Address of the db service of the master
- **push_address** – Address the workers push to. If left blank, fetches it from the master
- **pull_address** – Address the workers pull from. If left blank, fetches it from the master
- **log_level** – Log level for this server.
- **messages** – Number of messages before it is shut down.

delete(*key*)
Deletes data in the server's internal cache.

Parameters **key** – Key of the data to be deleted

Returns

get(*key*)
Gets a value from server's internal cache

Parameters **key** – Key for the data to be selected.

Returns

set(*value*, *key=None*)
Sets a key value pare in the remote database.

Parameters

- **key** –
- **value** –

Returns

start()
Starts the server

class pylm.servers.**Pipeline**(*name*, *db_address*, *sub_address*, *pub_address*, *previous*, *to_client=True*, *log_level=20*, *messages=9223372036854775807*)
Minimal server that acts as a pipeline.

Parameters

- **name** (*str*) – Name of the server
- **db_address** (*str*) – ZeroMQ address of the cache service.
- **sub_address** (*str*) – Address of the pub socket of the previous server
- **pub_address** (*str*) – Address of the pub socket
- **previous** – Name of the previous server.
- **to_client** – True if the message is sent back to the client. Defaults to True
- **log_level** – Minimum output log level.
- **messages** (*int*) – Total number of messages that the server processes. Useful for debugging.

echo(*payload*)
Echo utility function that returns the unchanged payload. This function is useful when the server is there as just to modify the stream of messages.

Returns payload (bytes)

handle_stream(*message*)
Handle the stream of messages.

Parameters **message** – The message about to be sent to the next step in the cluster

Returns topic (str) and message (PalmMessage)

The default behaviour is the following. If you leave this function unchanged and pipeline is set to False, the topic is the ID of the client, which makes the message return to the client. If the pipeline parameter is set to True, the topic is set as the name of the server and the step of the message is incremented by one.

You can alter this default behaviour by overriding this function. Take into account that the message is also available in this function, and you can change other parameters like the stage or the function.

start(*cache_messages=9223372036854775807*)
Start the server

Parameters **cache_messages** – Number of messages the cache service handles before it shuts down. Useful for debugging

class pylm.servers.**Server**(*name*, *db_address*, *pull_address*, *pub_address*, *pipelined=False*, *log_level=20*, *messages=9223372036854775807*)
Standalone and minimal server that replies single requests.

Parameters

- **name** (*str*) – Name of the server
- **db_address** (*str*) – ZeroMQ address of the cache service.
- **pull_address** (*str*) – Address of the pull socket
- **pub_address** (*str*) – Address of the pub socket
- **pipelined** – True if the server is chained to another server.
- **log_level** – Minimum output log level.
- **messages** (*int*) – Total number of messages that the server processes. Useful for debugging.

echo(*payload*)
Echo utility function that returns the unchanged payload. This function is useful when the server is there as just to modify the stream of messages.

Returns payload (bytes)

handle_stream(*message*)
Handle the stream of messages.

Parameters **message** – The message about to be sent to the next step in the cluster

Returns topic (str) and message (PalmMessage)

The default behaviour is the following. If you leave this function unchanged and pipeline is set to False, the topic is the ID of the client, which makes the message return to the client. If the pipeline parameter is set to True, the topic is set as the name of the server and the step of the message is incremented by one.

You can alter this default behaviour by overriding this function. Take into account that the message is also available in this function, and you can change other parameters like the stage or the function.

start(*cache_messages=9223372036854775807*)
Start the server

Parameters **cache_messages** – Number of messages the cache service handles before it shuts down. Useful for debugging

class pylm.servers.**Sink**(*name*, *db_address*, *sub_addresses*, *pub_address*, *previous*, *to_client=True*, *log_level=20*, *messages=9223372036854775807*)
Minimal server that acts as a sink of multiple streams.

Parameters

- **name** (*str*) – Name of the server

- **db_address** (*str*) – ZeroMQ address of the cache service.
- **sub_addresses** (*str*) – List of addresses of the pub socket of the previous servers
- **pub_address** (*str*) – Address of the pub socket
- **previous** – List of names of the previous servers.
- **to_client** – True if the message is sent back to the client. Defaults to True
- **log_level** – Minimum output log level. Defaults to INFO
- **messages** (*int*) – Total number of messages that the server processes. Defaults to Infty Useful for debugging.

echo(*payload*)
Echo utility function that returns the unchanged payload. This function is useful when the server is there as just to modify the stream of messages.

Returns payload (bytes)

handle_stream(*message*)
Handle the stream of messages.

Parameters **message** – The message about to be sent to the next step in the cluster

Returns topic (str) and message (PalmMessage)

The default behaviour is the following. If you leave this function unchanged and pipeline is set to False, the topic is the ID of the client, which makes the message return to the client. If the pipeline parameter is set to True, the topic is set as the name of the server and the step of the message is incremented by one.

You can alter this default behaviour by overriding this function. Take into account that the message is also available in this function, and you can change other parameters like the stage or the function.

start(*cache_messages=9223372036854775807*)
Start the server

Parameters **cache_messages** – Number of messages the cache service handles before it shuts down. Useful for debugging

class pylm.servers.**Worker**(*name=''*, *db_address=''*, *push_address=None*, *pull_address=None*, *log_level=20*, *messages=9223372036854775807*)
Standalone worker for the standalone master.

Parameters

- **name** – Name assigned to this worker server
- **db_address** – Address of the db service of the master
- **push_address** – Address the workers push to. If left blank, fetches it from the master
- **pull_address** – Address the workers pull from. If left blank, fetches it from the master
- **log_level** – Log level for this server.
- **messages** – Number of messages before it is shut down.

delete(*key*)
Deletes data in the server's internal cache.

Parameters **key** – Key of the data to be deleted

Returns

get(*key*)
Gets a value from server's internal cache

Parameters **key** – Key for the data to be selected.

Returns

set (*value*, *key=None*)
Sets a key value pare in the remote database.

Parameters

- **key** –
- **value** –

Returns

start ()
Starts the server

Clients

class pylm.clients.**Client** (*server_name: str*, *db_address: str*, *push_address: str = None*, *sub_address: str = None*, *session: str = None*, *logging_level: int = 20*, *this_config=False*)
Client to connect to parallel servers

Parameters

- **server_name** – Server you are connecting to
- **db_address** – Address for the cache service, for first connection or configuration.
- **push_address** – Address of the push service of the server to pull from
- **sub_address** – Address of the pub service of the server to subscribe to
- **session** – Name of the pipeline if the session has to be reused
- **logging_level** – Specify the logging level.
- **this_config** – Do not fetch configuration from the server

delete (*key*)
Deletes data in the server's internal cache.

Parameters **key** – Key of the data to be deleted

Returns

eval (*function*, *payload: bytes*, *messages: int = 1*, *cache: str = ''*)
Execute single job.

Parameters

- **function** – Sting or list of strings following the format `server.function`.
- **payload** – Binary message to be sent
- **messages** – Number of messages expected to be sent back to the client
- **cache** – Cache data included in the message

Returns If messages=1, the result data. If messages > 1, a list with the results

get (*key*)
Gets a value from server's internal cache

Parameters **key** – Key for the data to be selected.

Returns Value

job (*function*, *generator*, *messages: int = 9223372036854775807*, *cache: str = ''*)
Submit a job with multiple messages to a server.

Parameters

- **function** – Sting or list of strings following the format `server.function`.
- **payload** – A generator that yields a series of binary messages.
- **messages** – Number of messages expected to be sent back to the client. Defaults to infinity (sys.maxsize)
- **cache** – Cache data included in the message

Returns an iterator with the messages that are sent back to the client.

set (*value: bytes*, *key=None*)

Sets a key value pare in the remote database. If the key is not set, the function returns a new key. Note that the order of the arguments is reversed from the usual.

> **Warning:** If the session attribute is specified, all the keys will be prepended with the session id.

Parameters

- **value** – Value to be stored
- **key** – Key for the k-v storage

Returns New key or the same key

CHAPTER 5

Low level API documentation

The router and the parts

class pylm.parts.core.**BypassInbound**(*name*, *listen_address*, *socket_type*, *reply=True*, *bind=False*, *logger=None*, *cache=None*, *messages=9223372036854775807*)

Generic inbound part that does not connect to the router.

Parameters

- **name** – Name of the component
- **listen_address** – ZMQ socket address to listen to
- **socket_type** – ZMQ inbound socket type
- **reply** – True if the listening socket blocks waiting a reply
- **bind** – True if the component has to bind instead of connect.
- **logger** – Logger instance
- **cache** – Access to the server cache

recv(*reply_data=None*)

Receives, yields and returns reply_data if needed

Parameters **reply_data** – Message to send if connection needs an answer.

class pylm.parts.core.**BypassOutbound**(*name*, *listen_address*, *socket_type*, *reply=True*, *bind=False*, *logger=None*, *cache=None*, *messages=9223372036854775807*)

Generic inbound component that does not connect to the broker.

Parameters

- **name** – Name of the component
- **listen_address** – ZMQ socket address to listen to
- **socket_type** – ZMQ inbound socket type
- **reply** – True if the listening socket blocks waiting a reply
- **bind** – True if the socket has to bind instead of connect

- **logger** – Logger instance
- **cache** – Access to the cache of the server

class pylm.parts.core.**Inbound**(*name*, *listen_address*, *socket_type*, *reply=True*, *broker_address='inproc://broker'*, *bind=False*, *logger=None*, *cache=None*, *messages=9223372036854775807*)

Generic part that connects a REQ socket to the broker, and a socket to an inbound external service.

Parameters

- **name** – Name of the component
- **listen_address** – ZMQ socket address to listen to
- **socket_type** – ZMQ inbound socket type
- **reply** – True if the listening socket blocks waiting a reply
- **broker_address** – ZMQ socket address for the broker
- **bind** – True if socket has to bind, instead of connect.
- **logger** – Logger instance
- **cache** – Cache for shared data in the server
- **messages** – Maximum number of inbound messages. Defaults to infinity.

handle_feedback(*message_data*)

Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()

Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)

Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()

Call this function to start the component

class pylm.parts.core.**Outbound**(*name*, *listen_address*, *socket_type*, *reply=True*, *broker_address='inproc://broker'*, *bind=False*, *logger=None*, *cache=None*, *messages=9223372036854775807*)

Generic part that connects a REQ socket to the broker, and a socket to an inbound external service.

Parameters

- **name** – Name of the component
- **listen_address** – ZMQ socket address to listen to
- **socket_type** – ZMQ inbound socket type
- **reply** – True if the listening socket blocks waiting a reply
- **broker_address** – ZMQ socket address for the broker,
- **bind** – True if the socket has to bind instead of connect.
- **logger** – Logger instance
- **cache** – Access to the cache of the server
- **messages** – Maximum number of inbound messages. Defaults to infinity.

handle_feedback(*message_data*)

Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.core.**Router**(*inbound_address='inproc://inbound'*, *outbound_address='inproc://outbound'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
Router for the internal event-loop. It is a ROUTER socket that blocks waiting for the parts to send something. This is more a bus than a broker.

Parameters

- **inbound_address** – Valid ZMQ bind address for inbound parts
- **outbound_address** – Valid ZMQ bind address for outbound parts
- **logger** – Logger instance
- **cache** – Global cache of the server
- **messages** – Maximum number of inbound messages. Defaults to infinity.
- **messages** – Number of messages allowed before the router starts buffering.

register_inbound(*name*, *route=''*, *block=False*, *log=''*)
Register component by name.

Parameters

- **name** – Name of the component. Each component has a name, that uniquely identifies it to the broker
- **route** – Each message that the broker gets from the component may be routed to another component. This argument gives the name of the target component for the message.
- **block** – Register if the component is waiting for a reply.
- **log** – Log message for each inbound connection.

Returns

register_outbound(*name*, *route=''*, *log=''*)
Register outbound component by name

Parameters

- **name** – Name of the component
- **route** – Each message sent back to the component can be routed
- **log** – Logging for each message that comes from the router.

Returns

The server templates

class pylm.parts.servers.**ServerTemplate**(*logging_level=20*, *router_messages=9223372036854775807*)
Low-level tool to build a server from parts.

Parameters **logging_level** – A correct logging level from the logging module. Defaults to INFO.

It has important attributes that you may want to override, like

Cache The key-value database that the server should use

Logging_level Controls the log output of the server.

Router Here's the router, you may want to change its attributes too.

preset_cache(***kwargs*)
Send the following keyword arguments as cache variables. Useful for configuration variables that the workers or the clients fetch straight from the cache.

Parameters **kwargs** –

register_bypass(*part*, *name=''*, *listen_address=''*, ***kwargs*)
Register a bypass part to this server

Parameters

- **part** – part class
- **name** – part name
- **listen_address** – Valid ZeroMQ address listening to the exterior
- **kwargs** – Additional keyword arguments to pass to the part

register_inbound(*part*, *name=''*, *listen_address=''*, *route=''*, *block=False*, *log=''*, ***kwargs*)
Register inbound part to this server.

Parameters

- **part** – part class
- **name** – Name of the part
- **listen_address** – Valid ZeroMQ address listening to the exterior
- **route** – Outbound part it routes to
- **block** – True if the part blocks waiting for a response
- **log** – Log message in DEBUG level for each message processed.
- **kwargs** – Additional keyword arguments to pass to the part

register_outbound(*part*, *name=''*, *listen_address=''*, *route=''*, *log=''*, ***kwargs*)
Register outbound part to this server

Parameters

- **part** – part class
- **name** – Name of the part
- **listen_address** – Valid ZeroMQ address listening to the exterior
- **route** – Outbound part it routes the response (if there is) to
- **log** – Log message in DEBUG level for each message processed
- **kwargs** – Additional keyword arguments to pass to the part

start()
Start the server with all its parts.

Services

class pylm.parts.services.**CacheService**(*name*, *listen_address*, *logger=None*, *cache=None*, *messages=9223372036854775807*)

Cache service for clients and workers

class pylm.parts.services.**HttpService**(*name*, *hostname*, *port*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*)

Similar to PullService, but the connection offered is an HTTP server that deals with inbound messages.

ACHTUNG: this thing is deliberately single threaded

debug()

Starts the component and serves the http server forever.

handle_feedback(*message_data*)

Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()

Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)

Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()

Starts the component and serves the http server forever.

class pylm.parts.services.**PubService**(*name*, *listen_address*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*, *pipelined=False*, *server=None*)

PullService binds to a socket waits for messages from a push-pull queue.

Parameters

- **name** – Name of the service
- **listen_address** – ZMQ socket address to bind to
- **broker_address** – ZMQ socket address of the broker
- **logger** – Logger instance
- **messages** – Maximum number of messages. Defaults to infinity.
- **pipelined** – Defaults to False. Pipelined if publishes to a server, False if publishes to a client.
- **server** – Name of the server, necessary to pipeline messages.

handle_feedback(*message_data*)

Abstract method. Handles the feedback from the broker

Parameters **message_data** –

handle_stream(*message*)

Handle the stream of messages.

Parameters **message** – The message about to be sent to the next step in the cluster

Returns topic (str) and message (PalmMessage)

The default behaviour is the following. If you leave this function unchanged and pipeline is set to False, the topic is the ID of the client, which makes the message return to the client. If the pipeline

parameter is set to True, the topic is set as the name of the server and the step of the message is incremented by one.

You can alter this default behaviour by overriding this function. Take into account that the message is also available in this function, and you can change other parameters like the stage or the function.

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.services.**PullService**(*name*, *listen_address*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
PullService binds to a socket waits for messages from a push-pull queue.

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.services.**PushPullService**(*name*, *push_address*, *pull_address*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
Push-Pull Service to connect to workers

handle_feedback(*message_data*)
To be overriden. Handles the feedback from the broker :param message_data: :return:

reply_feedback()
To be overriden. Returns the feedback if the component has to reply. :return:

scatter(*message_data*)
To be overriden. Picks a message and returns a generator that multiplies the messages to the broker. :param message_data: :return:

class pylm.parts.services.**PushService**(*name*, *listen_address*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
PullService binds to a socket waits for messages from a push-pull queue.

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.services.**RepBypassService**(*name*, *listen_address*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
Generic connection that opens a Rep socket and bypasses the broker.

recv(*reply_data=None*)
Receives, yields and returns reply_data if needed

Parameters **reply_data** – Message to send if connection needs an answer.

class pylm.parts.services.**RepService**(*name*, *listen_address*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
RepService binds to a given socket and returns something.

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.services.**WorkerPullService**(*name*, *listen_address*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
This is a particular pull service that does not modify the messages that the broker sends.

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.services.**WorkerPushService**(*name*, *listen_address*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
This is a particular push service that does not modify the messages that the broker sends.

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

Gateways

class pylm.parts.gateways.**GatewayDealer**(*name=''*, *listen_address='inproc://gateway_router'*, *broker_address='inproc://broker'*, *cache=None*, *logger=None*, *messages=9223372036854775807*)
Generic component that connects a REQ socket to the broker, and a socket to an inbound external service.

This part is a companion for the gateway router, and has to connect to it to work properly:

```
-->|          v-------------------------------------|
   |-->Gateway Router ---> |-\  /->| --> *Dealer* --|
<--|                       |  \/    |
                           |  /\    |
     Workers -> Inbound -> |-/  \->| --> Outbound --> Workers
```

Parameters

- **broker_address** – ZMQ socket address for the broker,
- **logger** – Logger instance
- **cache** – Access to the cache of the server
- **messages** – Maximum number of inbound messages. Defaults to infinity.

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.gateways.**GatewayRouter**(*name='gateway_router'*, *listen_address='inproc://gateway_router'*, *broker_address='inproc://broker'*, *cache=<pylm.persistence.kv.DictDB object>*, *logger=None*, *messages=9223372036854775807*)
Router that allows a parallel server to connect to multiple clients. It also allows to recv messages from a dealer socket that feeds back the output from the same router. The goal is to provide blocking jobs to multiple clients.

Parameters

- **broker_address** – Broker address

- **cache** – K-v database for the cache
- **logger** – Logger class
- **messages** – Number of messages until it is shut down

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.gateways.**HttpGateway**(*name=''*, *listen_address='inproc://gateway_router'*, *hostname=''*, *port=8888*, *cache=<pylm.persistence.kv.DictDB object>*, *logger=None*)
HTTP Gateway that adapts an HTTP server to a PALM master

Parameters

- **name** – Name of the part
- **listen_address** – Address listening for reentrant messages
- **hostname** – Hostname for the HTTP server
- **port** – Port for the HTTP server
- **cache** – Cache of the master
- **logger** – Logger class

class pylm.parts.gateways.**MyServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)
Server that handles multiple requests

close_request(*request*)
Called to clean up an individual request.

fileno()
Return socket file number.

Interface required by selector.

finish_request(*request*, *client_address*)
Finish one request by instantiating RequestHandlerClass.

get_request()
Get the request and client address from the socket.

May be overridden.

handle_error(*request*, *client_address*)
Handle an error gracefully. May be overridden.

The default is to print a traceback and continue.

handle_request()
Handle one request, possibly blocking.

Respects self.timeout.

handle_timeout()
Called if no new request arrives within self.timeout.

Overridden by ForkingMixIn.

process_request(*request*, *client_address*)
Start a new thread to process the request.

process_request_thread(*request*, *client_address*)
Same as in BaseServer but as a thread.

In addition, exception handling is done here.

serve_forever(*poll_interval=0.5*)
Handle one request at a time until shutdown.

Polls for shutdown every poll_interval seconds. Ignores self.timeout. If you need to do periodic tasks, do them in another thread.

server_activate()
Called by constructor to activate the server.

May be overridden.

server_bind()
Override server_bind to store the server name.

server_close()
Called to clean-up the server.

May be overridden.

service_actions()
Called by the serve_forever() loop.

May be overridden by a subclass / Mixin to implement any code that needs to be run during the loop.

shutdown()
Stops the serve_forever loop.

Blocks until the loop has finished. This must be called while serve_forever() is running in another thread, or it will deadlock.

shutdown_request(*request*)
Called to shutdown and close an individual request.

verify_request(*request*, *client_address*)
Verify the request. May be overridden.

Return True if we should proceed with this request.

Connections

class pylm.parts.connections.**HttpConnection**(*name*, *listen_address*, *reply=True*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *max_workers=4*, *messages=9223372036854775807*)
Similar to PushConnection. An HTTP client deals with outbound messages.

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.connections.**PullBypassConnection**(*name*, *listen_address*, *logger=None*, *messages=9223372036854775807*)
Generic connection that opens a Sub socket and bypasses the broker.

recv(*reply_data=None*)
Receives, yields and returns reply_data if needed

Parameters **reply_data** – Message to send if connection needs an answer.

class pylm.parts.connections.**PullConnection**(*name*, *listen_address*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
PullConnection is a component that connects a REQ socket to the broker, and a PULL socket to an external service.

Parameters

- **name** – Name of the component
- **listen_address** – ZMQ socket address to listen to
- **broker_address** – ZMQ socket address for the broker
- **logger** – Logger instance
- **messages** – Maximum number of inbound messages. Defaults to infinity.

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.connections.**PushBypassConnection**(*name*, *listen_address*, *logger=None*, *messages=9223372036854775807*)
Generic connection that sends a message to a sub service. Good for logs or metrics.

class pylm.parts.connections.**PushConnection**(*name*, *listen_address*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
PushConnection is a component that connects a REQ socket to the broker, and a PUSH socket to an external service.

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.connections.**RepConnection**(*name*, *listen_address*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
RepConnection is a component that connects a REQ socket to the broker, and a REP socket to an external service.

Parameters

- **name** – Name of the component
- **listen_address** – ZMQ socket address to listen to
- **broker_address** – ZMQ socket address for the broker
- **logger** – Logger instance
- **messages** – Maximum number of inbound messages. Defaults to infinity.

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

Parameters **message_data** –

start()
Call this function to start the component

class pylm.parts.connections.**SubConnection**(*name*, *listen_address*, *previous*, *broker_address='inproc://broker'*, *logger=None*, *cache=None*, *messages=9223372036854775807*)
Part that connects to a Pub service and subscribes to its message queue

Parameters

- **name** –
- **listen_address** –
- **previous** –
- **broker_address** –
- **logger** –
- **cache** –
- **messages** –

handle_feedback(*message_data*)
Abstract method. Handles the feedback from the broker

Parameters **message_data** –

reply_feedback()
Abstract method. Returns the feedback if the component has to reply.

scatter(*message_data*)
Abstract method. Picks a message and returns a generator that multiplies the messages to the broker.

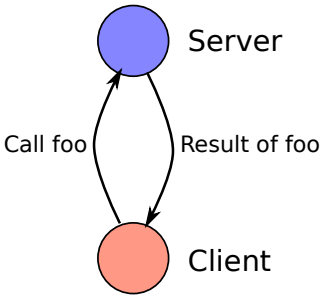
Parameters **message_data** –

start()
Call this function to start the component

CHAPTER 6

Examples

Simple server and client communication

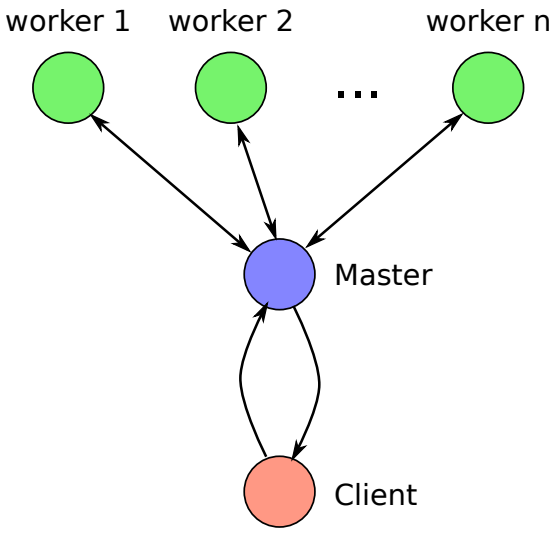



```
from pylm.servers import Server
import logging


class MyServer(Server):
    def foo(self, message):
        self.logger.warning('Got a message')
        return b'you sent me ' + message


if __name__ == '__main__':
    server = MyServer('my_server',
                      db_address='tcp://127.0.0.1:5555',
                      pull_address='tcp://127.0.0.1:5556',
                      pub_address='tcp://127.0.0.1:5557',
                      log_level=logging.DEBUG
                      )
    server.start()
```

```
from pylm.clients import Client

client = Client('my_server', 'tcp://127.0.0.1:5555')

if __name__ == '__main__':
```

```
    result = client.eval('my_server.foo', b'a message')
    print('Client got: ', result)
```

Simple parallel server and client communication




```
from pylm.servers import Master


server = Master(name='server',
                pull_address='tcp://127.0.0.1:5555',
                pub_address='tcp://127.0.0.1:5556',
                worker_pull_address='tcp://127.0.0.1:5557',
                worker_push_address='tcp://127.0.0.1:5558',
                db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
from pylm.servers import Worker
from uuid import uuid4
import sys


class MyWorker(Worker):
    def foo(self, message):
        return self.name.encode('utf-8') + b' processed ' + message

server = MyWorker(str(uuid4()), 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()

```

```
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
```

```
    for response in client.job('server.foo',
                               repeat(b'a message', 10),
                               messages=10):
        print(response)
```

Cache operation for the standalone parallel version

```
from pylm.servers import Master

server = Master(name='server',
                pull_address='tcp://127.0.0.1:5555',
                pub_address='tcp://127.0.0.1:5556',
                worker_pull_address='tcp://127.0.0.1:5557',
                worker_push_address='tcp://127.0.0.1:5558',
                db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
from pylm.servers import Worker
import sys


class MyWorker(Worker):
    def foo(self, message):
        data = self.get('cached')
        return self.name.encode('utf-8') + data + message

server = MyWorker(sys.argv[1],
                  db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    client.set(b' cached data ', 'cached')
    print(client.get('cached'))

    for response in client.job('server.foo', repeat(b'a message', 10),␣
↪messages=10):
        print(response)
```

Usage of the scatter function

```
from pylm.servers import Master


class MyMaster(Master):
    def scatter(self, message):
        for i in range(3):
            yield message
```

```

server = MyMaster(name='server',
                  pull_address='tcp://127.0.0.1:5555',
                  pub_address='tcp://127.0.0.1:5556',
                  worker_pull_address='tcp://127.0.0.1:5557',
                  worker_push_address='tcp://127.0.0.1:5558',
                  db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
from pylm.servers import Worker
import sys


class MyWorker(Worker):
    def foo(self, message):
        data = self.get('cached')
        return self.name.encode('utf-8') + data + message

server = MyWorker(sys.argv[1],
                  db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    client.set(b' cached data ', 'cached')
    print(client.get('cached'))

    for response in client.job('server.foo', repeat(b'a message', 10), messages=30):
        print(response)
```

Usage of the gather function

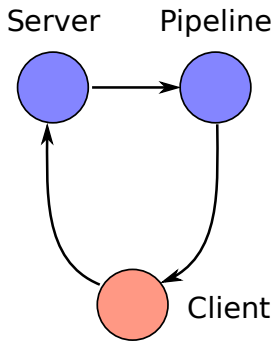
```
from pylm.servers import Master


class MyMaster(Master):
    def __init__(self, *args, **kwargs):
        self.counter = 0
        super(MyMaster, self).__init__(*args, **kwargs)

    def scatter(self, message):
        for i in range(3):
            yield message

    def gather(self, message):
        self.counter += 1

        if self.counter == 30:
            yield self.change_payload(message, b'final message')
        else:
```

```
            yield message

server = MyMaster(name='server',
                  pull_address='tcp://127.0.0.1:5555',
                  pub_address='tcp://127.0.0.1:5556',
                  worker_pull_address='tcp://127.0.0.1:5557',
                  worker_push_address='tcp://127.0.0.1:5558',
                  db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
from pylm.servers import Worker
import sys


class MyWorker(Worker):
    def foo(self, message):
        data = self.get('cached')
        return self.name.encode('utf-8') + data + message

server = MyWorker(sys.argv[1], db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    client.set(b' cached data ', 'cached')
    print(client.get('cached'))

    for response in client.job('server.foo', repeat(b'a message', 10), messages=30):
        print(response)
```

A pipelined message stream




```
from pylm.servers import Server
import logging


```

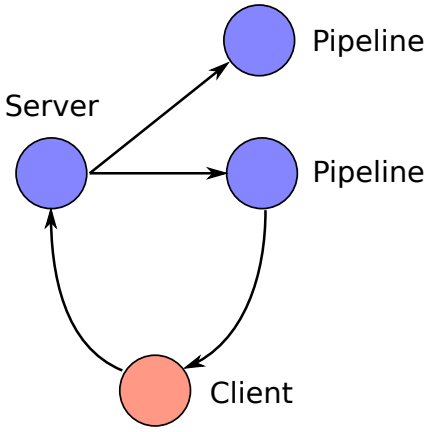
```
class MyServer(Server):
    def foo(self, message):
        self.logger.warning('Got a message')
        return b'you sent me ' + message


if __name__ == '__main__':
    server = MyServer('my_server',
                      db_address='tcp://127.0.0.1:5555',
                      pull_address='tcp://127.0.0.1:5556',
                      pub_address='tcp://127.0.0.1:5557',
                      pipelined=True,
                      log_level=logging.DEBUG
                      )
    server.start()
```

```
from pylm.servers import Pipeline
import logging


class MyPipeline(Pipeline):
    def foo(self, message):
        self.logger.warning('Got a message')
        return b'and I pipelined ' + message


if __name__ == '__main__':
    server = MyPipeline('my_pipeline',
                        db_address='tcp://127.0.0.1:5560',
                        sub_address='tcp://127.0.0.1:5557',
                        pub_address='tcp://127.0.0.1:5561',
                        previous='my_server',
                        to_client=True,
                        log_level=logging.DEBUG
                        )
    server.start()
```

```
from pylm.clients import Client

client = Client('my_server', 'tcp://127.0.0.1:5555',
                sub_address='tcp://127.0.0.1:5561')

if __name__ == '__main__':
    result = client.eval(['my_server.foo', 'my_pipeline.foo'], b'a message')
    print('Client got: ', result)
```

A pipelined message stream forming a tee

```
from pylm.servers import Server
import logging


class MyServer(Server):
    def foo(self, message):
        self.logger.warning('Got a message')
        return b'you sent me ' + message


if __name__ == '__main__':
```




```
    server = MyServer('my_server',
                      db_address='tcp://127.0.0.1:5555',
                      pull_address='tcp://127.0.0.1:5556',
                      pub_address='tcp://127.0.0.1:5557',
                      pipelined=True,
                      log_level=logging.DEBUG
                      )
    server.start()
```

```
from pylm.servers import Pipeline
import logging


class MyPipeline(Pipeline):
    def foo(self, message):
        self.logger.warning('Got a message')
        return b'and I pipelined ' + message


if __name__ == '__main__':
    server = MyPipeline('my_pipeline',
                        db_address='tcp://127.0.0.1:5560',
                        sub_address='tcp://127.0.0.1:5557',
                        pub_address='tcp://127.0.0.1:5561',
                        previous='my_server',
                        to_client=True,
                        log_level=logging.DEBUG
                        )
    server.start()
```

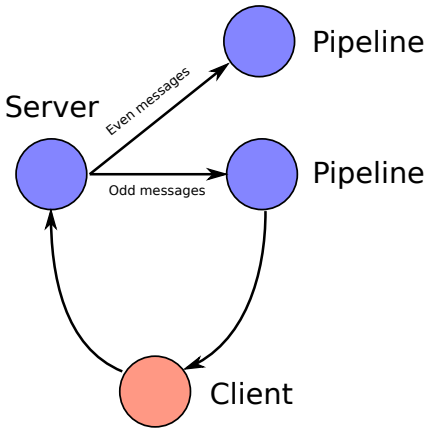
Important: If the method of a pipeline does not return any value, pylm assumes that no message has to be delivered

```
from pylm.servers import Pipeline
import logging


class MyPipeline(Pipeline):
    def foo(self, message):
        self.logger.warning('Just echo, nothing else')


if __name__ == '__main__':
```

```
    server = MyPipeline('my_pipeline',
                        db_address='tcp://127.0.0.1:5570',
                        sub_address='tcp://127.0.0.1:5557',
                        pub_address='tcp://127.0.0.1:5571',
                        previous='my_server',
                        to_client=True,
                        log_level=logging.DEBUG
                        )
    server.start()
```

```
from pylm.clients import Client

client = Client('my_server', 'tcp://127.0.0.1:5555',
                sub_address='tcp://127.0.0.1:5561')

if __name__ == '__main__':
    result = client.eval(['my_server.foo', 'my_pipeline.foo'], b'a message')
    print('Client got: ', result)
```

A pipelined message stream forming a tee and controls the stream of messages




```
from pylm.servers import Server


class MyServer(Server):
    def __init__(self, *args, **kwargs):
        super(MyServer, self).__init__(*args, **kwargs)
        self.counter = 0

    def foo(self, message):
        self.logger.info('Got a message')
        return b'you sent me ' + message

    def handle_stream(self, message):
        # if message is even
        if self.counter % 2 == 0:
            self.logger.info('Even')
            topic = 'even'

        else:
            self.logger.info('Odd')
```

```
            topic = 'odd'

        # Remember to increment the stage
        message.stage += 1

        # Increment the message counter
        self.counter += 1
        return topic, message


if __name__ == '__main__':
    server = MyServer('my_server',
                      db_address='tcp://127.0.0.1:5555',
                      pull_address='tcp://127.0.0.1:5556',
                      pub_address='tcp://127.0.0.1:5557',
                      pipelined=True)
    server.start()
```

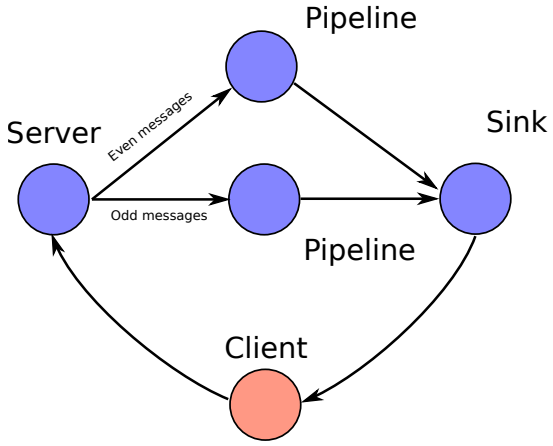
```
from pylm.servers import Pipeline


class MyPipeline(Pipeline):
    def foo(self, message):
        self.logger.info('Got a message')
        return b'and I pipelined ' + message


if __name__ == '__main__':
    server = MyPipeline('my_pipeline',
                        db_address='tcp://127.0.0.1:5560',
                        sub_address='tcp://127.0.0.1:5557',
                        pub_address='tcp://127.0.0.1:5561',
                        previous='even',
                        to_client=True)
    server.start()
```

```
from pylm.servers import Pipeline


class MyPipeline(Pipeline):
    def foo(self, message):
        self.logger.info('Echo: {}'.format(message.decode('utf-8')))


if __name__ == '__main__':
    server = MyPipeline('my_pipeline',
                        db_address='tcp://127.0.0.1:5570',
                        sub_address='tcp://127.0.0.1:5557',
                        pub_address='tcp://127.0.0.1:5571',
                        previous='odd',
                        to_client=True)
    server.start()
```

```
from pylm.clients import Client
from itertools import repeat

client = Client('my_server', 'tcp://127.0.0.1:5555',
                sub_address='tcp://127.0.0.1:5561')

if __name__ == '__main__':
    for response in client.job(['my_server.foo', 'my_pipeline.foo'],
```

```
                                repeat(b'a message', 10),
                                messages=5):
        print('Client got: ', response)
```

A pipelined message stream forming a tee and controls the stream of messages with a sink




```
from pylm.servers import Server


class MyServer(Server):
    def __init__(self, *args, **kwargs):
        super(MyServer, self).__init__(*args, **kwargs)
        self.counter = 0

    def foo(self, message):
        self.logger.info('Got a message')
        return b'you sent me ' + message

    def handle_stream(self, message):
        # if message is even
        if self.counter % 2 == 0:
            self.logger.info('Even')
            topic = 'even'

        else:
            self.logger.info('Odd')
            topic = 'odd'

        # Remember to increment the stage
        message.stage += 1

        # Increment the message counter
        self.counter += 1
        return topic, message


if __name__ == '__main__':
    server = MyServer('my_server',
                      db_address='tcp://127.0.0.1:5555',
                      pull_address='tcp://127.0.0.1:5556',
                      pub_address='tcp://127.0.0.1:5557',
```

```
                        pipelined=True)
    server.start()
```

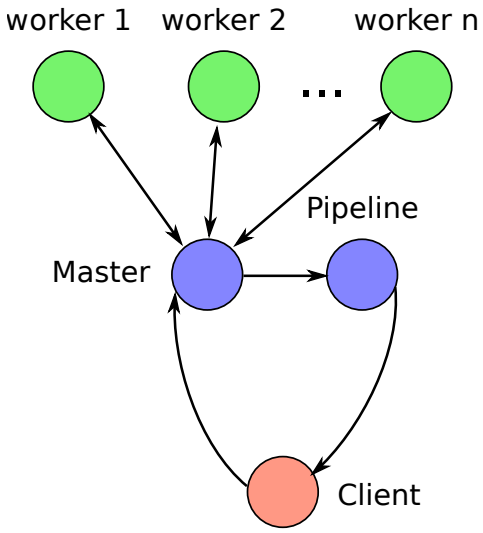
```
from pylm.servers import Pipeline


class MyPipeline(Pipeline):
    def foo(self, message):
        self.logger.info('Got a message')
        return b'and I pipelined ' + message


if __name__ == '__main__':
    server = MyPipeline('my_pipeline',
                        db_address='tcp://127.0.0.1:5570',
                        sub_address='tcp://127.0.0.1:5557',
                        pub_address='tcp://127.0.0.1:5571',
                        previous='odd',
                        to_client=False)
    server.start()
```

```
from pylm.servers import Pipeline


class MyPipeline(Pipeline):
    def foo(self, message):
        self.logger.info('Got a message')
        return b'and I pipelined ' + message


if __name__ == '__main__':
    server = MyPipeline('my_pipeline',
                        db_address='tcp://127.0.0.1:5560',
                        sub_address='tcp://127.0.0.1:5557',
                        pub_address='tcp://127.0.0.1:5561',
                        previous='even',
                        to_client=False)
    server.start()
```

```
from pylm.servers import Sink
import logging


class MySink(Sink):
    def foo(self, message):
        self.logger.warning('Got a message')
        return b'and gathered ' + message


if __name__ == '__main__':
    server = MySink('my_sink',
                    db_address='tcp://127.0.0.1:5580',
                    sub_addresses=['tcp://127.0.0.1:5561', 'tcp://127.0.0.1:5571'],
                    pub_address='tcp://127.0.0.1:5581',
                    previous=['my_pipeline', 'my_pipeline'],
                    to_client=True,
                    log_level=logging.DEBUG
    )
    server.start()

```

```
from pylm.clients import Client
from itertools import repeat

client = Client('my_server', 'tcp://127.0.0.1:5555',
                sub_address='tcp://127.0.0.1:5581')

if __name__ == '__main__':
    for response in client.job(['my_server.foo', 'my_pipeline.foo', 'my_sink.foo'],
                               repeat(b'a message', 10),
                               messages=10):
        print('Client got: ', response)
```

Connecting a pipeline to a master




```
from pylm.servers import Master
import logging


server = Master(name='server',
                pull_address='tcp://127.0.0.1:5555',
                pub_address='tcp://127.0.0.1:5556',
                worker_pull_address='tcp://127.0.0.1:5557',
                worker_push_address='tcp://127.0.0.1:5558',
                db_address='tcp://127.0.0.1:5559',
                pipelined=True)

if __name__ == '__main__':
    server.start()
```

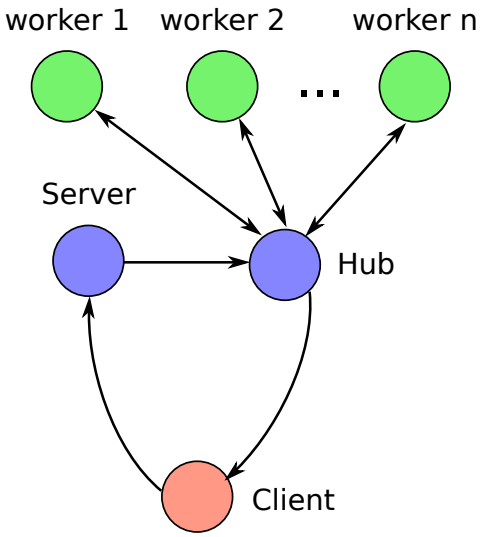
```
from pylm.servers import Pipeline
import logging


class MyPipeline(Pipeline):
    def foo(self, message):
        self.logger.info('Got a message')
        return b'and I pipelined ' + message


if __name__ == '__main__':
```

```
    server = MyPipeline('my_pipeline',
                        db_address='tcp://127.0.0.1:5560',
                        sub_address='tcp://127.0.0.1:5556',
                        pub_address='tcp://127.0.0.1:5561',
                        previous='server',
                        to_client=True)
    server.start()
```

```
from pylm.servers import Worker
import sys


class MyWorker(Worker):
    def foo(self, message):
        self.logger.info('Processed')
        return self.name.encode('utf-8') + b' processed ' + message

server = MyWorker(sys.argv[1], 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559',
                sub_address='tcp://127.0.0.1:5561')

if __name__ == '__main__':
    for response in client.job(['server.foo', 'my_pipeline.foo'],
                               repeat(b'a message', 10),
                               messages=10):
        print(response)
```

Connecting a hub to a server



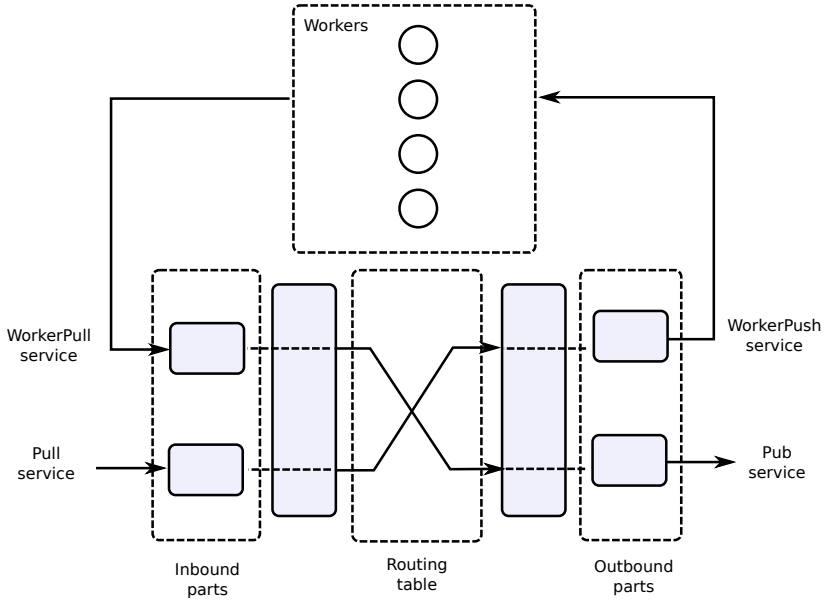

```
from pylm.servers import Server
import logging

```

```

class MyServer(Server):
    def foo(self, message):
        self.logger.warning('Got a message')
        return b'you sent me ' + message


if __name__ == '__main__':
    server = MyServer('server',
                      db_address='tcp://127.0.0.1:5555',
                      pull_address='tcp://127.0.0.1:5556',
                      pub_address='tcp://127.0.0.1:5557',
                      pipelined=True,
                      log_level=logging.DEBUG
                      )
    server.start()
```

```
from pylm.servers import Hub
import logging


server = Hub(name='hub',
             sub_address='tcp://127.0.0.1:5557',
             pub_address='tcp://127.0.0.1:5558',
             worker_pull_address='tcp://127.0.0.1:5559',
             worker_push_address='tcp://127.0.0.1:5560',
             db_address='tcp://127.0.0.1:5561',
             previous='server',
             pipelined=False)

if __name__ == '__main__':
    server.start()
```

```
from pylm.servers import Worker
import sys


class MyWorker(Worker):
    def foo(self, message):
        self.logger.info('Processed')
        return self.name.encode('utf-8') + b' processed ' + message

server = MyWorker(sys.argv[1], 'tcp://127.0.0.1:5561')

if __name__ == '__main__':
    server.start()
```

```
from pylm.clients import Client

client = Client('server', 'tcp://127.0.0.1:5555',
                sub_address='tcp://127.0.0.1:5558')

if __name__ == '__main__':
    result = client.eval(['server.foo', 'hub.foo'], b'a message')
    print('Client got: ', result)
```



Building a master server from its components

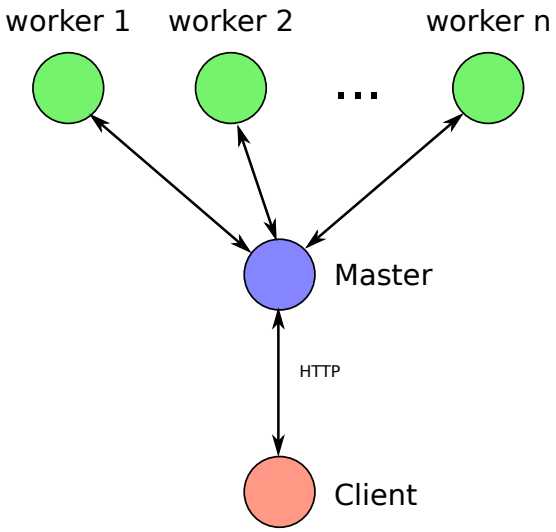
```
from pylm.parts.servers import ServerTemplate
from pylm.parts.services import PullService, PubService, WorkerPullService, WorkerPushService, \
    CacheService

server = ServerTemplate()

db_address = 'tcp://127.0.0.1:5559'
pull_address = 'tcp://127.0.0.1:5555'
pub_address = 'tcp://127.0.0.1:5556'
worker_pull_address = 'tcp://127.0.0.1:5557'
worker_push_address = 'tcp://127.0.0.1:5558'

server.register_inbound(PullService, 'Pull', pull_address, route='WorkerPush')
server.register_inbound(WorkerPullService, 'WorkerPull', worker_pull_address, route='Pub')
server.register_outbound(WorkerPushService, 'WorkerPush', worker_push_address)
server.register_outbound(PubService, 'Pub', pub_address)
server.register_bypass(CacheService, 'Cache', db_address)
server.preset_cache(name='server',
                    db_address=db_address,
                    pull_address=pull_address,
                    pub_address=pub_address,
                    worker_pull_address=worker_pull_address,
                    worker_push_address=worker_push_address)

if __name__ == '__main__':
    server.start()
```

```
import sys

from pylm.servers import Worker


class MyWorker(Worker):
    def foo(self, message):
        return self.name.encode('utf-8') + b' processed ' + message
```

```

server = MyWorker(sys.argv[1], 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    for response in client.job('server.foo', repeat(b'a message', 10),␣
↪messages=10):
        print(response)
```

Turning a master into a web server with the HTTP gateway



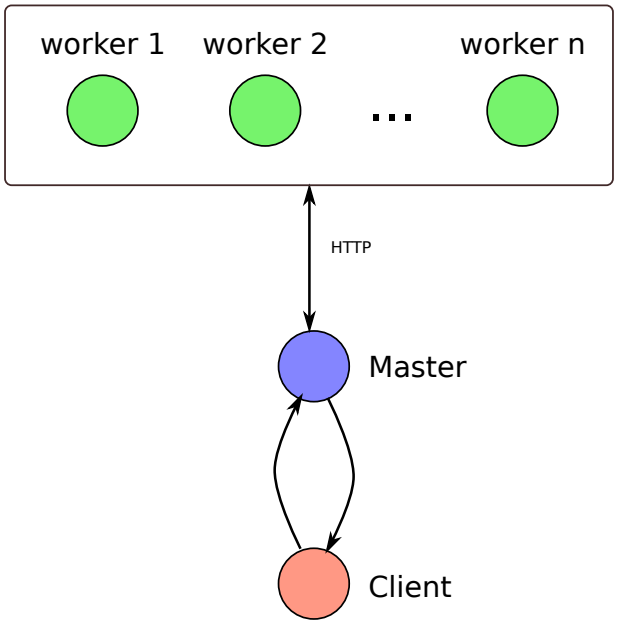

```
from pylm.parts.servers import ServerTemplate
from pylm.parts.services import WorkerPullService, WorkerPushService, \
    CacheService
from pylm.parts.gateways import GatewayDealer, GatewayRouter, HttpGateway

server = ServerTemplate()

worker_pull_address = 'tcp://127.0.0.1:5557'
worker_push_address = 'tcp://127.0.0.1:5558'
db_address = 'tcp://127.0.0.1:5559'

server.register_inbound(GatewayRouter,
                        'gateway_router',
                        'inproc://gateway_router',
                        route='WorkerPush')
server.register_outbound(GatewayDealer,
                         'gateway_dealer',
                         listen_address='inproc://gateway_router')
server.register_bypass(HttpGateway,
                       name='HttpGateway',
                       listen_address='inproc://gateway_router',
                       hostname='localhost',
```

```
                        port=8888)
server.register_inbound(WorkerPullService, 'WorkerPull', worker_pull_address,
                        route='gateway_dealer')
server.register_outbound(WorkerPushService, 'WorkerPush', worker_push_address)
server.register_bypass(CacheService, 'Cache', db_address)

server.preset_cache(name='server',
                    db_address=db_address,
                    worker_pull_address=worker_pull_address,
                    worker_push_address=worker_push_address)

if __name__ == '__main__':
    server.start()
```

```
import sys

from pylm.servers import Worker


class MyWorker(Worker):
    def function(self, message):
        return b'acknowledged'

server = MyWorker(sys.argv[1], db_address='tcp://127.0.0.1:5559')

if __name__ == '__main__':
    server.start()
```

```
import requests

print(requests.get('http://localhost:8888/function').content)
```

Using server-less infrastructure as workers via the HTTP protocol




```
from pylm.parts.servers import ServerTemplate
from pylm.parts.services import PullService, PubService, CacheService
```

```python
from pylm.parts.connections import HttpConnection

server = ServerTemplate()

db_address = 'tcp://127.0.0.1:5559'
pull_address = 'tcp://127.0.0.1:5555'
pub_address = 'tcp://127.0.0.1:5556'

server.register_inbound(PullService, 'Pull', pull_address,
                        route='HttpConnection')
server.register_outbound(HttpConnection, 'HttpConnection',
                         'http://localhost:8888', route='Pub', max_workers=1)
server.register_outbound(PubService, 'Pub', pub_address)
server.register_bypass(CacheService, 'Cache', db_address)
server.preset_cache(name='server',
                    db_address=db_address,
                    pull_address=pull_address,
                    pub_address=pub_address)

if __name__ == '__main__':
    server.start()
```

```python
from pylm.remote.server import RequestHandler, DebugServer, WSGIApplication


class MyHandler(RequestHandler):
    def foo(self, payload):
        return payload + b' processed online'

app = WSGIApplication(MyHandler)

if __name__ == '__main__':
    server = DebugServer('localhost', 8888, MyHandler)
    server.serve_forever()
```

```python
from pylm.clients import Client
from itertools import repeat

client = Client('server', 'tcp://127.0.0.1:5559')

if __name__ == '__main__':
    for response in client.job('server.foo', repeat(b'a message', 10),␣
↪messages=10):
        print(response)
```

CHAPTER 7

Adapting your components to the pylm registry

If you plan to run moderately complex clusters of PALM components, you have to take a look at the pylm registry. The registry is a centralized service that manages the configuration, the execution and the monitoring of components.

The central registry is a web service that stores the following things:

- The configuration file of the cluster
- The status of the configuration of the cluster, useful to check if enough servers have been added to it.
- The output of all the components that were launched with the runner, a script provided by the registry.

To use the capabilities of the registry you have to turn your components in executable scripts in a particular way. Despite you can force the runner to run almost anything, we recommend you to follow these simple guidelines.

- Turn each component into an executable script. It may be the usual python script starting with the shebang or an entry point in your `setup.py`.
- Use `argparse.ArgumentParser` to let the script get the runtime arguments
- To allow testing your script, it is a good practice to define a main function that then is called with the usual `if __name__...`.

What follows is a simple example that adapts the components of a simple parallel server that can be found here (*Simple parallel server and client communication*).

```
from pylm.servers import Master
from argparse import ArgumentParser


def parse_arguments():
    parser = ArgumentParser()
    parser.add_argument('--name', type=str,
                        help="Name of the component", required=True)
    parser.add_argument('--pull', type=str,
                        help="Tcp address of the pull service",
                        default='tcp://127.0.0.1:5555')
    parser.add_argument('--pub', type=str,
                        help="Tcp address of the pub service",
                        default='tcp://127.0.0.1:5556')
    parser.add_argument('--wpush', type=str,
                        help="Tcp address of the push-to-workers service",
```

```
                        default='tcp://127.0.0.1:5557')
    parser.add_argument('--wpull', type=str,
                        help="Tcp address of the pull-from-workers service",
                        default='tcp://127.0.0.1:5558')
    parser.add_argument('--db', type=str,
                        help="Tcp address of the cache service",
                        default='tcp://127.0.0.1:5559')

    return parser.parse_args()


def main():
    args = parse_arguments()
    server = Master(name=args.name,
                    pull_address=args.pull,
                    pub_address=args.pub,
                    worker_pull_address=args.wpull,
                    worker_push_address=args.wpush,
                    db_address=args.db)
    server.start()


if __name__ == '__main__':
    main()
```

```
from pylm.servers import Worker
from argparse import ArgumentParser
from uuid import uuid4


class MyWorker(Worker):
    def foo(self, message):
        return self.name.encode('utf-8') + b' processed' + message


def parse_arguments():
    parser = ArgumentParser()
    parser.add_argument('--name', type=str, help='Name of this worker '
                                                 'component',
                        default=str(uuid4()))
    parser.add_argument('--db', type=str,
                        help='Address for the db socket of the master '
                             'component',
                        default='tcp://127.0.0.1:5559')

    return parser.parse_args()


def main():
    args = parse_arguments()
    server = MyWorker(args.name, args.db)
    server.start()


if __name__ == '__main__':
    main()
```

```
from pylm.clients import Client
from itertools import repeat
from argparse import ArgumentParser


```

```python
def parse_arguments():
    parser = ArgumentParser()
    parser.add_argument('--server', type=str,
                        help="Name of the component you want to connect to",
                        required=True)
    parser.add_argument('--function', type=str,
                        help="Name of the function you want to call",
                        required=True)
    parser.add_argument('--db', type=str,
                        help="tcp address of the cache service of the master "
                             "component",
                        default='tcp://127.0.0.1:5559')
    return parser.parse_args()


def main():
    args = parse_arguments()
    client = Client(args.server, args.db)

    for response in client.job('.'.join([args.server, args.function]),
                               repeat(b' a message', 10),
                               messages=10):
        print(response)


if __name__ == '__main__':
    main()
```

From the testing point of view, there is little difference on how to run the master:

```
$> python master.py --name foo
2017-02-01 10:11:41,485 - root - INFO - Starting the router
2017-02-01 10:11:41,485 - root - INFO - Starting inbound part Pull
2017-02-01 10:11:41,485 - root - INFO - Starting inbound part WorkerPull
2017-02-01 10:11:41,485 - root - INFO - Starting outbound part WorkerPush
2017-02-01 10:11:41,485 - root - INFO - Starting outbound part Pub
2017-02-01 10:11:41,485 - root - INFO - Starting bypass part Cache
2017-02-01 10:11:41,485 - root - INFO - Launch router
2017-02-01 10:11:41,485 - root - INFO - Inbound Pull connects to WorkerPush
2017-02-01 10:11:41,486 - root - INFO - b'Pull' successfully started
2017-02-01 10:11:41,486 - root - INFO - Inbound WorkerPull connects to Pub
2017-02-01 10:11:41,486 - root - INFO - b'WorkerPull' successfully started
2017-02-01 10:11:41,486 - root - INFO - b'WorkerPush' successfully started
2017-02-01 10:11:41,487 - root - INFO - Outbound WorkerPush connects to exterior
2017-02-01 10:11:41,487 - root - INFO - b'Pub' successfully started
2017-02-01 10:11:41,488 - root - INFO - Outbound Pub connects to exterior
```

The worker:

```
$> python worker.py
2017-02-01 10:12:16,674 - e29029... - INFO - Got worker push address: ...
2017-02-01 10:12:16,674 - e29029... - INFO - Got worker pull address: ...
```

And the client in the form of a launcher:

```
$> python launcher.py --server test --function foo
2017-02-01 10:12:18,394 - INFO - Fetching configuration from the server
2017-02-01 10:12:18,394 - INFO - CLIENT 29796938-e3d7-4f9a-b69b...
2017-02-01 10:12:18,395 - INFO - CLIENT 29796938-e3d7-4f9a-b69b...
b'e29029a3-6943-4797-a8c2-6005134d8228 processed a message'
b'e29029a3-6943-4797-a8c2-6005134d8228 processed a message'
b'e29029a3-6943-4797-a8c2-6005134d8228 processed a message'
```

```
b'e29029a3-6943-4797-a8c2-6005134d8228 processed a message'
b'e29029a3-6943-4797-a8c2-6005134d8228 processed a message'
b'e29029a3-6943-4797-a8c2-6005134d8228 processed a message'
b'e29029a3-6943-4797-a8c2-6005134d8228 processed a message'
b'e29029a3-6943-4797-a8c2-6005134d8228 processed a message'
b'e29029a3-6943-4797-a8c2-6005134d8228 processed a message'
b'e29029a3-6943-4797-a8c2-6005134d8228 processed a message'
```

With the addition that now the components are ready to be run with the registry.

CHAPTER 8

Beyond Python

Sometimes you need to implement a very simple piece in another language that is not Python, but you don't want to wait for a different PALM implementation to exist. It's probable that you only need a worker, which is the simplest piece among the whole PALM ecosystem.

A Simple worker in C++

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

This project has been funded by the Spanish Ministry of Economy and Competitivity under the grant IDI-20150936, cofinanced with FEDER funds.

Python Module Index

p

pylm.clients, 34
pylm.parts.connections, 46
pylm.parts.core, 37
pylm.parts.gateways, 44
pylm.parts.servers, 39
pylm.parts.services, 41
pylm.servers, 27

Index

B

BypassInbound (class in pylm.parts.core), 37
BypassOutbound (class in pylm.parts.core), 37

C

CacheService (class in pylm.parts.services), 41
change_payload() (pylm.servers.Hub method), 27
change_payload() (pylm.servers.Master method), 29
Client (class in pylm.clients), 34
close_request() (pylm.parts.gateways.MyServer method), 45

D

debug() (pylm.parts.services.HttpService method), 41
delete() (pylm.clients.Client method), 34
delete() (pylm.servers.MuxWorker method), 31
delete() (pylm.servers.Worker method), 33

E

echo() (pylm.servers.Pipeline method), 31
echo() (pylm.servers.Server method), 32
echo() (pylm.servers.Sink method), 33
eval() (pylm.clients.Client method), 34

F

fileno() (pylm.parts.gateways.MyServer method), 45
finish_request() (pylm.parts.gateways.MyServer method), 45

G

GatewayDealer (class in pylm.parts.gateways), 44
GatewayRouter (class in pylm.parts.gateways), 44
gather() (pylm.servers.Hub method), 27
gather() (pylm.servers.Master method), 29
get() (pylm.clients.Client method), 34
get() (pylm.servers.MuxWorker method), 31
get() (pylm.servers.Worker method), 33
get_request() (pylm.parts.gateways.MyServer method), 45

H

handle_error() (pylm.parts.gateways.MyServer method), 45
handle_feedback() (pylm.parts.connections.HttpConnection method), 46
handle_feedback() (pylm.parts.connections.PullConnection method), 47
handle_feedback() (pylm.parts.connections.PushConnection method), 47
handle_feedback() (pylm.parts.connections.RepConnection method), 48
handle_feedback() (pylm.parts.connections.SubConnection method), 48
handle_feedback() (pylm.parts.core.Inbound method), 38
handle_feedback() (pylm.parts.core.Outbound method), 38
handle_feedback() (pylm.parts.gateways.GatewayDealer method), 44
handle_feedback() (pylm.parts.gateways.GatewayRouter method), 45
handle_feedback() (pylm.parts.services.HttpService method), 41
handle_feedback() (pylm.parts.services.PubService method), 41
handle_feedback() (pylm.parts.services.PullService method), 42
handle_feedback() (pylm.parts.services.PushPullService method), 42
handle_feedback() (pylm.parts.services.PushService method), 42
handle_feedback() (pylm.parts.services.RepService method), 43
handle_feedback() (pylm.parts.services.WorkerPullService method), 43
handle_feedback() (pylm.parts.services.WorkerPushService method), 43
handle_request() (pylm.parts.gateways.MyServer method), 45
handle_stream() (pylm.parts.services.PubService method), 41
handle_stream() (pylm.servers.Hub method), 28
handle_stream() (pylm.servers.Master method), 29
handle_stream() (pylm.servers.Pipeline method), 31
handle_stream() (pylm.servers.Server method), 32
handle_stream() (pylm.servers.Sink method), 33
handle_timeout() (pylm.parts.gateways.MyServer

method), 45
HttpConnection (class in pylm.parts.connections), 46
HttpGateway (class in pylm.parts.gateways), 45
HttpService (class in pylm.parts.services), 41
Hub (class in pylm.servers), 27

I

Inbound (class in pylm.parts.core), 38

J

job() (pylm.clients.Client method), 34

M

Master (class in pylm.servers), 29
MuxWorker (class in pylm.servers), 30
MyServer (class in pylm.parts.gateways), 45

O

Outbound (class in pylm.parts.core), 38

P

Pipeline (class in pylm.servers), 31
preset_cache() (pylm.parts.servers.ServerTemplate method), 40
preset_cache() (pylm.servers.Hub method), 28
preset_cache() (pylm.servers.Master method), 29
process_request() (pylm.parts.gateways.MyServer method), 46
process_request_thread() (pylm.parts.gateways.MyServer method), 46
PubService (class in pylm.parts.services), 41
PullBypassConnection (class in pylm.parts.connections), 47
PullConnection (class in pylm.parts.connections), 47
PullService (class in pylm.parts.services), 42
PushBypassConnection (class in pylm.parts.connections), 47
PushConnection (class in pylm.parts.connections), 47
PushPullService (class in pylm.parts.services), 42
PushService (class in pylm.parts.services), 42
pylm.clients (module), 34
pylm.parts.connections (module), 46
pylm.parts.core (module), 37
pylm.parts.gateways (module), 44
pylm.parts.servers (module), 39
pylm.parts.services (module), 41
pylm.servers (module), 27

R

recv() (pylm.parts.connections.PullBypassConnection method), 47
recv() (pylm.parts.core.BypassInbound method), 37
recv() (pylm.parts.services.RepBypassService method), 43
register_bypass() (pylm.parts.servers.ServerTemplate method), 40
register_bypass() (pylm.servers.Hub method), 28
register_bypass() (pylm.servers.Master method), 30
register_inbound() (pylm.parts.core.Router method), 39
register_inbound() (pylm.parts.servers.ServerTemplate method), 40
register_inbound() (pylm.servers.Hub method), 28
register_inbound() (pylm.servers.Master method), 30
register_outbound() (pylm.parts.core.Router method), 39
register_outbound() (pylm.parts.servers.ServerTemplate method), 40
register_outbound() (pylm.servers.Hub method), 28
register_outbound() (pylm.servers.Master method), 30
RepBypassService (class in pylm.parts.services), 43
RepConnection (class in pylm.parts.connections), 48
reply_feedback() (pylm.parts.connections.HttpConnection method), 46
reply_feedback() (pylm.parts.connections.PullConnection method), 47
reply_feedback() (pylm.parts.connections.PushConnection method), 47
reply_feedback() (pylm.parts.connections.RepConnection method), 48
reply_feedback() (pylm.parts.connections.SubConnection method), 48
reply_feedback() (pylm.parts.core.Inbound method), 38
reply_feedback() (pylm.parts.core.Outbound method), 38
reply_feedback() (pylm.parts.gateways.GatewayDealer method), 44
reply_feedback() (pylm.parts.gateways.GatewayRouter method), 45
reply_feedback() (pylm.parts.services.HttpService method), 41
reply_feedback() (pylm.parts.services.PubService method), 42
reply_feedback() (pylm.parts.services.PullService method), 42
reply_feedback() (pylm.parts.services.PushPullService method), 42
reply_feedback() (pylm.parts.services.PushService method), 42
reply_feedback() (pylm.parts.services.RepService method), 43
reply_feedback() (pylm.parts.services.WorkerPullService method), 43
reply_feedback() (pylm.parts.services.WorkerPushService method), 43
RepService (class in pylm.parts.services), 43
Router (class in pylm.parts.core), 39

S

scatter() (pylm.parts.connections.HttpConnection method), 46
scatter() (pylm.parts.connections.PullConnection method), 47
scatter() (pylm.parts.connections.PushConnection method), 48

scatter() (pylm.parts.connections.RepConnection method), 48
scatter() (pylm.parts.connections.SubConnection method), 49
scatter() (pylm.parts.core.Inbound method), 38
scatter() (pylm.parts.core.Outbound method), 39
scatter() (pylm.parts.gateways.GatewayDealer method), 44
scatter() (pylm.parts.gateways.GatewayRouter method), 45
scatter() (pylm.parts.services.HttpService method), 41
scatter() (pylm.parts.services.PubService method), 42
scatter() (pylm.parts.services.PullService method), 42
scatter() (pylm.parts.services.PushPullService method), 42
scatter() (pylm.parts.services.PushService method), 42
scatter() (pylm.parts.services.RepService method), 43
scatter() (pylm.parts.services.WorkerPullService method), 43
scatter() (pylm.parts.services.WorkerPushService method), 44
scatter() (pylm.servers.Hub method), 28
scatter() (pylm.servers.Master method), 30
serve_forever() (pylm.parts.gateways.MyServer method), 46
Server (class in pylm.servers), 32
server_activate() (pylm.parts.gateways.MyServer method), 46
server_bind() (pylm.parts.gateways.MyServer method), 46
server_close() (pylm.parts.gateways.MyServer method), 46
ServerTemplate (class in pylm.parts.servers), 39
service_actions() (pylm.parts.gateways.MyServer method), 46
set() (pylm.clients.Client method), 35
set() (pylm.servers.MuxWorker method), 31
set() (pylm.servers.Worker method), 34
shutdown() (pylm.parts.gateways.MyServer method), 46
shutdown_request() (pylm.parts.gateways.MyServer method), 46
Sink (class in pylm.servers), 32
start() (pylm.parts.connections.HttpConnection method), 47
start() (pylm.parts.connections.PullConnection method), 47
start() (pylm.parts.connections.PushConnection method), 48
start() (pylm.parts.connections.RepConnection method), 48
start() (pylm.parts.connections.SubConnection method), 49
start() (pylm.parts.core.Inbound method), 38
start() (pylm.parts.core.Outbound method), 39
start() (pylm.parts.gateways.GatewayDealer method), 44
start() (pylm.parts.gateways.GatewayRouter method), 45
start() (pylm.parts.servers.ServerTemplate method), 40
start() (pylm.parts.services.HttpService method), 41
start() (pylm.parts.services.PubService method), 42
start() (pylm.parts.services.PullService method), 42
start() (pylm.parts.services.PushService method), 43
start() (pylm.parts.services.RepService method), 43
start() (pylm.parts.services.WorkerPullService method), 43
start() (pylm.parts.services.WorkerPushService method), 44
start() (pylm.servers.Hub method), 29
start() (pylm.servers.Master method), 30
start() (pylm.servers.MuxWorker method), 31
start() (pylm.servers.Pipeline method), 32
start() (pylm.servers.Server method), 32
start() (pylm.servers.Sink method), 33
start() (pylm.servers.Worker method), 34
SubConnection (class in pylm.parts.connections), 48

V

verify_request() (pylm.parts.gateways.MyServer method), 46

W

Worker (class in pylm.servers), 33
WorkerPullService (class in pylm.parts.services), 43
WorkerPushService (class in pylm.parts.services), 43