dploy-lib Documentation

Release 0.0.1-dev

Reuven V. Gonzales

Sep 27, 2017

Contents

dploy-lib is a shared library used throughout the dploy system. It provides a standard library to facilitate a more coherent design between dploy's various components.

CHAPTER 1

Wait, what is dploy?

dploy is an application deployment system that is meant to be similar to systems like heroku. It utilizes many similar technologies as heroku but allows for customization at various points in the stack. dploy was designed for Blue Water Ads, by Reuven V. Gonzales. Many components of the dploy stack are provided as open sourced projects on github.

CHAPTER 2

Building Services

One of the primary functions this library serves is to aid in the creation of new dploy services. Read more here to learn how to build new services.

Getting started with services

The dploy stack is composed of a many separate services interacting with each other. The majority of these services are built using zeromq as a transport the rest are built using HTTP to communicate. Since HTTP has a plethora of tools for creating related web services dploylib does not need to do much to aid that process. However, ZeroMQ based services can be more complicated to implement. The dploylib simplifies and unifies portions of this process through it's concept of services.

What are services?

In dploy, services are complete applications composed of multiple dploy servers. Each of these servers reacts to input events on various zeromq sockets.

Simple Echo Service

The simplest way to understand services is to create a very simple service. Let's start by creating a very simple echo service.

Here is one of the simplest services you could define:

```
from dploylib import servers
from dploylib.services import Service

class EchoServer(servers.Server):
    @servers.bind_in('request', 'rep')
    def echo_request(self, socket, received):
        socket.send_envelope(received)
```

```
service = Service()
service.add_server('echo', EchoServer)

if __name__ == '__main__':
    config_dict = {
        "servers": {
            "echo": {
                "request": {
                    "uri": "tcp://127.0.0.1:5000",
                },
            },
        },
    }
    service.run(config_dict=config_dict)
```

To run simply do:

```
$ python myservice.py
```

If you run the service you will be able to interact with it as follows:

```
>>> from dploylib.transport import Socket
>>> echo_socket = Socket.new('req', 'tcp://127.0.0.1:14445')
>>> echo_socket.send_text('hello')
>>> print echo_socket.receive_text()
hello
```

Fantastic! You now have a working echo server.

If you can't tell, server and service definition was inspired by both flask and django web development frameworks. Let's have a look at what just happened:

1. First we import *servers*. This import will allow us to create our echo server easily.
2. Next we import the *Service* class. If you're familiar with flask, the service class is much like the Flask class in that it can be instantiated at the module level and run as an application later.
3. For the next few lines we define the `EchoServer` class.

1. The first line of the class makes it so we subclass from *Server*.
2. The next line decorates the method `echo_request`. The decorator *bind_in()* provides instructions to the containing server class, `EchoServer`. It tells the server class the following:
    - Bind a zeromq `REP` socket named `requests` to the server
    - The `_in` suffix on the decorator means the decorated method is the socket's input handler
3. Finally, within the method `echo_request`, we define the echo server logic. It simply gets the data it receives and sends it back to the user. When using dploylib's sockets, data is received in a standard envelope. This will be explained later.
4. The line starting with `service =` instantiates a Service object into the module's namespace. On the next line the EchoServer we defined on step 3 is registered to the service as a server named `echo` via the method `add_server`.
5. A fake configuration is defined and set saved in `config_dict`
6. Finally, the service is started by the method *run()*. It takes the fake configuration as the keyword argument, `config_dict`.

To stop the server, use control-C.

Standard service configuration

One of the things that we don't want to do is hard code configuration. However, in the previous example we hard coded the configuration into the `if __name__ == '__main__'` block. Luckily, services are not meant to be used in this way, although the facility is available for easy debugging or testing if necessary. Let's do a better job by using configuration files that we can change without touching any code.

dploy defines a standard configuration file that can is to be used with all services. The basic structure is able to translate to a multitude of configuration languages, but YAML is chosen by default due to it's writability, readability and portability to other languages.

Here is a basic configuration:

```
servers:  # Server configurations
  echo:   # Config for "echo" server
    request:   # Config for "echo" server's "request" socket
      uri: tcp://127.0.0.1:14445  # URI for "request" socket

# General configuration
general:
  someconfig1: somevalue1
  someconfig2: somevalue2
```

If this file is saved to `config.yaml` we can simplify the previous service to this:

```
from dploylib import servers
from dploylib.services import Service

class EchoServer(servers.Server):
    @servers.bind_in('request', 'rep')
    def echo_request(self, socket, received):
        socket.send_envelope(received)

service = Service()
service.add_server('echo', EchoServer)

if __name__ == '__main__':
    service.run(config_file="config.yaml")
```

Now, all we have to do to change the uri of the request server is change the config.yaml file. At this time the service does not yet have a standard command line interface. This feature is planned for the not-so-distant future.

Servers Library

Server setup for dploy-lib is inspired by heavily by flask. The syntax looks like this:

```
from dploylib import servers

class BroadcastServer(servers.Server):
    # Bind a socket but don't listen for it's input. Useful for output
    publish = servers.bind('pub', name='out')

    @servers.bind_in('pull', name='in')
```

```
    def receive_message(self, socket, envelope):
        self.sockets.publish.send_envelope(envelope)

class QueueServer(servers.Server):
    @servers.bind_in('rep', name='request')
    def receive_request(self, socket, envelope):
        object = self.handle_queue(envelope)k
        socket.send_obj(object)
```

CHAPTER 3

ZeroMQ Transport Wrapper

dploy-lib provides a simple wrapper for zeromq. It provides some convenience methods and functions, but also defines a standard messaging envelope for use in dploy applications.

Why create a wrapper?

`dploylib.transport` sets up the custom transport library used by dploy. This custom transport library is simply a wrapper around zeromq. The main purpose of creating the wrapper is to allow for the usage of dploy's *`Envelope`*. This envelope will allow for greater extension later as well as allowing the transport layer to be replaced if we ever need it. The main impetus for creating a wrapper is the inclusion of an encryption layer later down the line. However, the wrapper is also able to simplify the usage of zeromq in any particular application.

The wrapper tries to stay as close to the original API of pyzqm as possible. This is to prevent the need to learn much more than the zeromq guide provides.

Simple REQ-REP echo server

Here's the REP server `server.py`:

```
from dploylib.transport import Context

def main():
    context = Context.new()
    socket = context.socket('rep')
    socket.bind('tcp://127.0.0.1:5555')

    while True:
        text = socket.receive_text()
        socket.send_text(text)

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

Now the REQ server in `client.py`:

```
from dploylib.transport import Context

def main():
    context = Context.new()
    socket = context.socket('req')
    socket.connect('tcp://127.0.0.1:5555')

    socket.send_text('hello')
    text = socket.receive_text()

    print text

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

First run the server in one process:

```
$ python server.py
```

Then run the client and you should see this:

```
$ python client.py
hello
```

All of zeromq's socket types are available.

CHAPTER 4

Dploy Notes

Here are a collection of documents describing portions of dploy's architecture

Common Message and Data Structure Definitions

The following is a list of message and data schemas that are necessary for dploy. These structures are generally used to define communication protocols between various services. They should be language agnostic.

BuildRequest

Used to describe build jobs. These are sent to the DeployQueue.

Schema

`broadcast_id`
: A broadcast id of the format *[random-uuid]:[commit]*

`app`
: The app name

`archive_uri`
: URI to a tar.gz of the app

`commit`
: The SHA1 commit of the app

`update_message`
: A message about the update

`release_version`
: The release version to use. 0 means the latest version

BroadcastMessage

Used for broadcasting messages to the client

Schema

type
The message type

Must be `output` or `status`

body
The message body

Must be data of type *BroadcastOutputData* or *BroadcastStatusData Schema*

BroadcastOutputData

type
Must be `line` or `raw`

data
(optional) output string

BroadcastStatusData Schema

type
Must be `info`, `error`, or `completed`

data
(optional) A status message

AppBuildRequest

Used to describe app build jobs. These are sent to the BuildCenter. They are created by processing DeployRequests.

Schema

app_release
The current *AppRelease*

archive_uri
URI to a tar.gz file of the app's repository

AppRelease

Used throughout different sections of the build process. It is also a major component of the cargo file. These snapshots are also used to track versions of a particular app.

Schema

version
The release version number

app
The app name

commit
The SHA1 commit of the app

env
An *EnvVars* type

processes
A dict of the available processes and their associated commands

EnvVars

Dictionary of services and their environment variables. This is meant to be persisted in some kind of database.

ZoneDeployOrder

Instructions for a dploy-zone to deploy an app given its cargo file.

Schema

app
The app name

cargo_uri
URI to a downloadable cargo file

ZoneStopDeploy

Stop a set of running apps

Schema

apps
A list of apps to stop

CHAPTER 5

API

Services Library

class dploylib.services.**Service**(*templates=None*, *config_mapper=None*, *coordinator=None*)

The Service object provides a way to create a zeromq-based dploy service. In dploy, a the service object is in charge of a combination of `Server` objects. Each of the Server objects acts as a definition for a server which is used to spawn threads, processes, or greenlets of each server.

Parameters

- **templates** – a list of service templates to apply to this service
- **config_mapper** – default YAMLConfigMapper, configuration mapper for the service
- **coordinator** – the server coordinator for the service. Defaults to `ThreadedServerCoordinator`

add_server(*name*, *server_cls*)

Register a `Server` to the Service instance

Parameters

- **name** – name of the server
- **server_cls** – A `Server`

run(**args*, ***kwargs*)

A default method for running a service

start(*config_file=None*, *config_string=None*, *config_dict=None*)

Starts the service with the given configuration information. Configuration data is accepted from one of three different types: *a file path*, *a string*, or *a dictionary*.

Parameters

- **config_file** – file path for the configuration
- **config_string** – a string of the configuration

- **config_dict** – a dictionary for the configuration

stop()
Stop the service

wait()
Wait for the service forever or until it fails

Servers Library

class dplylib.servers.**Server**(*name*, *settings*, *control_uri*, *context*, *poll_loop=None*)

class dplylib.servers.**ServerDescription**(*server_cls*)

class dplylib.servers.**DployServer**(*name*, *settings*, *control_uri*, *context*, *poll_loop=None*)
The actual server behind the scenes

add_socket(*name*, *socket*, *handler=None*)
Add the socket and it's handler

start()
Run the poll loop for the server

dplylib.servers.**bind_in**(*name*, *socket_type*, *obj=None*)
A decorator that creates a SocketDescription describing a socket bound to receive input. The decorated function or method is used as the input event handler.

Parameters

- **name** – The name of the socket (for configuration purposes)
- **socket_type** (*str*) – The lowercase name of the zeromq socket type
- **obj** – (optional) A class or object that implements the deserialize method to deserialize incoming data

Returns A SocketDescription

dplylib.servers.**bind**(*name*, *socket_type*)
A decorator that creates a SocketDescription that describes a bound socket. This socket does not listen for input.

Parameters

- **name** – The name of the socket (for configuration purposes)
- **socket_type** (*str*) – The lowercase name of the zeromq socket type

dplylib.servers.**connect_in**(*name*, *socket_type*, *obj=None*)
A decorator that creates a SocketDescription describing a socket connected to receive input. The decorated function or method is used as the input event handler.

Parameters

- **name** – The name of the socket (for configuration purposes)
- **socket_type** (*str*) – The lowercase name of the zeromq socket type
- **obj** – (optional) A class or object that implements the deserialize method to deserialize incoming data

Returns A SocketDescription

dploylib.servers.**connect**(*name*, *socket_type*)
A decorator that creates a SocketDescription that describes a connected socket. This socket does not listen for input.

Parameters

- **name** – The name of the socket (for configuration purposes)
- **socket_type** (*str*) – The lowercase name of the zeromq socket type

Transport Library

class dploylib.transport.**Context**(*zmq_context*)
A wrapper around a zeromq Context

Parameters **zmq_context** – The zeromq context

class dploylib.transport.**Socket**(*zmq_socket*, *zmq_context*)
A wrapper around a zeromq Socket

Parameters

- **zmq_socket** – The underlying zeromq socket
- **zmq_context** – The zeromq context related to the zeromq socket

bind(*uri*)
Bind the socket to a URI

classmethod bind_new(*socket_type*, *uri*, *options=None*, *context=None*)
Create and bind a new socket

Parameters

- **socket_type** (*str*) – Name of the socket type
- **uri** – URI of the socket to bind to
- **context** – (optional) A *Context*. Defaults to creating a new *Context*

bind_to_random(*uri*, *min_port=None*, *max_port=None*, *max_tries=None*)
Bind the socket to a random port at URI

connect(*uri*)
Connect the socket to a URI

classmethod connect_new(*socket_type*, *uri*, *options=None*, *context=None*)
Create and connect a new socket

Parameters

- **socket_type** (*str*) – Name of the socket type
- **uri** – URI of the socket to connect to
- **context** – (optional) A *Context*. Defaults to creating a new *Context*

classmethod new(*socket_type*, *context=None*)
Creates a new socket

Parameters

- **socket_type** (*str*) – Name of the socket type
- **context** – (optional) A *Context*. Defaults to creating a new *Context*

receive_envelope()
Receive an Envelope

receive_obj(*handler*)
Receives an Envelope and calls an object to handle the envelope data.

Parameters **handler** – A callable that transforms the data into an object

receive_text()
Convenience method to receive plain text

send_envelope(*envelope*)
Send an Envelope

send_obj(*obj*, *id=''*)
Sends encoded an object as encoded data.

The encoding can be anything. Default is JSON. This could change later and should not affect communications.

The object must implement the method __serialize__

Parameters

- **obj** – An object that implements a serialize method that returns any data that can be serialized (ie. lists, dict, strings, ints)
- **id** – The id for the envelope. Defaults to ''

send_text(*text*, *id=''*)
Sends a simple text message

Parameters

- **text** – Text to send
- **id** – The id for the envelope. Defaults to ''

set_option(*option*, *value*)
Set a socket option

Parameters

- **option** (*str*) – Name of the option
- **value** (*str or int. Depends on the option.*) – Value of the option

class dploylib.transport.**Envelope**(*id*, *mimetype*, *data*, *request_frames=None*)
Dploy's message envelope.

This is a standard definition so that all messages are decoded the same way. The envelope is as follows (for the time being):

```
 ----------------------
| any request frames   |
 ----------------------
| empty frame if above |
 ----------------------
| id - a string or ''  |
 ----------------------
| mimetype             |
 ----------------------
| body                 |
 ----------------------
```

Note: The `id` portion of the envelope may seem like unnecessary data, but it allows the envelope to be used in pub-sub effectively.

Parameters

- **id** (*str*) – A string id for the envelope
- **mimetype** (*str*) – The mimetype for the envelope
- **data** – The envelope's body

classmethod from_raw(*raw*)
Creates an envelope from a tuple or list

Parameters **raw** (*tuple or list*) – Raw data for an envelope

classmethod new(*mimetype*, *data*, *id=''*, *request_frames=None*)
Create a new envelope. This is the preferred way to create a new envelope.

Parameters

- **mimetype** – The mimetype for the envelope
- **data** – The envelope's body
- **id** – (optional) A string id for the envelope. Defaults to ''

response_envelope(*mimetype*, *data*, *id=None*)
Shortcut to create a response envelope from the current envelope

By default this will create an envelope with the same request_frames, id and mimetype as this envelope.

transfer_object()
This is the object to be sent over the wire. The reverse of this is Envelope.from_raw

For zmq this should be an list

class dploylib.transport.**PollLoop**(*poller*)
A custom poller that automatically routes the handling of poll events

The handlers of poll events are simply callables. This only handles POLLIN events at this time.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index