pseud Documentation

Release 0.1.1dev

nicolas.delaby@ezeep.com

August 28, 2014

Contents

Initialize an RPC peer playing as a server

```
# The server
from pseud import Server


server = Server('service')
server.bind('tcp://127.0.0.1:5555')

@server.register_rpc
def hello(name):
    return 'Hello {0}'.format(name)

server.start() # this would block within its own io_loop
```

Prepare a tornado-based client

```
# The tornado client
# Assume tornado IOLoop is running
from pseud import Client


client = Client('service')
client.connect('tcp://127.0.0.1:5555')
```

then make a remote procedure call (rpc)

```
# Assume we are inside a coroutine
response = yield client.hello('Charly')
assert response == 'Hello Charly'
```

A gevent api is also available for clients

```
# The gevent client
from pseud import Client


client = Client('service')
client.connect('tcp://127.0.0.1:5555')

assert client.hello('Charly').get() == 'Hello Charly'
```

Narrative Documentation

1.1 Introduction

There are already plenty RPC libraries for Python. Many of them mature, tested and with an active community behind. So why build yet another one?

We discovered that most of those libraries make the assumption that they're running within a trusted network; that a client/server architecture means clients connect and consume resources exposed by the server and not vice versa.

RESTful APIs are great to consume them in the browser or in a simple client/server architecture. Once you add more distributed components and services to the game, running on potentially hostile networks, the common HTTP/RESTful design pattern becomes less practical. With *pseud* we can get over these limitations by providing secure, fault-tolerant, RPC style communication built for fast and easy machine to machine communication.

pseud is based on the amazing ØMQ library and pyzmq . It provides a convenient and pythonic API to hide some of the library's complexity and provides boilerplate code to save your time and headaches.

Also thanks to the ZCA, *pseud* comes with a pluggable architecture that allows easy integration within your existing stack. It is usable within any web application (Django, Pyramid, Tornado, ...).

pseud also comes with gevent event loop and the Tornado event loop, just choose your favorite weapon.

1.2 Remote Calls

To perform remote procedure calls you just need to connect two peers, and then, on your local peer instance, call a registered function with the right parameters. You will then receive the return value of the remotely executed function.

```
# server.py
import string

import gevent
import pseud
from pseud.utils import register_rpc


server = pseud.Server('remote')
server.bind('tcp://127.0.0.1:5555')

# register locally for this server only
server.register_rpc(string.lower)
# register globally for all rpc instances
register_rpc(string.upper)
```

```
server.start()
gevent.wait()

# client.py
import pseud


client = pseud.Client('remote')
client.connect('tcp://127.0.0.1:5555')

future1 = client.lower('ABC')
future2 = client.upper('def')

assert future1.get() == 'abc'
assert future2.get() == 'DEF'
```

1.2.1 Registration

Registration is a necessary step to control what callable you want to expose for remote peers.

Global

The *register_rpc* decorator from `pseud.utils` module must be used to register a callable for all workers of the current process.

```
from pseud.utils import regsiter_rpc


@register_rpc
def call_me():
     return 'Done'
```

Local

An RPC instance exposes its own *register_rpc* function, which is used to register a callable only for that same RPC instance.

```
def call_me():
    return 'Done'

server.register_rpc(call_me)
```

You can also instantiate a registry and give it to `pseud.utils.register_rpc`, and pass it as an init parameter in the RPC. It is more convenient to use register_rpc as a decorator

```
import pseud
from pseud.utils import register_rpc, create_local_registry

registry = create_local_registry('worker')

@register_rpc(registry=registry)
def call_me():
    return 'Done'
```

```
server = pseud.Server('worker', registry=registry)
```

Name it !

You can also decide to provide your own name (dotted name) to the callable

```
from pseud.utils import regsiter_rpc


@register_rpc('this.is.a.name')
def call_me():
     return 'Done'

client.this.is.a.name().get() == 'Done'
```

Server wants to make the client working

In order to let the server send jobs to its connected clients, the caller should know the identity of the specified client beforehand. How to get a list of currently connected clients is described in the *Heartbeating* section.

Given a client whose identity is `'client'`, with a registered function named `addition`, the following statement may be used to send work from the server to the client

```
# gevent process
server.send_to('client').addition(2, 4).get() == 6
```

1.3 Authentication

pseud allows you to build your own Authentication Backend. Your implementation must conform to its `Interface` defined in `pseud.interfaces.IAuthenticationBackend`

Also all your plugin must `adapts` `pseud.interfaces.IClient` or `pseud.interfaces.IServer` and being registered thanks to `pseud.utils.register_auth_backend()` decorator.

Implementing your own authentication backend can be used to support CURVE encryption. And also for more advanced use-case with external ID provider. That is your favorite web-framework or simple PAM, you name it.

You can start with the following snippet

```
@register_auth_backend
@zope.interface.implementer(IAuthenticationBackend)
@zope.component.adapter(IClient)
class MyAuthenticationBackend(object):
    """
    This implementation implements
    IAuthenticationBackend and adapts IClient
    """
    name = 'my_auth_backend'

    def __init__(self, rpc):
        self.rpc = rpc

    def stop(self):
        pass
    def configure(self):
        pass

    def handle_hello(self, *args):
        pass

    def handle_authenticated(self, message):
        pass

    def is_authenticated(self, peer_id):
        return True

    def save_last_work(self, message):
        pass

    def get_predicate_arguments(self, peer_id):
        return {}
```

In this example the name *'my_auth_backend'* will be used when instanciating your RPC endpoint.

```
client = pseud.Client('local', 'remote',
                      security_plugin='my_auth_backend')
```

Read *Protocol v1* for more explanation. Also in `pseud.auth` you will find examples that are used in tests.

1.4 Heartbeating

pseud allows you to build your own Heartbeat Backend. Your implementation must conform to its `Interface` defined in `pseud.interfaces.IHeartbeatBackend`

Also all your plugin must `adapts` `pseud.interfaces.IClient` or `pseud.interfaces.IServer` and being registered thanks to `pseud.utils.register_heartbeat_backend()` decorator.

Heartbeat backends aim to define your the policy you need regarding exclusion of disconnected peer, e.g.. after 3 heartbeat missed, you can decide to exclude peer from list of known connected peers.

Also, very important, thanks to heartbeat backends you can maintain an accurate list of currently connected clients and their ids. It is up to you to decide to store this list in memory (simple dict), or to use redis if you think the number of peers will be huge.

You can start with the following snippet

```
@register_heartbeat_backend
@zope.interface.implementer(IHeartbeatBackend)
@zope.component.adapter(IClient)
class MyHeartbeatBackend(object):
    name = 'my_heartbeat_backend'

    def __init__(self, rpc):
        self.rpc = rpc

    def handle_heartbeat(self, peer_id):
        pass

    def handle_timeout(self, peer_id):
        pass
    def configure(self):
        pass

    def stop(self):
        pass
```

In this example the name *'my_heartbeat_backend'* will be used when instanciating your RPC endpoint.

```
client = pseud.Client('local', 'remote',
                      heartbeat_plugin='my_heartbeat_backend')
```

Read *Protocol v1* for more explanation. Also in `pseud.heartbeat` you will find examples that are used in tests.

1.5 Job Routing

1.5.1 Predicates

During registration, user can associate a domain to the callable. Each domain will be linked to a specific Predicate with its own Policy. By default all rpc-callable are registered within *default* domain, that allow all callable to be called. In case of rejection, `pseud.interfaces.ServiceNotFoundError` exception will be raised.

You can of course define your own predicate and register some callable under restricted domain for instance.

```
@register_rpc(name='try_to_call_me')
def callme(*args, **kw):
    return 'small power'

@register_rpc(name='try_to_call_me',
              domain='restricted')
def callme_admin(*args, **kw):
    return 'great power'
```

In this example we have 2 callable registered with same name but with different domain. Assuming we a have a Authentication Backend that is able to return a user instance and from this user instance we can know if he is admin. then we can assume the following behaviour

```
# gevent client + user lambda

client.try_to_callme().get() == 'small power'
```

Then with user with admin rights

```
# gevent client + user admin

client.try_to_callme().get() == 'great power'
```

From this behaviour we can perform routing based on user permissions.

1.6 Protocol v1

pseud uses to transport its messages ØMQ with ROUTER sockets. the structure of every frames follow this specification.

ENVELOPE + PSEUD MESSAGE

1.6.1 ENVELOPE

The envelope belongs to ømq typology to route messages to right recipient. the are separated from pseud message with empty delimiter `''`. Basically the envelope will be

```
['peer_identity', '']
```

1.6.2 PSEUD MESSAGE

FRAME 0: *VERSION* of current protocol

```
utf-8 string 'v1'
```

FRAME 1: message uuid

```
bytes uuid4 or empty string for hearbeat messages
```

FRAME 2: message type

```
byte
```

FRAME 3: body

```
WORK, OK, ERROR and HELLO expect msgpack.
AUTHENTICATED, UNAUTHORIZED and HEARTBEAT expect utf-8 strings.
```

1.6.3 MESSAGE TYPES

WORK

```
'\x03'
```

the body content is a tuple of 3 items

1. dotted name of the rpc-callable
2. tuple of positional arguments
3. dict of keyword arguments

OK

```
'\x01'
```

ERROR

```
'\x10'
```

the body content is a tuple of 3 items

1. string of Exception class name e.g. 'AttributeError'
2. message of the exception
3. Remote traceback

UNAUTHORIZED

`'\x11'`

HELLO

`'\x02'`

the body content is a tuple of 2 items

1. login
2. password

AUTHENTICATED

`'\x04'`

HEARTBEAT

`'\x06'`

1.6.4 COMMUNICATION

1. client sends work to server and receive successful answer.

client	-> WORK	<- OK	server

2. client sends work to server and receive an error.

client	-> WORK	<- ERROR	server

3. server sends work to client and receive successful answer.

client	-> OK	<- WORK	server

4. client sends an heartbeat

client	-> HEARTBEAT	<-	server

5. server sends an heartbeat

client	->	<- HEARTBEAT	server

6. client send a job and server requires authentication

client	->	<-	server
	WORK		
		UNAUTHORIZED	
	HELLO		
		AUTHENTICATED	
	WORK		
		OK	

7. client send a job and server requires authentication but fails

client	->	<-	server
	WORK		
		UNAUTHORIZED	
	HELLO		
		UNAUTHORIZED	

1.7 `pseud.interfaces`

1.7.1 RPC-Related Interfaces

1.7.2 Plugins-Related Interfaces

1.7.3 Constants

`WORK`

`OK`

`ERROR`

`HELLO`

`UNAUTHORIZED`

`AUTHENTICATED`

`HEARTBEAT`

1.7.4 Exceptions

1.8 Changelog history

1.8.1 0.0.6 - Not Yet Released

1.8.2 0.0.5 - 2014/08/27

- Add python3.4 support for Tornado backend

1.8.3 0.0.4 - 2014/03/25

1.8.4 0.0.3 - 2014/02/24

- Add support of Aysnc RPC callables for Tornado
- Add support of datetime (tz aware) serializations by msgpack

1.8.5 0.0.2 - 2014/02/13

1.8.6 0.0.1 - 2014/01/27

- Scaffolding of the lib

CHAPTER 2

API Documentation

2.1 `pseud.auth`

2.2 `pseud.heartbeat`

2.3 `pseud.predicate`

2.4 `pseud.utils`

Indices and tables

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

3.1 Glossary

AUTHENTICATED Status member of pseud protocol

domain Apply to predicates for job routing

UNAUTHORIZED Status member of pseud protocol