lymph Documentation

Release 0.16.0-dev

deliveryhero

February 25, 2016

Contents

lymph is a framework for Python services. lymph intends to be the glue between your services so you don't get sticky fingers.

This is what a service looks like with lymph:

```
import lymph


class Greeting(lymph.interface):

    @lymph.rpc()
    def greet(self, name):
        '''
        Returns a greeting for the given name
        '''
        print(u'Saying to hi to %s' % name)
        self.emit(u'greeted', {'name': name})
        return u'Hi, %s' % name
```

Contents:

CHAPTER 1

Installation

Installing lymph itself (for Python 2.7 or 3.4) is as simple as:

```
pip install lymph
```

Yet, in order to make full use of lymph you'll also need to install lymph's dependencies: ZooKeeper (for service discovery) and RabbitMQ (for events) and have them running.

If these are already set up, you can skip straight and continue the next chapter.

1.1 Installing dependencies

The RabbitMQ server's default configuration is enough for development and testing. For detailed information on how to configure ZooKeeper refer to the ZooKeeper webpage and the Getting Started Guide. However, it's default configuration should also be enough.

1.1.1 On Ubuntu

If you haven't already install Python essentials:

```
$ sudo apt-get install build-essential python-dev python-pip
```

Install and start ZooKeeper using:

```
$ sudo apt-get install zookeeper zookeeperd
$ sudo service zookeeper start
```

ZooKeeper's configuration file is located at `/etc/zookeeper/conf/zoo.cfg`.

Install and start the RabbitMQ server:

```
$ sudo apt-get install rabbitmq-server
$ sudo service rabbitmq-server start
```

1.1.2 On OSX

Install RabbitMQ and ZooKeeper:

```
$ brew install rabbitmq zookeeper
```

ZooKeeper's configuration file is located at `/usr/local/etc/zookeeper/zoo.cfg`.

CHAPTER 2

User guide

You can find an introduction to lymph in Max Brauer's import lymph presentation. It attempts to get you up and running and covers most features of lymph.

CHAPTER 3

Command Line Interface

Lymph's cli lets you run, discover, inspect and interact with services. It is built to be your toolbelt when developing and running services. The cli is extensible. You can write custom lymph subcommands, e.g. lymph top.

Note: Many of lymph's commands produce unicode output. Therefore, you'll have to set your locale (LC_ALL or LC_CTYPE) to UTF-8.

If you want to pipe lymph commands with Python 2, you might have to set PYTHONIOENCODING to UTF-8 as well.

Check the *FAQ*.

This is an overview of lymph's cli. We don't document every command's arguments and parameters on purpose. Each is self-documenting:

```
$ lymph help <command>  # or
$ lymph <command> --help
```

3.1 lymph list

Prints a list of all available commands with their description.

3.2 lymph instance

Runs a service instance.

3.3 lymph discover

Discovers all available services and their instances, e.g.:

3.4 lymph inspect

Prints the RPC interface of a service with signature and docstrings.

3.5 lymph request

Invokes an RPC method of a service and prints the response.

3.6 lymph emit

Emits an event in the event system.

3.7 lymph subscribe

Subscribes to an event type and prints every occurence.

3.8 lymph node

This is lymph's development server. It can run any number of services with any number of instances as well as any other dependency.

3.9 lymph shell

Starts an interactive Python shell for service instance, locally or remotely.

3.10 lymph config

Prints configuration for inspection

CHAPTER 4

Topic guides

4.1 Running services

4.1.1 Overview

There are two ways to start services with lymph. You can either start a lymph service directly from the command line using `lymph instance` or define all the services to start in a configuration file and start them all with lymph's development server `lymph node`.

4.1.2 lymph instance

This command runs a single service instance given a config file with *interfaces*

```
lymph instance --config=$PATH_TO_CONFIG_FILE
```

4.1.3 Writing configuration files for `lymph instance`

A configuration file of a lymph service requires the following sections:

- container
- interfaces

You need to define a separate configuration file for each service or instance setup. If you have many services running, which would be the normal case in a productive lymph setup, the same information about `container` would be present in each file. In order to avoid having to copy the same information into every file and obtain a configuration mess, it is possible to set a default configuration file where lymph extracts the necessary information. This is usually the `.lymph.yml` file, which is also needed by `lymph node` (the standard way to start lymph services, see :doc: `lymph node` below).

The default configuration file is set using the `LYMPH_NODE_CONFIG` environmental variable and is usually set by

```
$ export LYMPH_NODE_CONFIG="/path/to/lymph/config/.lymph.yml"
```

`interfaces`

Each service needs to have its `interfaces` defined in the respective service configuration file. The `interfaces` section defines which endpoints a service has (a service can have multiple endpoints) and the configuration of each endpoint (you can have multiple endpoints to the same service interface class, with different configurations).

The interfaces section is made up of

interfaces.<name>
Mapping from service name to instance configuration that will be passed to the implementation's `lymph.Service.apply_config()` method.

which gives a name to a specific interface (i.e. the `namespace` part when referencing a service). If the interface has been named, it needs to be linked to a class that is a subclass of :class: *lymph.Interface*.

interfaces.<name>.class
The class that implements this interface, e.g. a subclass of *`lymph.Interface`*.

After the interface class has been defined, any additional configuration can be passed on to the interface class by defining any

interfaces.<name>.<param>
The whole `interfaces.<name>` dict is available as configuration for the interface class.

A simple example for an interface definition is:

```
interfaces:
    echo:
        class: echo:EchoService
```

and another example showing the use of additional interface options and the definition of multiple interfaces:

```
interfaces:
    echo_small_valley:
        class: echo:EchoService
        delay: 1

    echo_large_valley:
        class: echo:EchoService
        delay: 10
```

lymph node

This command will start instances of services as defined in a configuration file. It will load as many instances as specified for each defined service. By default it will read the `.lymph.yml` file, but through the `--config` option, you can specify another configuration. You run this command by initiating:

```
$ lymph node
```

4.1.4 Configuring `lymph node`

instances.<name>

Besides the usual configuration sections for the `container`, a section on `instances` needs to be added. In this section, each service is defined, together with the `lymph instance` command to start it, and the number of processes `numprocesses` each service should have.

instances.<name>.command:
A command (does not necessarily have to be a `lymph instance` command) that will be spawned by `lymph node`

instances.<name>.numprocesses:
Number of times the defined command is spawned

An example of such an `instances` configuration block:

```
instances:
    echo:
        command: lymph instance --config=conf/echo.yml
        numprocesses: 10

    demo:
        command: lymph instance --config=conf/demo.yml
```

4.2 Configuration

Lymph needs to be configured so that it knows how and where to find its service registry and its event system.

```
container:
    registry:
        class: lymph.discovery.zookeeper:ZookeeperServiceRegistry
        zkclient:
            class: kazoo.client:KazooClient
            hosts: 120.0.0.1:2181

    events:
        class: lymph.events.kombu:KombuEventSystem
        transport: amqp
        hostname: 127.0.0.1
```

You can find this sample configuration file in `conf/sample-node.yml`.

4.2.1 Environment Variables

Lymph config files support environment variable substitution for string values:

```
key: protocol://$(env.USER):$(env.PASSWORD)@host/path
```

You can also inject structured environment configuration from a YAML file, e.g. `lymph -c conf.yml --vars=vars.yml command`:

```
# vars.yml
key: value
struct:
    foo: bar
```

```
# conf.yml
foo: $(var.key)
var: $(var.struct)
interpolation: prefix_$(var.key)_suffix
```

Dependencies

Lymph supports a way to inject dependencies from configuration file.

You start by defining a top level "dependencies" key that you want to inject and share between different components, this should be in the format

```
dependencies:
    <name>:
        class: <class path>
        <extra class arguments>
```

Then you can reference a dependency anywhere in your configuration by using the `dep:<name>` format, as shown in the example above.

Container Configuration

`container.ip`
use this IP address. The `--ip` option for **`lymph`** takes precedence. Default: `127.0.0.1`.

`container.port`
Use this port for the service endpoint. The `--port` option for **`lymph`** takes precedence. If no port is configured, lymph will pick a random port.

`container.class`
the container implementation. You probably don't have to change this. Default: `lymph.core.container:Container`

`container.log_endpoint`
the local ZeroMQ endpoint that should be used to publish logs via the *_zmqpub* handler.

`container.pool_size`
Size of the pool of Greenlets, default is unlimited.

Registry Configuration

`container.registry.class`

Defaults to `lymph.discovery.zookeeper:ZookeeperServiceRegistry`

4.2.2 ZooKeeper

To use ZooKeeper for serivce discovery set `class` to `lymph.discovery.zookeeper:ZookeeperServiceRegistry`.

`container.registry.zkclient`

A reference to zookeeper client either as a dependency or a class.

Event Configuration

`container.events.class`

4.2.3 Kombu

To use the kombu backend set `class` to `lymph.events.kombu:KombuEventSystem`. All other keys will be passed as keyword arguments to the kombu Connection.

4.2.4 Null

The null backend doesn't transport any events. Set `class` to `lymph.events.null.NullEventSystem` if that is what you want.

Metrics Configuration

`container.metrics.tags`

A dictionary of tags to be sent with all monitoring data from this container, e.g.

```
container:
    metrics:
        tags:
            env: $(env.NAMESPACE)
```

Interface Configuration

`interfaces.<name>`
: Mapping the name to instance which will be used to send requests and discover this interface. This name is also configuration that will be passed to the implementation's *`lymph.Interface.apply_config()`* method.

`interfaces.<name>.class`
: The class that implements this interface, e.g. a subclass of *`lymph.Interface`*.

Components Configuration

Extra component can be defined under the components namespace e.g `SerialEventHandler`.

```
components:
    SerialEventHandler:
        zkclient: dep:kazoo
```

Logging Configuration

`logging`

Logging can be configured in standard dictConfig format. In addition to the setup provided via `logging`, one formatter and two handlers are created. You can change them by providing different configuration for the ids.

The formatter (`_trace`) includes the trace-id and is used for both built-in handlers. The `_zmqpub` handler publishes log messages on a ZeroMQ pub socket (see *container.log_endpoint*).

The `_console` handler writes messages to either stdout or the file given by `--logfile`. The level of the handler is set to `--loglevel`.

Debugging Configuration

`debug.backdoor_ip`

Specify which ip address the backdoor terminal should listen too.

4.3 Tests

You can test if your installation of lymph has been successful by running the unittests. You'll also have to set `ZOOKEEPER_PATH` to the directory that contains your ZooKeeper binaries (e.g. `/usr/share/java` on Ubuntu).

You can then run the tests with either **'tox'_** or `nosetests` directly.

4.4 Events

4.4.1 Overview

Lymph uses events to communicate between various services. For this, RabbitMQ is currently used to do the event passing. Services can emit events and subscribe to the queue to consume events.

The way events are communicated is pluggable and can be easily exchanged at will. The unittests for instance are using a local event system `LocalEventSystem` to not rely on RabbitMQ.

4.4.2 Other event brokers

Lymph allows other event brokers to be easily incorporated. Lymph also provides the following additional event broker services:

- Null (a black hole)
- Local (simple event broker that runs in the scope of the main lymph process)
- Kombu (interfaces to RabbitMQ as a broker using the kombu library)

The event broker service can be set in the `.lymph.yml` configuration file:

```
container:
    events:
        class: lymph.events.kombu:KombuEventSystem
        transport: amqp
        hostname: 127.0.0.1
```

See *Event Configuration* for details.

4.4.3 Subscribing to events

In order to have methods executed whenever a given event is emitted, you decorate the function with the `event` decorator.

@event(**event_types*)

> **Parameters** **event_types** – may contain wildcards (`#` matching zero or more words and `*` matches one word), e.g. `'subject.*'`
>
> Marks the decorated interface method as an event handler. The service container will automatically subscribe to given `event_types`.

```
    import lymph

    class Example(lymph.Interface):
        @lymph.event('task_done')
```

```
        def on_task_done(self, event):
            assert isinstance(event, lymph.core.events.Event)
```

A new queue will be created for every service name and event handler combination.

4.4.4 Dynamically subscribing to events

Subscribing to events using the `event` decorator only works at service instantiation time. If you need to subscribe to events at runtime, you need to use the `subscribe` decorator:

`@subscribe(*event_types, sequential=True)`
Behaves like *`lymph.event()`*, but can be used at runtime

```
    class Example(lymph.Service):
        def on_start(self):
            @self.subscribe('dynamic_event_type')
            def on_event(event):
                assert isinstance(event, lymph.core.events.Event)
```

4.4.5 Emitting events

The `lymph.Interface` provides a method for emitting events.

`lymph.Interface.emit(self, event_type, payload)`

Parameters

- **`event_type`** – name of the event
- **`payload`** – a dict of serializable data structures

A simple example of a class emitting a signal with a simple event would be:

```
class SomeClass(lymph.Interface):
        def emit_event(self):
                self.emit('simple_event', {'article': 'foo', 'quantity': 5})
```

4.4.6 Command line interface

To interact with the event system from the command line, the following commands are available:

```
$ lymph subscribe
```

and

```
$ lymph emit
```

lymph subscribe

With this command, you can register to a specific event and have all events printed out on stdout.

For the default example services, this might be:

```
$ lymph subscribe uppercase_transform_finished
uppercase_transform_finished: {'text': u'foo_282'}
uppercase_transform_finished: {'text': u'foo_283'}
uppercase_transform_finished: {'text': u'foo_284'}
...
```

This lists all the events sent to `uppercase_transform_finished` produced by the demo loop which calls the echo service. Each line represents an individual event, stating its name and its payload.

You can also subscribe to multiple events at once:

```
$ lymph subscribe event_a event_b
event_a: {u'data': u'nice'}
event_b: {u'information': u'data'}
```

lymph emit

With this command, you can manually emit a specific event from the command line. You need to specify the name of the event and provide a JSON encoded body.

For the default example services, this might be:

```
$ lymph emit uppercase_transform_finished '{"text": "bar_foo_234"}'
```

This would emit an event with the name `uppercase_transform_finished` with the given payload to any service that is listening to this event. We can inspect the events sent through the system with the *lymph subscribe* command in another terminal:

```
$ lymph subscribe uppercase_transform_finished
...
uppercase_transform_finished: {'text': u'foo_2629'}
uppercase_transform_finished: {'text': u'foo_2630'}
uppercase_transform_finished: {u'text': u'bar_foo_234'}
uppercase_transform_finished: {'text': u'foo_2631'}
...
```

We can see that the event has been routed to the instance along with all the other events from the demo loop.

4.5 RPC

4.5.1 Overview

Synchronous communication with lymph services is realised through RPC. RPC messages are sent via ØMQ. If a RPC call fails, it is the responsibility of the calling code to deal with it.

4.5.2 Registering methods as RPC callable

Any class inheriting from *`lymph.Interface`* can receive RPC calls. By specifying the `name` argument when initializing the class, the lymph service will be reachable through its interface name `name`.

By default the service is registered under the name given when you configure the service.

```
import lymph

class EchoService(lymph.Interface):
    pass
```

```
interfaces:
    echo:
        class: project.interfaces:EchoService
```

will be reachable with the service name `echo`. This is the name with which lymph knows that the RPC messages should be sent to `EchoService`.

In order to make a method in a lymph interface class RPC callable, it is sufficient to add the `@lymph.rpc()` (or `@lymph.raw_rpc()` for accessing the channel object) decorator in front of it.

@rpc
Marks the decorated interface method as an RPC method.

```
    import lymph

    class Example(lymph.Interface):
        @lymph.raw_rpc()
        def do_ack(self, channel, message):
            """
            HERE SOME FANCE HELP TEXT
            """
            assert isinstance(channel, lymph.core.channels.ReplyChannel)
            assert isinstance(message, lymph.core.messages.Message)
            channel.ack()

       @lymph.rpc()
       def echo(self, message):
           return message
```

If a docstring is specified after the RPC method definition, it will be used as a description of the service and will be returned by `lymph inspect`.

4.5.3 Difference between lymph.rpc and lymph.raw_rpc

lymph.rpc

The *lymph.rpc()* decorator is easier to understand compared to *lymph.raw_rpc()* since the former work as any Python function where what ever the RPC function return will be sent to the caller, as for exceptions there is two cases depending on the `raises` argument of *lymph.rpc()* :

- If the exception raised inside the RPC function is an instance of a class that is part of the `raises` argument then the client will see a `RemoteError`.
- Else the result will be a **NACK**.

lymph.raw_rpc

When *lymph.raw_rpc()* is used the underlying method call has to have the following form:

```
def some_rpc_method(self, channel, **kwargs):
    ...
```

The `channel` argument takes a `lymph.ReplyChannel` object which takes care of the communication from and to the RPC caller. From within the responding method, you communicate through the `channel` object with the calling party. The `ReplyChannel` object provides you with the following methods:

reply(*body*)

> **Parameters** **body** – reply
>
> sends `body` as a reply back to the caller

```
import lymph

class EchoService(lymph.Interface):

    @lymph.raw_rpc()
    def echo(self, channel, text=None):
        channel.reply(text)
```

ack(*unless_reply_sent=False*)

> **Parameters** **unless_reply_sent** – only send the acknowledgment if a reply has already been sent
>
> sends an acknowledgment to the caller.

nack(*unless_reply_sent=False*)

> **Parameters** **unless_reply_sent** – only send the non-acknowledgment if a reply has already been sent
>
> sends a non-acknowledgment to the caller.

error(*body*)

> **Parameters** **body** – error
>
> sends an error to the caller.

4.5.4 Sending RPC calls

In order to send RPC calls from within lymph services, you need to pass the call through the `proxy` class. You can obtain the system's proxy by calling the `proxy` method:

proxy(*address*)
> returns a proxy object that can be used to conveniently send requests to another service.

```
echo = self.proxy('echo')
result = echo.upper(text='foo')
assert result == 'FOO'
```

> This is equivalent to `self.request('echo', 'echo.upper', text='foo')`.

The proxy object proxies any method that is called in the proxy class, into a corresponding RPC call. It does not however make sure, that the RPC call actually exists. It will send the call regardless of availability and timeout accordingly if no response is obtained.

Any value that is returned by the RPC call is also returned by the call to the corresponding proxy method. In the example above, the service with the name `echo` provides the `upper(text)` endpoint. By calling the corresponding proxy method in the proxy object, the payload `text='foo'` is sent to the endpoint and its result returned and saved in the `result` variable.

RPC calls are synchronous, i.e. program execution is halted until the RPC call returns an answer or it times out. If you require asynchronous communication, please refer to Events.

4.5.5 Deferred RPC calls

By default, RPC blocks until the response is received. A deferred RPC call mechanism is available if you wish to consume the RPC response later, or simply ingore it.

The call interface is similar to making a regular RPC call, with the addition of adding .defer call after it.

In that case, the call will return a Future (the actual implementation is a gevent AsyncResult which will block only when it's .get method is called.

For instance:

```
    echo = self.proxy('echo')
    result_future = echo.upper.defer(text='foo')
    # do other stuff
    result = result_future.get()
    assert result == 'FOO'
```

4.6 HTTP

```
from lymph.web.interfaces import WebServiceInterface
from werkzeug.routing import Map, Rule
from werkzeug.wrappers import Response


class HttpHello(WebServiceInterface)
    url_map = Map([
        Rule('/hello/<string:name>/', endpoint='hello'),
    ])

    def hello(self, request, name):
        return Response('hello %s!' % name)
```

class WebServiceInterface

is_healthy()

4.6.1 Interface configuration

interfaces.<name>.healthcheck.enabled
Boolean: whether to respond to requests to `interfaces.<name>.healthcheck.endpoint`. Defaults to `True`.

interfaces.<name>.healthcheck.endpoint
Respond with 200 to requests for this path as long as `is_healthy()` returns True, and 503 otherwise. Defaults to `"/_health/"`.

interfaces.<name>.port
Listen on this port. Defaults to a random port.

interfaces.<name>.wsgi_pool_size

interfaces.<name>.tracing.request_header
Name of an HTTP request header that may provide the trace id. Defaults to `None`.

interfaces.<name>.tracing.response_header
Name of the HTTP response header that contains the trace id. Defaults to `"X-Trace-Id"`.

4.7 Serialization

4.7.1 Overview

Lymph uses msgpack to serialize events and rpc arguments. In addition to the types supported directly by msgpack, the lymph serializer also handles the following basic Python types: `set`, `datetime.datetime`, `datetime.date`, `datetime.time`, `uuid.UUID`, and `decimal.Decimal`.

4.7.2 Object level serialization

Object level serialization can be defined by implementing `_lymph_dump_` method in classes subject to serialization.

Object-level serialization can help to produce more concise code in certain situations, e.g.:

```
class Process(object):
    ...

    def _lymph_dump_(self):
        return {
            'pid': self.pid,
            'name': self.name,
        }


class Node(lymph.Interface):

    @lymph.rpc()
    def get_processes(self, service_type=None):
        procs = []
        for proc in self._processes:
            if not service_type or proc.service_type == service_type:
                procs.append(proc)
        return procs

    @lymph.rpc()
    def stop(self, service_type=None):
        for proc in self.get_processes(service_type):
            proc.stop()
```

In the example above by defining the `_lymph_dump_` in our Process class, we were able to reuse the rpc function `get_processes`.

4.8 Versioning interfaces

```
interfaces:
    echo@1.5.0:
        class: echo:Echo

    echo@2.0.0:
        class: echo:Echo2
```

4.8.1 Requesting Specific Versions

from the command line:

```
$ lymph request echo.upper@1.2 '{"text": "foo"}'
```

from code:

```
proxy = lymph.proxy('echo', version='1.1')
```

CHAPTER 5

API reference

Contents:

5.1 Service API

```
import lymph


class Echo(lymph.Interface):

    @lymph.rpc()
    def echo(self, text=None):
        return text

    @lymph.rpc()
    def upper(self, text=None):
        self.emit('uppercase_transform_finished', {'text': text})
        return text.upper()

    @lymph.event('uppercase_transform_finished')
    def on_uppercase(self, text=None):
        print "done", text
```

class lymph.**Interface**

name
The interface identifier that is used to register this service with the coordinator service. name is an instance attribute which is taken from the config of the interface taken from the config of the interface.

on_start()
Called when the service is started

on_stop()
Called when the service is stopped

apply_config(*config*)

Parameters **config** – dict

Called with instance specific configuration that is usually provided by a config file (see *Metrics Configuration*).

request(*address*, *method*, *body*)

Parameters

- **address** – the address where the request is sent to; either a ZeroMQ endpoint or a service name
- **method** – the remote method that will be called
- **body** – JSON serializable dict of parameters for the remote method

proxy(*address*)
returns a proxy object that can be used to conveniently send requests to another service.

```
echo = self.proxy('echo')
result = echo.upper(text='foo')
assert result == 'FOO'
```

This is equivalent to `self.request('echo', 'echo.upper', text='foo')`.

emit(*event_type*, *payload*, *delay=0*)

Parameters

- **event_type** – str
- **payload** – a dict of JSON serializable data structures
- **delay** – delay delivery of this event by *delay* seconds

@**subscribe**(**event_types*, *sequential=True*)
Behaves like *lymph.event()*, but can be used at runtime

```
class Example(lymph.Service):
    def on_start(self):
        @self.subscribe('dynamic_event_type')
        def on_event(event):
            assert isinstance(event, lymph.core.events.Event)
```

@lymph.**raw_rpc**
Marks the decorated interface method as an RPC method. Using this decorator the RPC function are expected to accept a `ReplyChannel` instance as a first argument.

```
import lymph

class Example(lymph.Interface):
    @lymph.raw_rpc()
    def do_something(self, channel, message):
        assert isinstance(channel, lymph.core.channels.ReplyChannel)
        assert isinstance(message, lymph.core.messages.Message)
        channel.ack()
```

@lymph.**rpc**
Marks the decorated interface method as an RPC method. The difference between this decorator and *raw_rpc()* is that the RPC functions must use return and raise like any normal Python function instead of using `channel.reply` and `channel.error`.

Parameters **raises** – tuple of exception classes that the RPC function is expected to raise.

```
import lymph

class Example(lymph.Interface):
    @lymph.rpc()
    def do_something(self, message):
        return message
```

`@lymph.`**`event`**(**event_types*, *sequential=False*)

Parameters

- **`event_types`** – may contain wildcards, e.g. `'subject.*'`
- **`sequential`** – force sequential event consumption
- **`broadcast`** – receive every event in all instances

Marks the decorated interface method as an event handler. The service container will automatically subscribe to given `event_types`. If `sequential=True`, events will be not be consumed in parallel, but one by one. If `broadcast=True`, every instance of the service will receive the event.

```
import lymph

class Example(lymph.Interface):
    @lymph.event('task_done')
    def on_task_done(self, event):
        assert isinstance(event, lymph.core.events.Event)
```

`@lymph.`**`task`**

Parameters

- **`sequential`** – force sequential task execution per instance
- **`broadcast`** – execute the task in all instances

Marks the decorated interface method as a task handler.

5.2 Core API

class `lymph.core.container.`**`ServiceContainer`**

classmethod `from_config`(*config*, ***kwargs*)

`start`()

`stop`()

`send_message`(*address*, *msg*)

Parameters

- **`address`** – the address for this message; either a ZeroMQ endpoint a service name
- **`msg`** – the *`lymph.core.messages.Message`* object that will be sent

Returns *`lymph.core.channels.ReplyChannel`*

`lookup`(*address*)

Parameters **`address`** – an lymph address

Returns *`lymph.core.services.Service`* or *`lymph.core.services.ServiceInstance`*

class `lymph.core.channels.`**`ReplyChannel`**

`reply`(*body*)

Parameters **`body`** – a JSON serializable data structure

ack()
acknowledges the request message

class lymph.core.channels.**RequestChannel**

get(*timeout=1*)

Returns *lymph.core.messages.Message*

returns the next reply message from this channel. Blocks until the reply is available. Raises `Timeout` after `timeout` seconds.

class lymph.core.messages.**Message**

id

type

subject

body

packed_body

class lymph.core.events.**Event**

type
the event type / name

body
dictionary with the payload of the message

source
id of the event source service

__getitem__(*name*)
gets an event parameter from the body

class lymph.core.services.**Service**
Normally created by *ServiceContainer.lookup()*. Service objects represent lymph services.

__iter__()
Yields all known *instances* of this service.

__len__()
Returns the number of known instances of this service.

class lymph.core.services.**ServiceInstance**
Describes a single service instance. Normally created by *ServiceContainer.lookup()*

identity
The identity string of this service instance

endpoint
The rpc endpoint for this

class lymph.core.connections.**Connection**
You can attain a connection to an lymph service instance directly from `lymph.core.container.ServiceContainer.connect()`, or from the higher-level API in `lymph.core.services`. For ZeroMQ endpoint addresses the following to statements are roughly equivalent:

```
container.connect(address)  # only works for tcp://... addresses
container.lookup(address).connect()  # will also work for service names
```

class lymph.core.interfaces.**Proxy**(*container*, *address*, *namespace=None*, *timeout=1*)

__getattr__(*self*, *name*)
Returns a callable that will execute the RPC method with the given name.

5.3 Config API

class lymph.config.**ConfigView**(*config*, *prefix*)
A ConfigView allows access to a subtree of a *Configuration* object. It implements the mapping protocol. Dotted path keys are translated into nested dictionary lookups, i.e. `cv.get('a.b')` is (roughly) equivalent to `cv.get('a').get('b')`.

If a value returned by *ConfigView* methods is a dict, it will be wrapped in a *ConfigView* itself. This – and getting dicts from a *Configuration* object – are the preferred way to create new ConfigViews.

root
A reference to the root *Configuration* instance.

class lymph.config.**Configuration**(*values=None*)

Parameters **values** – an optional initial mapping

Configuration implements the same interface as *ConfigView* in addition to the methods described here.

load(*file*, *sections=None*)
Reads yaml configuration from a file-like object. If sections is not None, only the keys given are imported

load_file(*path*, *sections=None*)
Reads yaml configuration from the file at `path`.

get_raw(*key*, *default*)
Like `get()`, but doesn't wrap dict values in *ConfigView*.

create_instance(*key*, *default_class=None*, ***kwargs*)

Parameters

- **key** – dotted config path (e.g. `"container.rpc"`)
- **default_class** – class object or fully qualified name of a class
- **kwargs** – extra keyword arguments to be passed to the factory

Creates an object from the config dict at `key`. The instance is created by a factory that is specified by its fully qualified name in a `class` key of the config dict.

If the factory has a `from_config()` method it is called with a *ConfigView* of `key`. Otherwise, the factory is called directly with the config values as keyword arguments.

Extra keyword arguments to `create_instance()` are passed through to `from_config()` or mixed into the arguments if the factory is a plain callable.

If the config doesn't have a `class` key the instance is create by `default_class`, which can be either a fully qualifed name or a factory object.

Given the following config file

```
foo:
    class: pack.age:SomeClass
    extra_arg: 42
```

you can create an instance of SomeClass

```
# in pack/age.py
class SomeClass(object):
    @classmethod
    def from_config(cls, config, **kwargs):
        assert config['extra_arg'] == 42
        assert kwargs['bar'] is True
        return cls(...)

# in any module
config = Configuration()
config.load(...)
config.create_instance('foo', bar=True)
```

get_instance(*key*, *default_class*, ***kwargs*)
Like `create_instance()`, but only creates a single instance for each key.

5.4 Web API

class `lymph.web.`**WebServiceInterface**

application
WSGI application instance that this interface is running

url_map
A werkzeug.routing.Map instance that is used to map requests to request handlers. Typically given as a class attribute.

5.5 Pattern API

`@lymph.patterns.serial_events.`**serial_event**(**event_types*, *partition_count=12*, *key=None*)

Parameters

- **event_types** – event types that should be partitioned
- **partition_count** – number of queues that should be used to partition the events
- **key** – a function that maps *Events* to string keys. This function should have two arguments in its signature: the instance of current *Interface* and instance of the handled *Event* object.

This event handler redistributes events into `partition_count` queues. These queues are then partitioned over all service instances and consumed sequentially, i.e. at most one event per queue at a time.

5.6 Components API

Components are objects that depend on a running service container. They are embedded in *Componentized* objects. Since Componentized objects themselves are components, they form a tree of *Component* instances with the container as the root. An example of a Component is `lymph.core.interfaces.Interface`.

class `lymph.core.components.`**Component**(*error_hook=None*, *pool=None*, *metrics=None*)

error_hook
A Hook object that propagates exceptions for this component. Defaults to the `error_hook` of the parent component.

pool
A pool that holds greenlets related to the component. Defaults to the `pool` of the parent component.

metrics
An *Aggregate* of metrics for this component. Defaults to the `metrics` of the parent component.

on_start()
Called when the container is started.

on_stop()
Called when the container is stopped.

spawn(*func*, **args*, ***kwargs*)
Spawns a new greenlet in the greenlet pool of this component. If `func` exits with an exception, it is reported to the `error_hook`.

class `lymph.core.components.`**Componentized**
A collection of components; itself a component.

add_component(*component*)

Parameters **component** – *Component*

Adds *component*.

on_start()
Calls *on_start()* on all added components.

on_stop()
Calls *on_stop()* on all added components.

5.7 Metrics API

To follow the metrics protocol objects must be iterable repeatedly and yield `(name, value, tags)`-triples, where `name` is a string, `value` is a float or int, and `tags` is a dict with string keys and values.

class `lymph.core.monitoring.metrics.`**Metric**(*name*, *tags=None*)
An abstract base class for single series metrics, i.e. metric objects that only yield a single triple.

__iter__()
[abstract] Yields metric values as a tuple in the form *(name, value, tags)*.

class `lymph.core.monitoring.metrics.`**Gauge**(*name*, *value=0*, *tags=None*)
A gauge is a metric that represents a single numerical value that can arbitrarily go up and down.

set(*value*)

class lymph.core.monitoring.metrics.**Callable**(*name*, *func*, *tags=None*)
Like a Gauge metric, but its value is determined by a callable.

class lymph.core.monitoring.metrics.**Counter**(*name*, *tags=None*)
A counter is a cumulative metric that represents a single numerical value that only ever goes up. A counter is typically used to count requests served, tasks completed, errors occurred, etc.

__iadd__(*value*)
Increment counter value.

class lymph.core.monitoring.metrics.**TaggedCounter**(*name*, *tags=None*)
A tagged counter is a container metric that represents multiple counters per tags. A tagged counter is typically used to track a group of counters as one e.g. request served per function name, errors ocurred per exception name, etc.

incr(*_by=1*, ***tags*)
Increment given counter `type` by `_by`.

class lymph.core.monitoring.metrics.**Aggregate**(*metrics=()*, *tags=None*)

Parameters

- **metrics** – iterable of metric objects
- **tags** – dict of tags to add to all metrics.

Aggregates a collection of metrics into a single metrics object.

add(*metric*)

Parameters **metric** – metric object

Adds the given metric to collection.

add_tags(***tags*)

Parameters **tags** – string-valued dict

Adds the given tags for all metrics.

5.8 Testings API

class lymph.testing.**RpcMockTestCase**
Base mixin test class that provice a highlevel interface for mocking remote rpc calls. By inheriting this class, test cases can supply mock return values for rpc functions.

Note: In case an rpc function is not mocked the actual RPC call will be made to the service.

rpc_mock_calls
A List of the called rpc functions.

setup_rpc_mocks(*rpc_functions*)
Setup RPC mocks by passing all mocked RPC functions as a dictionary in the form `{'<service_name>.<function_name>': <return_value>}`, in case `<return_value>` is an exception, call will raise the exception.

```
class SomeTest(RpcMockTestCase):

    def setUp(self):
        super().setUp()
```

```
            self.setup_rpc_mocks({
                'upper.upper': 'HELLO WORLD',
                'upper.echo': TypeError('...')
                ...
            })
```

update_rpc_mock(*func_name*, *return_value*)
Update a mock of an already mocked RPC function.

```
    class SomeTest(RpcMockTestCase):

        def setUp(self):
            super().setUp()
            self.setup_rpc_mocks({
                'upper.upper': 'HELLO WORLD',
                'upper.echo': 'hello world',
            })

        def test_something(self):
            self.update_rpc_mock('upper.upper', 'A NEW VALUE')
            ...
```

delete_rpc_mock(*func_name*)
Delete a mock of an already mocked RPC function.

Raises **KeyError** – In case the functions wasn't mocked previously.

```
    class SomeTest(RpcMockTestCase):

        def setUp(self):
            super().setUp()
            self.setup_rpc_mocks({
                'upper.upper': 'HELLO WORLD',
                'upper.echo': 'hello world',
            })

        def test_really_something(self):
            self.delete_rpc_mock('upper.upper')
            ...
```

assert_rpc_calls(**expected_calls*)
This method is a convenient way of asserting that rpc function calls were made in a particular way:

```
        class SomeTest(RpcMockTestCase):

            def setUp(self):
                super().setUp()
                self.setup_rpc_mocks({
                    'upper.upper': 'HELLO WORLD',
                    'upper.echo': 'hello world',
                })

            def test_something(self):
                ...

                self.assert_rpc_calls(
                    mock.call('upper.upper', text='hello world')
                )
```

`mock.call(..)` can contain PyHamcrest matchers for better and less brittle tests.

assert_any_rpc_calls(**expected_calls*)
At the opposite of `assert_rpc_calls` where you have to specify all mocked calls that were done, this method accept a list of mocked calls and assert that each one of them was done. The calls should be specified in the same order as they are made.

Note: This method do it's best to guess which function user is looking for and in case of a mismatch it try to generate a useful message for the user.

---

class lymph.testing.**EventMockTestCase**
Base mixin test class that provides a highlevel interface for mocking events emitted.

events
A List of the emitted events.

assert_events_emitted(**expected_emitted*)
This method is a convenient way of asserting that events were emitted:

```
class SomeTest(EventMockTestCase):

    def test_something(self):
        ...

        self.assert_events_emitted(
            mock.call('upper.uppercase_transform_finished', {'text': 'hello world'})
        )
```

class lymph.testing.**RPCServiceTestCase**
Test class for testing a unique RPC interface.

service_class
Interface class to test, this attribute is abstract and must be supplied by child class.

service_config
Configuration to pass to service when calling `apply_config()`.

client
Shortcut for getting default `lymph.Proxy` instance for the service under test.

get_proxy(***kwargs*)
Return a `lymph.Proxy` instance of the service under test.

request(**args*, ***kwargs*)
Low level method to send a request to service under tests return `ReplyChannel` instance.

emit(**args*, ***kwargs*)
Emit an event.

class lymph.testing.**WebServiceTestCase**
Test class for testing a unique Web interface.

client
Return a Werkzeug test client associated to web interface under test.

CHAPTER 6

The Lymph RPC Protocol

Message format:

Index	Name	Content
0	ID	a random uuid
1	Type	`REQ`, `REP`, `ACK`, `NACK`, or `ERROR`
2	Subject	method name for “REQ” messages, else: message id of the corresponding request
3	Headers	msgpack encoded header dict
4	Body	msgpack encoded body

CHAPTER 7

Glossary

service interface A collection of rpc methods and event listeners that are exposed by a service container. Interfaces are implemented as subclasses of *lymph.Interface*.

service container A service container manages rpc and event connections, service discovery, logging, and configuration for one or more service interfaces. There is one container per service instance.

Containers are *ServiceContainer* objects.

service instance A single process that runs a service container. It is usually created from the commandline with lymph instance. Each instance is assigned a unique identifier called *instances identity*.

Instances are described by *ServiceInstance* objects.

service A set of all service instances that exposes a common service interface is called a service. Though uncommon, instances may be part of more than one service.

Services are described by *Service* objects.

node A process monitor that runs service instances. You'd typically run one per machine. A node is started from the commandline with lymph node.

CHAPTER 8

FAQ

8.1 Why does lymph crash with UnicodeDecodeError: 'ascii' codec can't encode character . . . ?

Since many lymph commands produce unicode output, you have to set your locale to UTF-8, e.g. with

```
$ export LC_ALL=en_US.UTF-8
```

If you want to pipe lymph commands with Python 2, you might also have to set `PYTHONIOENCODING`

```
$ export PYTHONIOENCODING=UTF-8
```

CHAPTER 9

Contributing

We try to follow C4 (Collective Code Construction Contract) for lymph development. Issues are tracked on github. We accept code and documentation contributions via pull requests.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Symbols

A

B

C

D

E

F

G

I