Design and implementation of a RPC library in python

Rémi Audebert

2014-07-18
Introduction

Rémi Audebert

Design and implementation of a RPC library in python

Client

Service
Decorators
Signatures
More signatures

Conclusion
## Remote procedure call

<table>
<thead>
<tr>
<th>RPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Services with methods</td>
</tr>
<tr>
<td>- Clients</td>
</tr>
<tr>
<td>- Request from a client to a service</td>
</tr>
<tr>
<td>- Reply from a service to a client</td>
</tr>
</tbody>
</table>
Remote procedure call

RPC

- Services with methods
- Clients
- Request from a client to a service
- Reply from a service to a client

RPC system: Cellaserv2

- Based on TCP/IP
- Centralized server
- Uses protocol buffers
- More information: https://code.evolutek.org/cellaserv2
Register + Request + Reply

RPC Server (cellaserv)

Client

time Service

register("time")

request("time", "date", uuid, {})

request("time", "date", uuid, {})

reply("time", "date", uuid, "2014-07-18")

reply("time", "date", uuid, "2014-07-18")
The requirements of this library

**Technical**
- Python3
- No external libraries
- Handle Client and Service

**Usage**
- Easy to use
- Hard to misuse
- Fail gracefully
- Target users: mostly sleep deprived
The rush friendly requirement
Client
Design and implementation of a RPC library in python

Rémi Audebert

Introduction

Client

Service

Decorators

Signatures

More signatures

Conclusion

CellaservProxy

Code

```python
from cellaserv.proxy import CellaservProxy
client = CellaservProxy()
```

Configuration

- `CellaservProxy(host="example.org", port=4242)`
- Environment variables: `CS_HOST`, `CS_PORT`
- Configuration file: `/etc/conf.d/cellaserv`

Usage
Design and implementation of a RPC library in python

Rémi Audebert

Introduction

Client

Service

Decorators

Signatures

More signatures

Conclusion

CellaservProxy

Code

```
from cellaserv.proxy import CellaservProxy
client = CellaservProxy()
```

Configuration

- CellaservProxy(host="example.org", port=4242)
- Environment variables: CS_HOST, CS_PORT
- Configuration file: /etc/conf.d/cellaserv

Usage

```
now = client.time.date()
```
Usage

```python
client.time.date()
```

cellaserv/proxy.py

```python
class CellaservProxy(cellaserv.client.SynClient):
    ...
    def __getattr__(self, service_name):
        return ServiceProxy(self.conn, service_name)
```
Usage

```python
client.time.date()
```

cellaserv/proxy.py

```python
class CellaservProxy(cellaserv.client.SynClient):
    ...
    def __getattr__(self, "time"):
        return ServiceProxy(self.conn, "time")
```
client.time.date()

Figure 1: Execution
Design and implementation of a RPC library in python

Rémi Audebert

Introduction
Client
Service
Decorators
Signatures
More signatures

Conclusion
Decorators
Service’s design

Simple usage

```python
import time
from cellaserv.service import Service

class Time(Service):
    @Service.action
    def date(self):
        return time.time()

Time().run()
```
Python’s feature: decorators

### Simple usage

```python
>>> def log_usage(f):
...     def wrap(*args, **kwargs):
...         print("{{}} called".format(f))
...         return f(*args, **kwargs)
...     return wrap
...  
>>> my_len = log_usage(len)
>>> my_len([42])
<built-in function len> called
1
```

Note: this decorator should use `@functools.wraps(f)`. 
Advanced usage

```python
class Date(Service):
    @Service.action("heure")
    @Service.action("time_" + get_timezone())
    def time(self):
        return time.time()
```
What is the difference between:

**Name of the method is name of the function**

```python
@Service.action
def action1(self):
    pass
```

**Name of the method is given by the user**

```python
@Service.action("action_name")
def action2(self):
    pass
```
Function decorator: caution

Name of the method is name of the function

```
action1 = Service.action(action1)
```

Name of the method is given by the user

```
action2 = Service.action("action_name")(action2)
```
Design and implementation of a RPC library in python

Rémi Audebert

Introduction

Client

Service

Decorators

Signatures

More signatures

Conclusion

Python’s feature: decorators

class Service:
    @staticmethod
    def action(method_or_name):
        def _set_action(method, action):
            try:
                method._actions.append(action)
            except AttributeError:
                method._actions = [action]
        return method

        def _wrapper(method):
            return _set_action(method, method_or_name)

        if callable(method_or_name):
            return _set_action(method_or_name, method_or_name.__name__)
        else:
            return _wrapper
Signatures
We want to emit a warning when the user send a request with bad arguments.
Dealing with TypeError

We want to emit a warning when the user send a request with bad arguments.

**Bad prototype**

```python
>>> def f():
...     pass

>>> f(42)
TypeError: f() takes 0 positional arguments but 1 was given
```
Dealing with TypeError

We want to emit a warning when the user send a request with bad arguments.

**Bad prototype**

```python
>>> def f():
    ...    pass
>>> f(42)
TypeError: f() takes 0 positional arguments but 1 was given
```

**Bad service code**

```python
>>> def f():
    ...    1 + 'a'
>>> f()
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```
Report all exceptions to the user.
Checking signatures: quick and dirty solution

<table>
<thead>
<tr>
<th>Hack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the stack size.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bad prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;&gt; try:</td>
</tr>
<tr>
<td>... f(42)</td>
</tr>
<tr>
<td>... except:</td>
</tr>
<tr>
<td>... print(len(inspect.trace()))</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Internal error</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;&gt; try:</td>
</tr>
<tr>
<td>... f()</td>
</tr>
<tr>
<td>... except:</td>
</tr>
<tr>
<td>... print(len(inspect.trace()))</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
The signature object

- `inspect.signature(f)` (new in python3.3)

Code

```python
>>> def f(a, b):
...     pass
>>> sig = inspect.signature(f)
>>> print(sig)
(a, b)
>>> sig.parameters
mappingproxy(OrderedDict([("x", <Parameter at ... 'x'>), ("y", <Parameter at ... 'y'>)]))
```
Use the \texttt{bind()} method

\begin{verbatim}
>>> def f(a, b):
...    pass

>>> sig = inspect.signature(f)

>>> user_kwargs = {'a': 42, 'b': 1.2}

>>> sig.bind(**user_kwargs)
<inspect.BoundArguments object at ...>
\end{verbatim}
Use the `bind()` method

```python
>>> def f(a, b):
...     pass

>>> sig = inspect.signature(f)

>>> user_kwargs = {'a': 42, 'b': 1.2}

>>> sig.bind(**user_kwargs)
<inspect.BoundArguments object at ...>
```
Bind is smart

Advanced signatures

```python
>>> def f(a, *args, v=False, **kwargs):
...     pass
...

>>> sig = inspect.signature(f)

>>> user_args = ('x', 'y')

>>> user_kwargs = {'a': 42, 'v': True}

>>> sig.bind(*user_args, **user_kwargs)
<inspect.BoundArguments object at ...>

>>> sig.bind()
TypeError: 'a' parameter lacking default value
```
Bind is slow...

- 63 times slower than:

```python
try:
    f(**user_kwargs)
except TypeError as e:
    // use inspect.stack()
```

The EAFP coding style

_Easier to ask for forgiveness than permission._

- Assume the user is mostly right.
More signatures
Give the user all he needs

PEP-3107: Function Annotations (python3.0)

```python
>>> def is_safe(pos: '(x, y) 30mm radius') -> bool:
...    pass

>>> print(inspect.signature(is_safe))
(pos: '(x, y) 30mm radius') -> bool
```
Conclusion
More fun in this library

- Synchronous and Asynchronous clients
- Service’s dependencies
- Events
- Attributes over RPC
- Descriptors
- Identification of services
- Supports down to python3.1
- Manages user’s threads automatically
- Metaclass
- ...
Conclusion

This talk

- Code: http://code.evolutek.eu/python-cellaserv2
- Discuss: #evolutek<<@irc.rezosup.org

Contact

- IRC: halfr@irc.rezosup.org
- Mail: halfr@lse.epita.fr
- Twitter: @halfr