rpclib Documentation

Release

February 07, 2012

CONTENTS

Welcome to Rpclib documentation! We hope you find this text useful. Please do not hesitate to get in touch with us if you think you can help us improve Rpclib or its documentation one way or another. Contact information is below.

Have fun!

CHAPTER

ONE

GETTING STARTED

Warning! This is from rpclib's unstable development branch.

1.1 About

Rpclib aims to save the protocol implementers the hassle of implementing their own remote procedure call api and the application programmers the hassle of jumping through hoops just to expose their services using multiple protocols and transports.

Rpclib comes with the implementations of popular transport, protocol and interface document standards along with an easy-to-use API that lets you build on existing functionality.

Rpclib currently supports the WSDL 1.1 interface description standard, along with SOAP 1.1 and the rest-minus-the-verbs HttpRpc protocols which can be transported via HTTP or ZeroMQ. The transports can be used in a both client or server setting.

The following are the primary sources of information about rpclib:

- The latest documentation for Rpclib can be found at: http://arskom.github.com/rpclib
- The official source code repository is at: https://github.com/arskom/rpclib
- The official rpclib discussion forum is at: http://mail.python.org/mailman/listinfo/soap
- You can download Rpclib packages from github or pypi.

Rpclib is a generalized version of a soap processing library known as soaplib. The following legacy versions of soaplib are also available in the source repository at github as branches.

- Soaplib-0.8 branch: http://github.com/arskom/rpclib/tree/soaplib-0_8
- Soaplib-1.0 branch: http://github.com/arskom/rpclib/tree/soaplib-1_0
- Soaplib-2.0 was never released as a stable package, but the branch is still available: http://github.com/arskom/rpclib/tree/soaplib-2_0

1.2 Requirements

Rpclib reportedly runs on any version of Python from 2.4 through 2.7. We're also looking for volunteers to test Python 3.x.

While the aim is to have no requirements besides the standard Python library for the Rpclib core, the following packages are needed if you want to run any Rpclib service at all:

- lxml
- pytz

both of which are available through `easy_install`.

Additionally the following software packages are needed for various subsystems that Rpclib supports:

- SQLAlchemy for `rpclib.model.table.TableModel`.
- pyzmq for `rpclib.client.zeromq.ZeroMQClient` and `rpclib.server.zeromq.ZeroMQServer`.
- A Wsgi server of your choice to wrap `rpclib.server.wsgi.WsgiApplication`.

1.3 Installing

You can get rpclib via pypi:

```
easy_install rpclib
```

or you can clone from github:

```
git clone git://github.com/arskom/rpclib.git
```

or get the source distribution from one of the download sites and unpack it.

To install from source distribution, you should run its setup script as usual:

```
python setup.py install
```

To run the non-interop tests use:

```
python setup.py test
```

And if you want to make any changes to the rpclib code, it's more comfortable to use:

```
python setup.py develop
```

1.4 Contributing

The main developers of rpclib lurk in the official soap implementors forum in python.org, here. That's mostly because rpclib is the continuation of soaplib, but also because soap is an important part of rpclib.

If you wish to contribute to rpclib's development, create a personal fork on GitHub. When you are ready to push to the mainstream repository, submit a pull request to bring your work to the attention of the core committers. They will respond to review your patch and act accordingly.

To save both parties time, make sure the existing tests pass. If you are adding new functionality or fixing a bug, please have the accompanying test. This will both help us increase test coverage and insure your use-case is immune to feature code changes. You could also summarize in one or two lines how your work will affect the life of rpclib users in the CHANGELOG file.

Please follow the PEP 8 style guidelines for both source code and docstrings.

We could also use help with the docs, which are built from restructured text using Sphinx.

Regular contributors may be invited to join as a core rpclib committer on GitHub. Even if this gives the core committers the power to commit directly to the core repository, we highly value code reviews and expect every significant change to be committed via pull requests.

1.4.1 Submitting Pull Requests

Github's pull-request feature is awesome, but there's a subtlety that's not totally obvious for newcomers: If you continue working on the branch that you used to submit a pull request, your commits will "pollute" the pull request until it gets merged. This is not a bug, but a feature – it gives you the ability to address reviewers' concerns without creating pull requests over and over again. So, if you intend to work on other parts of rpclib after submitting a pull request, please do move your work to its own branch and never submit a pull request from your master branch. This will give you the freedom to continue working on rpclib while waiting for your pull request to be reviewed.

CHAPTER

TWO

LIBRARY DOCUMENTATION

2.1 Rpclib Manual

The API reference is not the best resource when learning use a new software library. Here, you will find a collection of documents that show you various aspects of Rpclib by examples.

2.1.1 High-Level Introduction to Rpclib

The impatient can just jump to the *In a nutshell* section.

Concepts

The following is a quick introduction to the Rpclib way of naming things:

- **Protocols:** Protocols define the rules for transmission of structured data. They are children of `rpclib.protocol._base.ProtocolBase` class.

  For example: Rpclib implements a subset of the Soap 1.1 protocol.

- **Transports:** Transports, also protocols themselves, encapsulate protocol data in their free-form data sections. They are children of either `rpclib.client._base.ClientBase` or `rpclib.server._base.ServerBase` classes.

  For example, Http is used as a transport for Soap, by tucking a Soap message in the Http byte-stream part of a Http POST request. The same Http is exposed as a "protocol" using the `rpclib.protocol.http.HttpRpc` class. One could use Soap as a transport by tucking a protocol message in its base64-encoded ByteArray container.

  Transports appear under two packages in Rpclib source code: `rpclib.client` and `rpclib.server`.

- **Models:** Models are used to define schemas. They are mere magic cookies that contain very little amount of serialization logic. They are children of `rpclib.model._base.ModelBase` class.

  Types like `String`, `Integer` or `ByteArray` are all models. They reside in the `rpclib.model` package,

- **Interfaces:** Interface documents provide a machine-readable description of the input the services expect and output they emit. It thus serves a roughly similar purpose as a method signature in a programming language. They are children of `rpclib.interface._base.InterfaceBase` class.

  `rpclib.interface` package is where you can find them.

- **Serializers:** Serializers are currently not distinguished in rpclib code. They are the protocol-specific representations of a serialized python object.

  They can be anything between an lxml.etree.Element instance to a gzipped byte stream. Apis around pickle, simplejson, YaML and the like also fall in this category.

How your code is wrapped

While the information in the previous section gave you an idea about how Rpclib code is organized, this section is supposed to give you an idea about how *you* should organize your code using the tools provided by Rpclib.

A typical Rpclib user will just write methods that will be exposed as remote procedure calls to the outside world. The following is used to wrap that code:

- **User Methods: User methods are the code that you wrote and decided to use** rpclib to expose to the outside world.
- **Decorators:** the `@rpc` and `@srpc` decorators from `rpclib.decorator` module are used to flag methods that will be exposed to the outside world by marking their input and output types, as well as other properties.
- **Service Definition:** The `rpclib.service.ServiceBase` is an abstract base class for service definitions, which are the smallest exposable service unit in rpclib. You can use one service class per method definition or you can use, say, a service class for read-only or read/write services or you can cram everything into one service class, it's up to you.

  Service definition classes are suitable for grouping services that have common properties like logging, transaction management and security policy. It's often a good idea to base your service definitions on your own ServiceBase children instead of using the vanilla `ServiceBase` class offered by Rpclib.
- **Application:** The `rpclib.application.Application` class is what ties services, interfaces and protocols together, ready to be wrapped by a transport. It also lets you define events and hooks like ServiceBase does, so you can do more general, application-wide customizations like exception management.

  Note: You may know that rpclib is a generalized version of a soap library. So inevitably, some artifacts of the Soap world creep in from here and there.

  One of those artifacts is xml namespaces. There are varying opinions about the usefulness of the concept of the namespace in the Xml standard, but we generally think it to be A Nice Thing, so we chose to keep it around.

  When instantiating the `rpclib.application.Application` class, you should also give it a targetNamespace (the `tns` argument to its constructor) string and an optional application name (the `name` argument to the `Application` constructor), which are used to generally distinguish your application from other applications. While it's conventionally the URL and the name of the class of your application, you can put `tns="Hogwarts", name="Harry"` there and just be done with it.

  Every object in the Rpclib world has a name and belongs to a namespace. Public functions (and the implicit `rpclib.model.complex.ComplexModel` children that are created for the input and output types of the functions you defined) are forced to be in the Application namespace, and have whatever you give them as public name in the `rpclib.decorator.srpc()` decorator. Rpclib-defined types generally belong to a pre-defined namespace by default. User-defined objects have the module name as namespace string and class name as name string by default.

In case you'd like to read on how *exactly* your code is wrapped, you can refer to the relevant part in the *Implementing Transports and Protocols* section.

In a nutshell

Your code is inside @rpc-wrapped methods in ServiceBase children, which are grouped in an Application instance, which communicates with the outside world using given interface and protocol classes, and which is finally wrapped by a client or server transport that takes the responsibility of moving the bits around.

What's next?

Now that you have a general idea about how Rpclib is supposed to work, let's get coding. You can start by *Hello World* tutorial right now.

2.1.2 Hello World

This example uses the stock simple wsgi webserver to deploy this service. You should probably use a full-fledged server when deploying your service for production purposes.

Defining an Rpclib Service

Here we introduce the fundamental mechanisms the rpclib offers to expose your services.

The simpler version of this example is available here: http://github.com/arskom/rpclib/blob/master/examples/helloworld_soap.py

```
import logging

from rpclib.application import Application
from rpclib.decorator import srpc
from rpclib.interface.wsdl import Wsdl11
from rpclib.protocol.soap import Soap11
from rpclib.service import ServiceBase
from rpclib.model.complex import Iterable
from rpclib.model.primitive import Integer
from rpclib.model.primitive import String
from rpclib.server.wsgi import WsgiApplication

class HelloWorldService(ServiceBase):
    @srpc(String, Integer, _returns=Iterable(String))
    def say_hello(name, times):
        '''
        Docstrings for service methods appear as documentation in the wsdl
        <b>what fun</b>
        @param name the name to say hello to
        @param the number of times to say hello
        @return the completed array
        '''

        for i in xrange(times):
            yield 'Hello, %s' % name

if __name__=='__main__':
    try:
        from wsgiref.simple_server import make_server
    except ImportError:
        print "Error: example server code requires Python >= 2.5"

    logging.basicConfig(level=logging.DEBUG)
```

```
    logging.getLogger('rpclib.protocol.xml').setLevel(logging.DEBUG)

    application = Application([HelloWorldService], 'rpclib.examples.hello.soap',
                interface=Wsdl11(), in_protocol=Soap11(), out_protocol=Soap11())

    server = make_server('127.0.0.1', 7789, WsgiApplication(application))

    print "listening to http://127.0.0.1:7789"
    print "wsdl is at: http://localhost:7789/?wsdl"

    server.serve_forever()
```

Dissecting this example: Application is the glue between one or more service definitions, interface and protocol choices.

```
from rpclib.application import Application
```

The srpc decorator exposes methods as remote procedure calls and declares the data types it accepts and returns. The 's' prefix is short for static. It means no implicit argument will be passed to the function. In the @rpc case, the function gets a rpclib.MethodContext instance as first argument.

```
from rpclib.decorator import srpc
```

We are going to expose the service definitions using the Wsdl 1.1 document standard. The methods will use Soap 1.1 protocol to communicate with the outside world. They're instantiated and passed to the Application constructor. You need to pass fresh instances to each application instance.

```
from rpclib.interface.wsdl import Wsdl11
from rpclib.protocol.soap import Soap11
```

For the sake of this tutorial, we are going to use HttpRpc as well. It's a rest-like protocol, but it doesn't care about HTTP verbs (yet).

```
from rpclib.protocol.http import HttpRpc
```

The HttpRpc serializer does not support complex types. So we will use the XmlObject serializer as the out_protocol to prevent the clients from dealing with Soap cruft.

```
from rpclib.protocol.http import XmlObject
```

ServiceBase is the base class for all service definitions.

```
from rpclib.service import ServiceBase
```

The names of the needed types for implementing this service should be self-explanatory.

```
from rpclib.model.complex import Iterable
from rpclib.model.primitive import Integer
from rpclib.model.primitive import String
```

Our server is going to use HTTP as transport, so we import the WsgiApplication from the server.wsgi module. It's going to wrap the application instance.

```
from rpclib.server.wsgi import WsgiApplication
```

We start by defining our service. The class name will be made public in the wsdl document unless explicitly overridden with *__service_name__* class attribute.

```
class HelloWorldService(ServiceBase):
```

The srpc decorator flags each method as a remote procedure call and defines the types and order of the soap parameters, as well as the type of the return value. This method takes in a string and an integer and returns an iterable of strings, just like that:

```
@srpc(String, Integer, _returns=Iterable(String))
```

The method itself has nothing special about it whatsoever. All input variables and return types are standard python objects:

```
def say_hello(name, times):
    for i in xrange(times):
        yield 'Hello, %s' % name
```

When returning an iterable, you can use any type of python iterable. Here, we chose to use generators.

Deploying the service using SOAP

Now that we have defined our service, we are ready to share it with the outside world.

We are going to use the ubiquitious Http protocol as a transport, using a Wsgi-compliant http server. This example uses Python's stock simple wsgi web server. Rpclib has been tested with several other web servers. Any WSGI-compliant server should work.

This is the required import:

```
if __name__=='__main__':
    from wsgiref.simple_server import make_server
```

Here, we configure the python logger to show debugging output. We have to specifically enable the debug output from the soap handler. That's because the xml formatting code is run only when explicitly enabled for performance reasons.

```
logging.basicConfig(level=logging.DEBUG)
logging.getLogger('rpclib.protocol.xml').setLevel(logging.DEBUG)
```

We glue the service definition, interface document and input and output protocols under the targetNamespace 'rpclib.examples.hello.soap':

```
application = Application([HelloWorldService], 'rpclib.examples.hello.soap',
            interface=Wsdl11(), in_protocol=Soap11(), out_protocol=Soap11())
```

We then wrap the rpclib application with its wsgi wrapper:

```
wsgi_app = WsgiApplication(application)
```

The above two lines can be replaced with an easier-to-use function that covers this common use case:

```
from rpclib.util.simple import wsgi_soap_application
wsgi_app = wsgi_soap_application([HelloWorldService], 'rpclib.examples.hello.soap')
```

We now register the WSGI application as the handler to the wsgi server, and run the http server:

```
server = make_server('127.0.0.1', 7789, wsgi_app)

print "listening to http://127.0.0.1:7789"
print "wsdl is at: http://localhost:7789/?wsdl"

server.serve_forever()
```

Note:

- **Django users:** See this gist for a django wrapper example: https://gist.github.com/1242760
- **Twisted users:** See the this example that illustrates deploying an Rpclib application using twisted: http://github.com/arskom/rpclib/blob/master/examples/helloworld_soap_twisted.py

---

You can test your service using suds. Suds is a separate project for building pure-python soap clients. To learn more visit the project's page: https://fedorahosted.org/suds/. You can simply install it using `easy_install suds`.

So here's how you can use suds to test your new rpclib service:

```
from suds.client import Client
hello_client = Client('http://localhost:7789/?wsdl')
result = hello_client.service.say_hello("Dave", 5)
print result
```

The script's output would be as follows:

```
(stringArray){
    string[] =
        "Hello, Dave",
        "Hello, Dave",
        "Hello, Dave",
        "Hello, Dave",
        "Hello, Dave",
    }
```

Deploying service using HttpRpc

This example is available here: http://github.com/arskom/rpclib/blob/master/examples/helloworld_http.py.

The only difference between the SOAP and the HTTP version is the application instantiation line:

```
application = Application([HelloWorldService], 'rpclib.examples.hello.http',
        interface=Wsdl11(), in_protocol=HttpRpc(), out_protocol=XmlObject())
```

We still want to keep Xml as the output protocol as the HttpRpc protocol is not able to handle complex types.

Here's how you can test your service using wget:

```
wget "http://localhost:7789/say_hello?times=5&name=Dave" -qO -
```

If you have HtmlTidy installed, you can use this command to get a more readable output.

```
wget "http://localhost:7789/say_hello?times=5&name=Dave" -qO - | tidy -xml -indent
```

The command's output would be as follows:

```
<?xml version='1.0' encoding='utf8'?>
<ns1:say_helloResponse xmlns:ns1="rpclib.examples.hello.http"
xmlns:ns0="http://schemas.xmlsoap.org/soap/envelope/">
  <ns1:say_helloResult>
    <ns1:string>Hello, Dave</ns1:string>
    <ns1:string>Hello, Dave</ns1:string>
    <ns1:string>Hello, Dave</ns1:string>
    <ns1:string>Hello, Dave</ns1:string>
    <ns1:string>Hello, Dave</ns1:string>
  </ns1:say_helloResult>
</ns1:say_helloResponse>
```

What's next?

See the *User Manager* tutorial that will walk you through defining complex objects and using events.

2.1.3 User Manager

This tutorial builds on the *Hello World* tutorial. If you haven't done so, we recommended you to read it first.

In this tutorial, we will talk about:

- Defining complex types.
- Customizing types.
- Defining events.

The following is an simple example using complex, nested data. It's available here: http://github.com/arskom/rpclib/blob/master/examples/user_manager/server_basic.py

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.getLogger('rpclib.protocol.xml').setLevel(logging.DEBUG)

from rpclib.application import Application
from rpclib.decorator import rpc
from rpclib.interface.wsdl import Wsdl11
from rpclib.protocol.soap import Soap11
from rpclib.model.primitive import String
from rpclib.model.primitive import Integer
from rpclib.model.complex import Array
from rpclib.model.complex import Iterable
from rpclib.model.complex import ComplexModel
from rpclib.server.wsgi import WsgiApplication
from rpclib.service import ServiceBase

_user_database = {}
_user_id_seq = 1

class Permission(ComplexModel):
    __namespace__ = 'rpclib.examples.user_manager'

    application = String
    operation = String

class User(ComplexModel):
    __namespace__ = 'rpclib.examples.user_manager'

    user_id = Integer
    user_name = String
    first_name = String
    last_name = String
    permissions = Array(Permission)

class UserManagerService(ServiceBase):
    @rpc(User, _returns=Integer)
    def add_user(ctx, user):
        user.user_id = ctx.udc.get_next_user_id()
        ctx.udc.users[user.user_id] = user
```

```
        return user.user_id

    @rpc(Integer, _returns=User)
    def get_user(ctx, user_id):
        return ctx.udc.users[user_id]

    @rpc(User)
    def set_user(ctx, user):
        ctx.udc.users[user.user_id] = user

    @rpc(Integer)
    def del_user(ctx, user_id):
        del ctx.udc.users[user_id]

    @rpc(_returns=Iterable(User))
    def get_all_users(ctx):
        return ctx.udc.users.itervalues()

application = Application([UserManagerService], 'rpclib.examples.user_manager',
            interface=Wsdl11(), in_protocol=Soap11(), out_protocol=Soap11())

def _on_method_call(ctx):
    ctx.udc = UserDefinedContext()

application.event_manager.add_listener('method_call', _on_method_call)

class UserDefinedContext(object):
    def __init__(self):
        self.users = _user_database

    @staticmethod
    def get_next_user_id():
        global _user_id_seq

        _user_id_seq += 1

        return _user_id_seq

if __name__=='__main__':
    try:
        from wsgiref.simple_server import make_server
    except ImportError:
        print "Error: example server code requires Python >= 2.5"

    server = make_server('127.0.0.1', 7789, WsgiApplication(application))

    print "listening to http://127.0.0.1:7789"
    print "wsdl is at: http://localhost:7789/?wsdl"

    server.serve_forever()
```

Juping into what's new: Rpclib uses `ComplexModel` as a general type that when extended will produce complex serializable types that can be used in a public service. The `Permission` class is a fairly simple class with just two members:

```
class Permission(ComplexModel):
    application = String
    feature = String
```

Let's also look at the `User` class:

```
class User(ComplexModel):
    user_id = Integer
    username = String
    firstname = String
    lastname = String
```

Nothing new so far.

Below, you can see that the `email` member which has a regular expression restriction defined. The String type accepts other restrictions, please refer to the `rpclib.model.primitive.String` documentation for more information:

```
email = String(pattern=r'\b[a-z0-9._%+-]+@[a-z0-9.-]+\.[A-Z]{2,4}\b')
```

The `permissions` attribute is an array, whose native type is a `list` of `Permission` objects.

```
permissions = Array(Permission)
```

The following is deserialized as a generator, but looks the same from the protocol and interface points of view:

```
permissions = Iterable(Permission)
```

The following is deserialized as a list of `Permission` objects, just like with the `Array` example, but is shown and serialized differently in Wsdl and Soap representations.

```
permissions = Permission.customize(max_occurs='unbounded')
```

Here, we need to use the `rpclib.model._base.ModelBase.customize()` call because calling a `ComplexModel` child instantiates that class, whereas calling a `SimpleModel` child returns a duplicate of that class. The `customize` function just sets given arguments as class attributes to `cls.Attributes` class. You can refer to the documentation of each class to see which member of the `Attributes` class is used for the given object.

Here, we define a function to be called for every method call. It instantiates the `UserDefinedContext` class and sets it to the context object's `udc` attribute, which is in fact short for 'user defined context'.

```
def _on_method_call(ctx):
    ctx.udc = UserDefinedContext()
```

We register it to the application's 'method_call' handler.

```
application.event_manager.add_listener('method_call', _on_method_call)
```

Note that registering it to the service definition's event manager would have the same effect:

```
UserManagerService.event_manager.add_listener('method_call', _on_method_call)
```

Here, we define the UserDefinedContext object. It's just a regular python class with no specific api it should adhere to, other than your own.

```
class UserDefinedContext(object):
    def __init__(self):
        self.users = _user_database

    @staticmethod
    def get_next_user_id():
        global _user_id_seq

        _user_id_seq += 1

        return _user_id_seq
```

Such custom objects could be used to manage everything from transactions to logging or to performance measurements. You can have a look at the events.py example in the examples directory in the source distribution for an example on using events to measure method performance)

What's next?

This tutorial walks you through what you need to know to expose basic services. You can read the *SQLAlchemy Integration* document where the `rpclib.model.table.TableModel` class and its helpers are introduced. You can also have look at the *Working with RPC Metadata* section where service metadata management apis are introduced.

Otherwise, please refer to the rest of the documentation or the mailing list if you have further questions.

2.1.4 SQLAlchemy Integration

This tutorial builds on the *User Manager* tutorial. If you haven't done so, we recommended you to read it first.

In this tutorial, we talk about using Rpclib tools that make it easy to deal with database-related operations. We will show how to integrate SQLAlchemy and Rpclib object definitions, and how to do painless transaction management using Rpclib events.

The full example is available here: http://github.com/arskom/rpclib/blob/master/examples/user_manager/server_sqlalchemy.py

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.getLogger('rpclib.protocol.xml').setLevel(logging.DEBUG)
logging.getLogger('sqlalchemy.engine.base.Engine').setLevel(logging.DEBUG)

import sqlalchemy

from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

from sqlalchemy import MetaData
from sqlalchemy import Column

from rpclib.application import Application
from rpclib.decorator import rpc
from rpclib.interface.wsdl import Wsdl11
from rpclib.protocol.soap import Soap11
from rpclib.model.complex import Iterable
from rpclib.model.primitive import Integer
from rpclib.model.table import TableSerializer
from rpclib.server.wsgi import WsgiApplication
from rpclib.service import ServiceBase

_user_database = create_engine('sqlite:///:memory:')
metadata = MetaData(bind=_user_database)
DeclarativeBase = declarative_base(metadata=metadata)
Session = sessionmaker(bind=_user_database)

class User(TableSerializer, DeclarativeBase):
    __namespace__ = 'rpclib.examples.user_manager'
    __tablename__ = 'rpclib_user'

    user_id = Column(sqlalchemy.Integer, primary_key=True)
    user_name = Column(sqlalchemy.String(256))
```

```
    first_name = Column(sqlalchemy.String(256))
    last_name = Column(sqlalchemy.String(256))

# this is the same as the above user object. Use this method of declaring
# objects for tables that have to be defined elsewhere.
class AlternativeUser(TableSerializer, DeclarativeBase):
    __namespace__ = 'rpclib.examples.user_manager'
    __table__ = User.__table__

class UserManagerService(ServiceBase):
    @rpc(User, _returns=Integer)
    def add_user(ctx, user):
        ctx.udc.session.add(user)
        ctx.udc.session.flush()

        return user.user_id

    @rpc(Integer, _returns=User)
    def get_user(ctx, user_id):
        return ctx.udc.session.query(User).filter_by(user_id=user_id).one()

    @rpc(User)
    def set_user(ctx, user):
        ctx.udc.session.merge(user)

    @rpc(Integer)
    def del_user(ctx, user_id):
        ctx.udc.session.query(User).filter_by(user_id=user_id).delete()

    @rpc(_returns=Iterable(AlternativeUser))
    def get_all_user(ctx):
        return ctx.udc.session.query(User)

class UserDefinedContext(object):
    def __init__(self):
        self.session = Session()

    def __del__(self):
        self.session.close()

def _on_method_call(ctx):
    ctx.udc = UserDefinedContext()

def _on_method_return_object(ctx):
    # we don't do this in UserDefinedContext.__del__ simply to be able to alert
    # the client in case the commit fails.
    ctx.udc.session.commit()

application = Application([UserManagerService], 'rpclib.examples.user_manager',
            interface=Wsdl11(), in_protocol=Soap11(), out_protocol=Soap11())

application.event_manager.add_listener('method_call', _on_method_call)
application.event_manager.add_listener('method_return_object', _on_method_return_object)

if __name__=='__main__':
    try:
        from wsgiref.simple_server import make_server
    except ImportError:
```

```
        print "Error: example server code requires Python >= 2.5"

    wsgi_app = WsgiApplication(application)
    server = make_server('127.0.0.1', 7789, wsgi_app)

    metadata.create_all()
    print "listening to http://127.0.0.1:7789"
    print "wsdl is at: http://localhost:7789/?wsdl"

    server.serve_forever()
```

Again, focusing on what's different from previous *User Manager* example:

```
class User(TableModel, DeclarativeBase):
    __namespace__ = 'rpclib.examples.user_manager'
    __tablename__ = 'rpclib_user'

    user_id = Column(sqlalchemy.Integer, primary_key=True)
    user_name = Column(sqlalchemy.String(256))
    first_name = Column(sqlalchemy.String(256))
    last_name = Column(sqlalchemy.String(256))
```

Defined this way, SQLAlchemy objects are regular Rpclib objects that can be used anywhere the regular Rpclib types go. The definition for the *User* object is quite similar to vanilla SQLAlchemy declarative syntax, save for two elements:

1. The object also bases on `rpclib.model.table.TableModel`, which bridges SQLAlchemy and Rpclib types.
2. It has a namespace declaration, which is just so the service looks good on wsdl.

The SQLAlchemy integration is far from perfect at the moment:

- SQL constraints are not reflected to the interface document.
- It's not possible to define additional schema constraints.
- Object attributes defined by mechanisms other than Column and a limited form of *relationship* (no string arguments) are not supported.

If you need any of the above features, you need to separate the rpclib and sqlalchemy object definitions.

Rpclib makes it easy to an extent with the following syntax:

```
class AlternativeUser(TableSerializer, DeclarativeBase):
    __namespace__ = 'rpclib.examples.user_manager'
    __table__ = User.__table__
```

Here, The AlternativeUser object is automatically populated using columns from the table definition.

The context object is also a little bit different – we start a transaction for every call in the constructor of the UserDefinedContext object, and close it in its destructor:

```
class UserDefinedContext(object):
    def __init__(self):
        self.session = Session()

    def __del__(self):
        self.session.close()
```

We implement an event handler that instantiates the UserDefinedContext object for every method call:

```
def _on_method_call(ctx):
    ctx.udc = UserDefinedContext()
```

We also implement an event handler that commits the transaction once the method call is complete.

```
def _on_method_return_object(ctx):
    ctx.udc.session.commit()
```

We register those handlers to the application's 'method_call' handler:

```
application.event_manager.add_listener('method_call', _on_method_call)
application.event_manager.add_listener('method_return_object', _on_method_return_object)
```

Note that the `method_return_object` event is only fired when the method call completes without throwing any exceptions.

What's next?

This tutorial walks you through most of what you need to know to expose your services. You can read the *Working with RPC Metadata* section where service metadata management apis are introduced.

Otherwise, you can refer to the reference of the documentation or the mailing list if you have further questions.

2.1.5 Input Validation

TODO

2.1.6 Working with RPC Metadata

This section builds on *User Manager* section. If you haven't done so, we recommended you to read it first.

In most of the real-world scenarios, an rpc request comes with additional baggage like authentication headers, routing history, and similar information. Rpclib comes with rich mechanisms that lets you deal with both protocol and transport metadata.

At the protocol level, the input and the output of the rpc function itself are kept in `ctx.in_object` and `ctx.out_object` attributes of the `rpclib.MethodContext` whereas the protocol metadata reside in `ctx.in_header` and `ctx.out_header` attributes.

You can set values to the header attributes in the function bodies or events. You just need to heed the order the events are fired, so that you don't overwrite data.

If you want to use headers in a function, you must denote it either in the decorator or the `rpclib.service.ServiceBase` child that you use to expose your functions.

A full example using most of the available metadata functionality is available here: https://github.com/plq/rpclib/blob/master/examples/authenticate/server_soap.py

Protocol Headers

The protocol headers are available in `ctx.in_header` and `ctx.out_header` objects. You should set the `ctx.out_header` to the native value of the declared type.

Header objects are defined just like any other object:

```
class RequestHeader(ComplexModel):
    user_name = Mandatory.String
    session_id = Mandatory.String
```

They can be integrated to the rpc definition either by denoting it in the service definition:

```
preferences_db = {}

class UserService(ServiceBase):
    __tns__ = 'rpclib.examples.authentication'
    __in_header__ = RequestHeader

    @rpc(_returns=Preferences)
    def get_preferences(ctx):
        retval = preferences_db[ctx.in_header.user_name]

        return retval
```

Or in the decorator:

```
@rpc(_in_header=RequestHeader, _returns=Preferences)
```

It's generally a better idea to set the header types in the ServiceBase child as it's likely that a lot of methods will use it. This will avoid cluttering the service definition with header declarations. The header declaration in the decorator will overwrite the one in the service definition.

Among the protocols that support headers, only Soap is supported.

Transport Headers

There is currently no general transport header api – transport-specific apis should be used for setting headers.

`rpclib.server.wsgi.WsgiApplication`: The `ctx.transport.resp_headers` attribute is a dict made of header/value pairs, both strings.

Exceptions

The base class for public exceptions in rpclib is `rpclib.model.fault.Fault`. The Fault object adheres to the SOAP 1.1 Fault definition, which has three main attributes:

1. `faultcode`: is a dot-delimited string whose first part is either 'Client' or 'Server'. Just like HTTP 4xx and 5xx codes, 'Client' indicates that something was wrong with the input, and 'Server' indicates something went wrong during the processing of the otherwise legitimate request.

   Protocol implementors should heed the values in `faultcode` to set proper return codes in the protocol level when necessary. E.g. HttpRpc protocol will return a HTTP 404 error when a `rpclib.error.ResourceNotFound` is raised, and a general HTTP 400 when the `faultcode` starts with 'Client.'.

2. `faultstring`: is the human-readable explanation of the exception.

3. `detail`: is the additional information as a valid xml document.

Here's how you define your own public exceptions:

```
class PublicKeyError(Fault):
    __type_name__ = 'KeyError'
    __namespace__ = 'rpclib.examples.authentication'
```

```
    def __init__(self, value):
        Fault.__init__(self,
                faultcode='Client.KeyError',
                faultstring='Value %r not found' % value
            )
```

Let's modify the python dict to throw our own exception class:

```
class RpclibDict(dict):
    def __getitem__(self, key):
        try:
            return dict.__getitem__(self, key)
        except KeyError:
            raise PublicKeyError(key)
```

We can now modify the decorator to expose the exception this service can throw:

```
preferences_db = RpclibDict()

class UserService(ServiceBase):
    __tns__ = 'rpclib.examples.authentication'
    __in_header__ = RequestHeader

    @rpc(_throws=PublicKeyError, _returns=Preferences)
    def get_preferences(ctx):
        retval = preferences_db[ctx.in_header.user_name]

        return retval
```

While this is not really necessary in the world of the dynamic languages, it'd still be nice to specify the exceptions your service can throw in the interface document. Plus, intefacing with your services will just feel more natural with languages like Java where exceptions are kept on a short leash.

What's next?

With this document, you know most of what rpclib has to offer for application programmers. You can refer to the *Implementing Transports and Protocols* section if you want to implement your own transports and protocols.

Otherwise, please refer to the rest of the documentation or the mailing list if you have further questions.

2.1.7 Implementing Transports and Protocols

Some preliminary information would be handy before delving into details:

How Exactly is User Code Wrapped?

Here's what happens when a request arrives to an Rpclib server:

The server transport decides whether this is a simple interface document request or a remote procedure call request. Every transport has its own way of dealing with this.

If the incoming request was for the interface document, it's easy: The interface document needs to be generated and returned as a nice chunk of string to the client. The server transport first calls `rpclib.interface._base.InterfaceBase.build_interface_document()` which builds and caches the document and later calls the `rpclib.interface._base.InterfaceBase.get_interface_document()` that returns the cached document.

If it was an RPC request, here's what happens:

1. The server must set the `ctx.in_string` attribute to an iterable of strings. This will contain the incoming byte stream.
2. The server calls the `rpclib.server._base.ServerBase.get_in_object` function from its parent class.
3. The server then calls the `create_in_document`, `decompose_incoming_envelope` and `deserialize` functions from the protocol class in the `in_protocol` attribute. The first call parses incoming stream to the protocol serializer's internal representation. This is then split to header and body parts by the second call and deserialized to the native python representation by the third call.
4. Once the protocol performs its voodoo, the server calls `get_out_object` which in turn calls the `rpclib.application.Application.process_request()` function.
5. The `process_request` function fires relevant events and calls the `rpclib.application.Application.call_wrapper()` function. This function is overridable by user, but the overriding function must call the one in `rpclib.application.Application`.
6. The `call_wrapper` function in turn calls the `rpclib.service.ServiceBase.call_wrapper()` function, which has has the same requirements.
7. The `rpclib.service.ServiceBase.call_wrapper()` finally calls the user function, and the value is returned to `process_request` call, which sets the return value to `ctx.out_object`.
8. The server object now calls the `get_out_string` function to put the response as an iterable of strings in `ctx.out_string`. The `get_out_string` function calls the `serialize` and `create_out_string` functions of the protocol class.
9. The server pushes the stream from ctx.out_string back to the client.

The same logic applies to client transports, in reverse.

So if you want to implement a new transport or protocol, all you need to do is to subclass the relevant base class and implement the missing methods.

A Transport Example: A DB-Backed Fan-Out Queue

Here's the source code in one file: https://github.com/arskom/rpclib/blob/master/examples/queue.py

The following block of code is SQLAlchemy boilerplate for creating the database and other related machinery. Under normal conditions, you should pass the sqlalchemy url to the Producer and Consumer instances instead of the connection object itself, but here as we deal with an in-memory database, global variable ugliness is just a nicer way to pass database handles.

```
db = create_engine('sqlite:///:memory:')
metadata = MetaData(bind=db)
DeclarativeBase = declarative_base(metadata=metadata)
```

This is the table where queued messages are stored. Note that it's a vanilla SQLAlchemy object:

```
class TaskQueue(DeclarativeBase):
    __tablename__ = 'task_queue'

    id = Column(sqlalchemy.Integer, primary_key=True)
    data = Column(sqlalchemy.LargeBinary, nullable=False)
```

This is the table where the task id of the last processed task for each worker is stored. Workers are identified by an integer.

```
class WorkerStatus(DeclarativeBase):
    __tablename__ = 'worker_status'

    worker_id = Column(sqlalchemy.Integer, nullable=False, primary_key=True,
                                                        autoincrement=False)
    task_id = Column(sqlalchemy.Integer, ForeignKey(TaskQueue.id),
                                                            nullable=False)
```

The consumer is a `rpclib.server._base.ServerBase` child that receives requests by polling the database.

The transport is for displaying it in the Wsdl. While it's irrelevant here, it's nice to put it in:

```
class Consumer(ServerBase):
    transport = 'http://sqlalchemy.persistent.queue/'
```

We set the incoming values, create a database connection and set it to *self.session*:

```
def __init__(self, db, app, consumer_id):
    ServerBase.__init__(self, app)

    self.session = sessionmaker(bind=db)()
    self.id = consumer_id
```

We also query the worker status table and get the id for the first task. If there is no record for own worker id, the server starts from the beginning:

```
try:
    self.session.query(WorkerStatus) \
                .filter_by(worker_id=self.id).one()
except NoResultFound:
    self.session.add(WorkerStatus(worker_id=self.id, task_id=0))
    self.session.commit()
```

This is the main loop for our server:

```
def serve_forever(self):
    while True:
```

We first get the id of the last processed task:

```
last = self.session.query(WorkerStatus).with_lockmode("update") \
            .filter_by(worker_id=self.id).one()
```

Which is used to get the next tasks to process:

```
task_queue = self.session.query(TaskQueue) \
        .filter(TaskQueue.id > last.task_id) \
        .order_by(TaskQueue.id)
```

Each task is an rpc request, so we create a `rpclib.MethodContext` instance for each task and set transport-specific data to the `ctx.transport` object:

```
for task in task_queue:
    ctx = MethodContext(self.app)
    ctx.in_string = [task.data]
    ctx.transport.consumer_id = self.id
    ctx.transport.task_id = task.id
```

This call parses the incoming request:

```
self.get_in_object(ctx)
```

In case of an error when parsing the request, the server logs the error and continues to process the next task in queue. The `get_out_string` call is smart enough to notice and serialize the error. If this was a normal server, we'd worry about returning the error to the client as well as logging it.

```
if ctx.in_error:
    self.get_out_string(ctx)
    logging.error(''.join(ctx.out_string))
    continue
```

As the request was parsed correctly, the user method can be called to process the task:

```
self.get_out_object(ctx)
```

The server should not care whether the error was an expected or unexpected one. So the error is logged and the server continues to process the next task in queue.

```
if ctx.out_error:
    self.get_out_string(ctx)
    logging.error(''.join(ctx.out_string))
    continue
```

If task processing went fine, the server serializes the out object and logs that instead.

```
self.get_out_string(ctx)
logging.debug(''.join(ctx.out_string))
```

Finally, the task is marked as processed.

last.task_id = task.id self.session.commit()

Once all tasks in queue are consumed, the server waits a pre-defined amount of time before polling the database for new tasks:

```
time.sleep(10)
```

This concludes the worker implementation. But how do we put tasks in the task queue? That's the job of the `Producer` class that is implemented as an Rpclib client.

Implementing clients is a two-stage operation. The main transport logic is in the `rpclib.client.RemoteProcedureBase` child that is a native Python callable whose function is to serialize the arguments, send it to the server, receive the reply, deserialize it and pass the return value to the python caller. However, in our case, the client does not return anything as calls are processed asyncronously.

We start with the constructor, where we initialize the SQLAlchemy database connection factory:

```
class RemoteProcedure(RemoteProcedureBase):
    def __init__(self, db, app, name, out_header):
        RemoteProcedureBase.__init__(self, db, app, name, out_header)

        self.Session = sessionmaker(bind=db)
```

The implementation of the client is much simpler because we trust that the Rpclib code will do The Right Thing. Here, we serialize the arguments:

```
def __call__(self, *args, **kwargs):
    session = self.Session()

    self.get_out_object(args, kwargs)
    self.get_out_string()
```

```
    out_string = ''.join(self.ctx.out_string)
```

And put the resulting bytestream to the database:

```
session.add(TaskQueue(data=out_string))
session.commit()
session.close()
```

Again here the function does not return anything because this is an asyncronous client.

Here's the `Producer` class whose sole purpose is to initialize the right callable factory.

```
class Producer(ClientBase):
    def __init__(self, db, app):
        ClientBase.__init__(self, db, app)

        self.service = Service(RemoteProcedure, db, app)
```

This is the worker service that will process the tasks.

```
class AsyncService(ServiceBase):
    @rpc(Integer)
    def sleep(ctx, integer):
        print "Sleeping for %d seconds..." % (integer)
        time.sleep(integer)
```

And this event is here to do some logging.

```
def _on_method_call(ctx):
    print "This is worker id %d, processing task id %d." % (
                            ctx.transport.consumer_id, ctx.transport.task_id)

AsyncService.event_manager.add_listener('method_call', _on_method_call)
```

It's now time to deploy our service. We start by configuring the logger and creating the necessary sql tables:

```
if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    logging.getLogger('sqlalchemy.engine.base.Engine').setLevel(logging.DEBUG)

    metadata.create_all()
```

We then initialize our application:

```
application = Application([AsyncService], 'rpclib.async',
        interface=Wsdl11(), in_protocol=Soap11(), out_protocol=Soap11())
```

And queue some tasks:

```
producer = Producer(db, application)
for i in range(10):
    producer.service.sleep(i)
```

And finally start the one worker to consume the queued tasks:

```
consumer = Consumer(db, application, 1)
consumer.serve_forever()
```

That's about it! You can switch to another database engine that accepts multiple connections and insert tasks from another connection to see the consumer in action. You could also start other workers in other processes to see the pub-sub functionality.

What's Next?

Start hacking! Good luck, and be sure to pop out to the mailing list if you have questions.

2.2 Rpclib API Reference

2.2.1 Fundamental Data Structures

MethodContext

MethodDescriptor

EventManager

Rpclib supports a simple event system that can be used to have repetitive boiler plate code that has to run for every method call nicely tucked away in one or more event handlers. The popular use-cases include things like database transaction management, logging and measuring performance.

Various Rpclib components support firing events at various stages during the processing of the request, which are documented in the relevant classes.

The classes that support events are:

- `rpclib.application.Application`
- `rpclib.service.ServiceBase`
- `rpclib.protocol._base.ProtocolBase`
- `rpclib.server.wsgi.WsgiApplication`

2.2.2 Models

In rpclib, models are used to mark how a certain message fragment will be converted to and from its native python format. It also holds validation information, and holds actually THE information interface document standards like WSDL are designed to exposed.

Rpclib's has built-in support most common data types and provides an API to those who'd like to implement their own.

Base Classes

Binary

Complex

Enum

Fault

Primitives

SQL Table

2.2.3 Interfaces

The *rpclib.interface* package contains the implementations of various schema definition standards.

Interface Base Class

XML Schema

Wsdl 1.1

2.2.4 Protocols

The *rpclib.protocol* package contains the implementations of various remote procedure call protocols.

Protocol Base Class

HttRpc

Soap 1.1

2.2.5 Client Transports

Client Base Class

HTTP

ZeroMQ

2.2.6 Server Transports

Server Base Class

HTTP (WSGI)

ZeroMQ

NullServer

2.2.7 Miscellanous Rpclib Utilities

Class Dictionary

Element Tree Conversion

Simple Wrappers

Xml Utilities

Ordered Dictionary

Ordered Set

Helpers

2.2.8 Application Definition

2.2.9 Service Definition

2.2.10 RPC Decorators

CHAPTER

THREE

OTHER GOODIES

3.1 Running Tests

While the test coverage for Rpclib is not that bad, we always accept new tests that cover new use-cases. Please consider contributing tests even if your use-case is working fine! Given the nature of open-source projects, Rpclib may shift focus or change maintainers in the future. This can result in patches which may cause incompatibilities with your existing code base. The only way to detect such corner cases is to have a great test suite.

3.1.1 Requirements

While simply executing test modules is normally enough to run Python tests, using py.test from pytest package is just a more pleasant way to run them. Simply easy_install pytest to get it. You can run the following command in the test directory:

```
py.test -v --tb=short
```

You can use the module name as an argument:

```
py.test -v --tb=short test_sqla.py
```

You can also choose which test to run:

```
py.test -v --tb=short test_sqla.py -k test_same_table_inheritance
```

See pytest documentation for more info.

Note that you need to do several other preparations to have the interop tests working. See the next section for the specifics.

3.1.2 Interoperability Tests

The interoperability servers require twisted.web.

Python

Python interop tests currently use Rpclib's own clients and suds. The suds test is the first thing we check and try not to break.

Two tests that fail in the suds interop tests due to the lack of proper assert statements, so they're false alarms.

Ruby

You need Ruby 1.8.x to run the ruby interop test against soap_http_basic. Unfortunately, the Ruby Soap client does not do proper handling of namespaces, so you'll need to turn off strict validation if you want to work with ruby clients.

Ruby test module is very incomplete, implementing only two (echo_string and echo_integer) tests. We're looking for volunteers who'd like to work on increasing test coverage for other use cases.

.Net

There isn't any .Net tests for rpclib. WS-I test compliance reportedly covers .Net use cases as well. Patches are welcome!

Java

The WS-I test is written in Java. But unfortunately, it only focuses on Wsdl document and not the soap functionality itself. We're looking for volunteers who'd like to work on writing Java interop tests for rpclib.

To run the Wsdl tests, you should first get wsi-interop-tools package from http://ws-i.org and unpack it next to test_wsi.py. Here are the relevant links:

http://www.ws-i.org/deliverables/workinggroup.aspx?wg=testingtools http://www.ws-i.org/Testing/Tools/2005/06/WSI_Test_Java_Final_1.1.zip

See also test_wsi.py for more info.

Now run the soap_http_basic interop test server and run test_wsi.py. If all goes well, you should get a new wsi-report-rpclib.xml file in the same directory.

Here's the directory tree from a working setup:

```
|-- README.rst
|-- (...)
|-- interop
|   |-- (...)
|   |-- test_wsi.py
|   `-- wsi-test-tools
|       |-- License.htm
|       |-- README.txt
|       `-- (...)
`-- (...)
```

3.2 Rpclib FAQ

Frequently asked questions about rpclib and related libraries.

3.2.1 Does rpclib support the SOAP 1.2 standard?

Short answer: No.

Long answer: Nope.

Patches are welcome.

3.2.2 How do I implement a predefined WSDL?

Short answer: By hand.

Long answer: This is not a strength of rpclib, which is more oriented toward implementing services from scratch. It does not have any functionality to parse an existing WSDL document to produce the necessary Python classes and method stubs.

Patches are welcome. You can start by adapting the WSDL parser from RSL.

3.2.3 Is it possible to use other decorators with @rpc/@srpc?

Short answer: Yes, but just use events. See the *User Manager* tutorial and the events example to learn how to do so. They work almost the same, except for the syntax.

Long Answer: Here's the magic from the `rpclib.decorator`:

```
argcount = f.func_code.co_argcount
param_names = f.func_code.co_varnames[arg_start:argcount]
```

So if `f` is your decorator, its signature should be the same as the user method, otherwise the parameter names and numbers in the interface are going to be wrong, which will cause weird errors.

Please note that if you just intend to have a convenient way to set additional method metadata, you can use the `_udp` argument to the `rpclib.decorator.srpc()` to your liking.

So if you're hell bent on using decorators, you should use the decorator package. Here's an example:

```
from decorator import decorator

def _do_something(func, *args, **kw):
    print "before call"
    result = func(*args, **kw)
    print "after call"
    return result

def my_decor(f):
    return decorator(_do_something, f)

class tests(ServiceBase):
    @my_decor
    @srpc(ComplexTypes.Integer, _returns=ComplexTypes.Integer)
    def testf(first):
        return first
```

Note that the place of the decorator matters. Putting it before `@srpc` will make it run once, on service initialization. Putting it after will make it run every time the method is called, but not on initialization.

Original thread: http://mail.python.org/pipermail/soap/2011-September/000565.html

PS: The next faq entry is also probably relevant to you.

3.2.4 How do I alter the behaviour of a user method without using decorators?

`ctx.descriptor.function` contains the handle to the original function. You can set that attribute to arbitrary callables to prevent the original user method from running.

Note that this property is initialized only when the process starts. So you should call `ctx.descriptor.reset_function()` to restore it to its original value

3.2.5 How do I use variable names that are also python keywords?

Due to restrictions of the python language, you can't do this:

class SomeClass(ComplexModel): and = String or = Integer import = Datetime

The workaround is as follows:

class SomeClass(ComplexModel):

_type_info = { 'and': String 'or': Integer 'import': Datetime

}

You also can't do this:

@rpc(String, String, String, _returns=String) def f(ctx, from, import):

return '1234'

The workaround is as follows:

@rpc(String, String, String, _returns=String,

_in_variable_names={'_from': 'from', '_import': 'import'},

_out_variable_name="return"

def f(ctx, _from, _import): return '1234'

See here: https://github.com/arskom/rpclib/blob/rpclib-2.5.0-beta/src/rpclib/test/test_service.py#L114

3.3 History and Future

3.3.1 Versioning

Rpclib respects Semantic Versioning rules outlined in http://semver.org.

This means you can do better than just adding `'rpclib'` to your list of dependencies. Assuming the current version of rpclib is 2.4.8, you can use the following as dependency string:

- `rpclib` if you feel adventurous or are willing to work on rpclib itself.
- `rpclib<3` if you only want backwards-compatible bug fixes and new features.
- `rpclib<2.5` if you only want backwards-compatible bug fixes.
- `rpclib=2.4.8` if you rather like that version.

We have a policy of pushing to pypi as soon as possible, so be sure to at least use the second option to prevent things from breaking unexpectedly.

Rpclib project uses -alpha -beta and -rc labels to denote unfinished code. We don't prefer using separate integers for experimental labels. So for example, instead of having 2.0.0-alpha47, we'll have 2.2.5-alpha.

- **-alpha** means unstable code. You may not recognize the project next time you look at it.
- **-beta** means stable(ish) api, unstable behavior, there are bugs everywhere! Don't be upset if some quirks that you rely on disappear.
- **-rc** means it's been working in production sans issues for some time on beta-testers' sites, but we'd still like it to have tested by a few more people.

These labels apply to the project as a whole. Thus, we won't tag the whole project as beta because some new feature is not yet well-tested, but we will clearly denote experimental code in its documentation.

3.3.2 Roadmap and Criticism

This is an attempt to make a free-for-all area to display opinions about the feature direction of rpclib. Doing it in a text file in the source repository may not be the best approach for the job, but it should be enough to at least spark a discussion around this topic.

So the following is missing in Rpclib:

Processing Pipeline

We think rpclib package has one last missing element whose addition can result in touching most of the codebase: A proper lazily-evaluated pipeline for request processing.

Currently, every artifact of the rpc processing pipeline remain in memory for the entire life time of the context object. This also results in having the whole message in memory while processing. While this is not a problem for small messages, which is rpclib's main target, it limits rpclib capabilities.

Serializer Support

See the *High-Level Introduction to Rpclib* section for a small introductory paragraph about serializers.

Currently, serializers are not distinguished in the rpclib source code. Making them pluggable would:

1. Make rpclib more flexible
2. Make it easy to share code between protocols.

An initial attempt to make them pluggable would result in the lxml dependency for Soap being relaxed, which would make it possible to deploy rpclib in pure-python environments. However, this is comparatively easy to do, given the fact that the ElementTree api is a well-known de-facto standard in the Python world.

Adding other serializers like json to the mix would certainly be a nice exercise in oo interface design, but this may be a solution in search of a problem. Would anybody be interested using Soap over Json instead of Xml? Probably not :)

It would, however, help newer serialization formats by reusing code from their more mature cousins. E.g. Soap already has a security layer defined. If the serializer is abstracted away, it could be easier to port security code from Soap to JsonRpc.

Miscellanous

The following would definitely be nice to have, but are just modules that should not cause a change in unrelated areas of rpclib. Those would increment the minor version number of the Rpclib version once implemented.

- Support for polymorphism in XML Schema.
- Support for the JsonObject (à la XmlObject) and JsonRpc protocols.
- Support for the JsonSchema interface document standard.
- Support for the Thrift binary protocol.
- Support for the Thrift IDL – The Thrift Interface Definition Language.
- Support for the XmlRpc standard.

- Support for EXI – The Efficient Xml Interchange as a serializer.
- SMTP as server transport.
- SMTP as client transport.
- Improve HttpRpc to be Rest compliant. Probably by dumping HttpRpc as it is and rewriting it as a wrapper to Werkzeug or a similar WSGI library.
- Implement converting csv output to pdf.
- Implement DNS as transport
- Support security extensions to Soap (maybe using PyXMLSec ?)
- Support addressing (routing) extensions to Soap
- Add WSDL Parsing support to Soap client
- Reflect transport and protocol pairs other than Soap/Http to the Wsdl.

3.3.3 Comparison with other rpc frameworks

RPC is a very popular subject. There's a plethora of active and inactive projects that satisfy a wide range of requirements. Here are the main sources of information:

- http://pypi.python.org/pypi?%3Aaction=search&term=rpc
- http://www.ohloh.net/tags/python/rpc
- http://stackoverflow.com/questions/1879971/what-is-the-current-choice-for-doing-rpc-in-python

Ladon

The Ladon project has almost the same goals and same approach to the rpc-related issues as rpclib.

Discussion thread: https://answers.launchpad.net/ladon/+question/171664

- **Supports JsonWSP protocol, which rpclib does not support.** The main motive for designing JSON-WSP was the need for a JSON-based RPC protocol with a service description specification with built-in service / method documentation.
- Supports both Python 2 and Python 3.
- Auto-generates human-readable API documentation. (example: http://ladonize.org/python-demos/AlbumService) In Rpclib, you need to do with the ugliness of a raw wsdl document.
- Does not support ZeroMQ.
- Uses standard python tools for xml parsing which is good for pure-python deployments. Rpclib uses lxml, due to its excellent namespace support and speed. So Rpclib-based solutions are easier to develop and faster to work with but more difficult to deploy.
- Does not do input validation for SOAP.
- Does not support events.
- Does not support HttpRpc.
- **Does not have a Soap client.** In fact, Ladon is pure server-side software - the whole idea of supporting a standard protocol like SOAP is that clients are already out there.
- Rpclib uses own classes for denoting types, whereas ladon uses Python callables. This lets ladon api to be simpler, but gives the rpclib api the power to have declarative restrictions on input types.

- Does not test against ws-i.org deliverables for testing soap compatibility.
- Does not support parsing and/or modifying protocol & transport headers.

WSME

“”” Web Service Made Easy (WSME) is a very easy way to implement webservices in your python web application. It is originally a rewrite of TGWebServices with focus on extensibility, framework-independance and better type handling. “””“

- Supports TurboGears
- Supports REST+Json, REST+Xml, (a subset of) SOAP and ExtDirect.
- Supports type validation.
- No client support.
- Does not test against ws-i.org deliverables for testing soap compatibility.
- Only supports http as transport.
- Uses genshi for Xml support.

RPyC

This is preliminary. Please correct these points if you spot any error.

- Uses own protocol
- Does not do validation.
- Python-specific.
- Fast.
- Not designed for public servers. ??

rfoo

This is preliminary. Please correct these points if you spot any error.

- Uses own protocol
- Does not do validation.
- Python-specific.
- Fast.
- Not designed for public servers. ??

Suds

- Soap 1.1 / Wsdl 1.1 Client only.
- Excellent wsdl parser, very easy to use.
- Recommended way to interface with rpclib services.
- Uses own pure-python xml implementation, so it’s roughly 10 times slower than rpclib.

- Only depends on pure-python solutions, so much easier to deploy.

ZSI

- Unmaintained, although still works with recent Python versions
- Contains SOAPpy, which is not the same as SOAPy (notice the extra P)
- Supports attachments
- Requires code generation (wsdl2py) for complex data structures
- Almost complete lack of user-friendliness
- Lack of WSDL generator

SOAPy

- Really simple (only two files, both less than 500 lines of code)
- Client only
- Requires PyXML, thus unusable with recent Python versions

rsl

- Client only.
- Unmaintained.

PyRo

- ???

3.3.4 Changelog

rpclib-2.5.2-beta

- Misc. fixes.
- Full change log: https://github.com/arskom/rpclib/pull/118

rpclib-2.5.1-beta

- Switched to magic cookie constants instead of strings in protocol logic.
- check_validator -> set_validator in ProtocolBase
- Started parsing Http headers in HttpRpc protocol.
- HttpRpc now properly validates nested value frequencies.
- HttpRpc now works with arrays of simple types as well.
- **Full change log: https://github.com/arskom/rpclib/pull/117** https://github.com/arskom/rpclib/pull/116

rpclib-2.5.0-beta

- Implemented fanout support for transports and protocols that can handle that.
- Implemented a helper module that generates a Soap/Wsdl 1.1 application in `rpclib.util.simple`
- Some work towards supporting Python3 using `2to3`. See issue #113.
- `ctx.descriptor.reset_function` implemented. It's now safe to fiddle with that value in event handlers.
- Incorporated a cleaned-up version of the Django wrapper: https://gist.github.com/1316025
- Fix most of the tests that fail due to api changes.
- Fix Http soap client.
- Full change log: https://github.com/arskom/rpclib/pull/115

rpclib-2.4.7-beta

- Made color in logs optional
- Fixed ByteArray serializer

rpclib-2.4.5-beta

- Time primitive was implemented.
- Fix for multiple ports was integrated.
- Added http cookie authentication example with suds.
- Full change log: https://github.com/arskom/rpclib/pull/109

rpclib-2.4.3-beta

- Many issues with 'soft' validation was fixed.
- `MethodDescriptor.udp` added. Short for "User-Defined Properties", you can use it to store arbitrary metadata about the decorated method.
- Fix HttpRpc response serialization.
- Documentation updates.

rpclib-2.4.1-beta

- Fixed import errors in Python<=2.5.
- A problem with rpclib's String and unicode objects was fixed.

rpclib-2.4.0-beta

- Fixed Fault publishing in Wsdl.
- Implemented 'soft' validation.
- Documentation improvements. It's mostly ready!
- A bug with min/max_occurs logic was fixed. This causes rpclib not to send null values for elements with min_occurs=0 (the default value).
- Native value for `rpclib.model.primitive.String` was changed to `unicode`. To exchange raw data, you should use `rpclib.model.binary.ByteArray`.
- Full change log: https://github.com/arskom/rpclib/pull/90

rpclib-2.3.3-beta

- Added MAX_CONTENT_LENGTH = 2 * 1024 * 1024 and BLOCK_LENGTH = 8 * 1024 constants to rpclib.server.wsgi module.
- rpclib.model.binary.Attachment is deprecated, and is replaced by ByteArray. The native format of ByteArray is an iterable of strings.
- Exception handling was formalized. HTTP return codes can be set by exception classes from rpclib.error or custom exceptions.
- Full change log: https://github.com/arskom/rpclib/pull/88

rpclib-2.3.2-beta

- Limited support for sqlalchemy.orm.relationship (no string arguments)
- Added missing event firings.
- Documented event api and fundamental data structures (rpclib._base)
- Full change log: https://github.com/arskom/rpclib/pull/87

rpclib-2.3.1-beta

- HttpRpc protocol now returns 404 when a requested resource was not found.
- New tests added for HttpRpc protocol.
- Miscellanous other fixes. See: https://github.com/arskom/rpclib/pull/86

rpclib-2.3.0-beta

- Documentation improvements.
- rpclib.protocol.xml.XmlObject is now working as out_protocol.
- Many fixes.

rpclib-2.2.3-beta

- Documentation improvements.
- rpclib.client.http.Client -> rpclib.client.http.HttpClient
- rpclib.client.zeromq.Client -> rpclib.client.zeromq.ZeroMQClient
- rpclib.server.zeromq.Server -> rpclib.server.zeromq.ZeroMQServer
- rpclib.model.table.TableSerializer -> rpclib.model.table.TableModel

rpclib-2.2.2-beta

- Fixed call to rpclib.application.Application.call_wrapper
- Fixed HttpRpc server transport instantiation.
- Documentation improvements.

rpclib-2.2.1-beta

- rpclib.application.Application.call_wrapper introduced
- Documentation improvements.

rpclib-2.2.0-beta

- The serialization / deserialization logic was redesigned. Now most of the serialization-related logic is under the responsibility of the ProtocolBase children.
- Interface generation logic was redesigned. The WSDL logic is separated to XmlSchema and Wsdl11 classes. 'add_to_schema' calls were renamed to just 'add' and were moved inside rpclib.interface.xml_schema package.
- Interface and Protocol assignment of an rpclib application is now more explicit. Both are also configurable during instantion. This doesn't mean there's much to configure :)
- WS-I Conformance is back!. See https://github.com/arskom/rpclib/blob/master/src/rpclib/test/interop/wsi-report-rpclib.xml for the latest conformance report.
- Numeric types now support range restrictions. e.g. Integer(ge=0) will only accept positive integers.
- Any -> AnyXml, AnyAsDict -> AnyDict. AnyAsDict is not the child of the AnyXml anymore.
- rpclib.model.exception -> rpclib.model.fault.

rpclib-2.1.0-alpha

- The method dispatch logic was rewritten: It's now possible for the protocols to override how method request strings are matched to methods definitions.
- Unsigned integer primitives were added.
- ZeroMQ client was fixed.
- Header confusion in native http soap client was fixed.
- Grouped transport-specific context information under ctx.transport attribute.
- Added a self reference mechanism.

rpclib-2.0.10-alpha

- The inclusion of base xml schemas were made optional.
- WSDL: Fix out header being the same as in header.
- Added type checking to outgoing Integer types. it's not handled as nicely as it should be.
- Fixed the case where changing the _in_message tag name of the method prevented it from being called.
- SOAP/WSDL: Added support for multiple {in,out}_header objects.
- Correct some bugs with the XMLAttribute model

rpclib-2.0.9-alpha

- Added inheritance support to rpclib.model.table.TableSerializer.

rpclib-2.0.8-alpha

- The NullServer now also returns context with the return object to have it survive past user-defined method return.

rpclib-2.0.7-alpha

- More tests are migrated to the new api.
- Function identifier strings are no more created directly from the function object itself. Function's key in the class definition is used as default instead.
- Base xml schemas are no longer imported.

rpclib-2.0.6-alpha

- Added rpclib.server.null.NullServer, which is a server class with a client interface that attempts to do no (de)serialization at all. It's intended to be used in tests.

rpclib-2.0.5-alpha

- Add late mapping support to sqlalchemy table serializer.

rpclib-2.0.4-alpha

- Add preliminary support for a sqlalchemy-0.7-compatible serializer.

rpclib-2.0.3-alpha

- Migrate the HttpRpc serializer to the new internal api.

rpclib-2.0.2-alpha

- SimpleType -> SimpleModel
- Small bugfixes.

rpclib-2.0.1-alpha

- EventManager now uses ordered sets instead of normal sets to store event handlers.
- Implemented sort_wsdl, a small hack to sort wsdl output in order to ease debugging.

rpclib-2.0.0-alpha

- Implemented EventManager and replaced hook calls with events.
- The rpc decorator now produces static methods. The methods still get an implicit first argument that holds the service contexts. It's an instance of the MethodContext class, and not the ServiceBase (formerly DefinitionBase) class.
- The new srpc decorator doesn't force the methods to have an implicit first argument.
- Fixed fault namespace resolution.
- Moved xml constants to rpclib.const.xml_ns
- **The following changes to soaplib were ported to rpclib's SOAP/WSDL parts:**
  - duration object is now compatible with Python's native timedelta.
  - WSDL: Support for multiple <service> tags in the wsdl (one for each class in the application)
  - WSDL: Support for multiple <portType> tags and multiple ports.
  - WSDL: Support for enumerating exceptions a method can throw was added.
  - SOAP: Exceptions got some love to be more standards-compliant.
  - SOAP: Xml attribute support
- Moved all modules with packagename.base to packagename._base.
- **Renamed classes to have module name as a prefix:**
  - rpclib.client._base.Base -> rpclib.client._base.ClientBase
  - rpclib.model._base.Base -> rpclib.model._base.ModelBase
  - rpclib.protocol._base.Base -> rpclib.protocol._base.ProtocolBase
  - rpclib.server._base.Base -> rpclib.server._base.ServerBase
  - rpclic.service.DefinitionBase -> rpclib.service.ServiceBase
  - rpclib.server.wsgi.Application -> rpclib.server.wsgi.WsgiApplication
- **Moved some classes and modules around:**
  - rpclib.model.clazz -> rpclib.model.complex
  - rpclib.model.complex.ClassSerializer -> rpclib.model.complex.ComplexModel
  - rpclib.Application -> rpclib.application.Application
  - rpclib.service.rpc, srpc -> rpclib.decorator.rpc, srpc

soaplib-3.x -> rpclib-1.1.1-alpha

- Soaplib is now also protocol agnostic. As it now supports protocols other than soap (like Rest-minus-the-verbs HttpRpc), it's renamed to rpclib. This also means soaplib can now support multiple versions of soap and wsdl standards.
- Mention of xml and soap removed from public api where it's not directly related to soap or xml. (e.g. a hook rename: on_method_exception_xml -> on_method_exception_doc)
- Protocol serializers now return iterables instead of complete messages. This is a first step towards eliminating the need to have the whole message in memory during processing.

soaplib-2.x

- This release transformed soaplib from a soap server that exclusively supported http to a soap serialization/deserialization library that is architecture and transport agnostic.
- Hard dependency on WSGI removed.
- Sphinx docs with working examples: http://arskom.github.com/rpclib/
- Serializers renamed to Models.
- Standalone xsd generation for ClassSerializer objects has been added. This allows soaplib to be used to define generic XML schemas, without SOAP artifacts.
- Annotation Tags for primitive Models has been added.
- The soaplib client has been re-written after having been dropped from recent releases. It follows the suds API but is based on lxml for better performance. WARNING: the soaplib client is not well-tested and future support is tentative and dependent on community response.
- 0mq support added.
- Twisted supported via WSGI wrappers.
- Increased test coverage for soaplib and supported servers

soaplib-1.0

- Standards-compliant Soap Faults
- Allow multiple return values and return types

soaplib-0.9.4

- pritimitive.Array -> clazz.Array
- Support for SimpleType restrictions (pattern, length, etc.)

soaplib-0.9.3

- Soap header support
- Tried the WS-I Test first time. Many bug fixes.

soaplib-0.9.2

- Support for inheritance.

soaplib-0.9.1

- Support for publishing multiple service classes.

soaplib-0.9

- Soap server logic almost completely rewritten.
- Soap client removed in favor of suds.
- Object definition api no longer needs a class types: under class definition.
- XML Schema validation is supported.
- Support for publishing multiple namespaces. (multiple <schema> tags in the wsdl)
- Support for enumerations.
- Application and Service Definition are separated. Application is instantiated on server start, and Service Definition is instantiated for each new request.
- @soapmethod -> @rpc

soaplib-0.8.1

- Switched to lxml for proper xml namespace support.

soaplib-0.8.0

- First public stable release.

3.4 Comparison with other rpc frameworks

RPC is a very popular subject. There's a plethora of active and inactive projects that satisfy a wide range of requirements. Here are the main sources of information:

- http://pypi.python.org/pypi?%3Aaction=search&term=rpc
- http://www.ohloh.net/tags/python/rpc
- http://stackoverflow.com/questions/1879971/what-is-the-current-choice-for-doing-rpc-in-python

3.4.1 Ladon

The Ladon project has almost the same goals and same approach to the rpc-related issues as rpclib.

Discussion thread: https://answers.launchpad.net/ladon/+question/171664

- **Supports JsonWSP protocol, which rpclib does not support.** The main motive for designing JSON-WSP was the need for a JSON-based RPC protocol with a service description specification with built-in service / method documentation.

- Supports both Python 2 and Python 3.
- Auto-generates human-readable API documentation. (example: http://ladonize.org/python-demos/AlbumService) In Rpclib, you need to do with the ugliness of a raw wsdl document.
- Does not support ZeroMQ.
- Uses standard python tools for xml parsing which is good for pure-python deployments. Rpclib uses lxml, due to its excellent namespace support and speed. So Rpclib-based solutions are easier to develop and faster to work with but more difficult to deploy.
- Does not do input validation for SOAP.
- Does not support events.
- Does not support HttpRpc.
- **Does not have a Soap client.** In fact, Ladon is pure server-side software - the whole idea of supporting a standard protocol like SOAP is that clients are already out there.
- Rpclib uses own classes for denoting types, whereas ladon uses Python callables. This lets ladon api to be simpler, but gives the rpclib api the power to have declarative restrictions on input types.
- Does not test against ws-i.org deliverables for testing soap compatibility.
- Does not support parsing and/or modifying protocol & transport headers.

3.4.2 WSME

“”” Web Service Made Easy (WSME) is a very easy way to implement webservices in your python web application. It is originally a rewrite of TGWebServices with focus on extensibility, framework-independance and better type handling. “””“

- Supports TurboGears
- Supports REST+Json, REST+Xml, (a subset of) SOAP and ExtDirect.
- Supports type validation.
- No client support.
- Does not test against ws-i.org deliverables for testing soap compatibility.
- Only supports http as transport.
- Uses genshi for Xml support.

3.4.3 RPyC

This is preliminary. Please correct these points if you spot any error.

- Uses own protocol
- Does not do validation.
- Python-specific.
- Fast.
- Not designed for public servers. ??

3.4.4 rfoo

This is preliminary. Please correct these points if you spot any error.

- Uses own protocol
- Does not do validation.
- Python-specific.
- Fast.
- Not designed for public servers. ??

3.4.5 Suds

- Soap 1.1 / Wsdl 1.1 Client only.
- Excellent wsdl parser, very easy to use.
- Recommended way to interface with rpclib services.
- Uses own pure-python xml implementation, so it's roughly 10 times slower than rpclib.
- Only depends on pure-python solutions, so much easier to deploy.

3.4.6 ZSI

- Unmaintained, although still works with recent Python versions
- Contains SOAPpy, which is not the same as SOAPy (notice the extra P)
- Supports attachments
- Requires code generation (wsdl2py) for complex data structures
- Almost complete lack of user-friendliness
- Lack of WSDL generator

3.4.7 SOAPy

- Really simple (only two files, both less than 500 lines of code)
- Client only
- Requires PyXML, thus unusable with recent Python versions

3.4.8 rsl

- Client only.
- Unmaintained.

3.4.9 PyRo

- ???

CHAPTER

FOUR

INDICES AND TABLES

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