A Quick Look At Rpclib

Burak Arslan
burak at arskom dot com dot tr

March 8, 2012

What is Rpclib?

Rpclib makes it convenient to

expose your services using multiple

protocols and/or transports.

How?

Here's a simple function:

Here's a simple function:

```
from datetime import datetime

def get_utc_time():
    return datetime.utcnow()
```

Now, to make this function remotely callable;

Now, to make this function remotely callable;

1)

We wrap it in a `ServiceBase` child:

```
def get_utc_time():
    return datetime.utcnow()
```

```python
from rpclib.model.primitive import DateTime
from rpclib.decorator import srpc
from rpclib.service import ServiceBase


    def get_utc_time():
        return datetime.utcnow()
```

```
from rpclib.model.primitive import DateTime
from rpclib.decorator import srpc
from rpclib.service import ServiceBase

class DateTimeService(ServiceBase):

    def get_utc_time():
        return datetime.utcnow()
```

```
from rpclib.model.primitive import DateTime
from rpclib.decorator import srpc
from rpclib.service import ServiceBase

class DateTimeService(ServiceBase):
    @srpc(_returns=DateTime)
    def get_utc_time():
        return datetime.utcnow()
```

2)

Now, we have to wrap the service definition in an `Application` definition.

```
[DateTimeService],
```

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

httprpc = Application(
        [DateTimeService],
```

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

httprpc = Application(
        [DateTimeService],
        tns='rpclib.examples.multiprot',
```

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

httprpc = Application(
        [DateTimeService],
        tns='rpclib.examples.multiprot',
        in_protocol=HttpRpc(),
        out_protocol=HttpRpc()
    )
```

3)

Finally, we wrap the application in

a transport.

```
from rpclib.server.wsgi import WsgiApplication

application = WsgiApplication(httprpc)
```

This is now a regular WSGI Application that

we can pass to wsgi-compliant servers like

CherryPy, mod_wsgi, etc.

```
from rpclib.server.wsgi import WsgiApplication

application = WsgiApplication(httprpc)
```

This is now a regular WSGI Application that

we can pass to wsgi-compliant servers like

CherryPy, mod_wsgi, etc.

```
$ curl http://localhost:9910/get_utc_time
2012-03-09T17:38:11.997784
```

Now, what if we wanted to expose this function using another protocol?

For example: SOAP

```
from rpclib.application import Application
from rpclib.protocol.soap import Soap11

soap = Application([DateTimeService],
        tns='rpclib.examples.multiprot',
        in_protocol=HttpRpc(),
        out_protocol=Soap11()
    )
```

For example: SOAP

```
$ curl http://localhost:9910/get_utc_time \
                                    | tidy -xml -indent
```

```
<?xml version='1.0' encoding='utf-8'?>
<senv:Envelope xmlns:wsa="http://schemas.xmlsoap.org/ws/2003/03/addressing"
 xmlns:tns="rpclib.examples.multiple_protocols"
 xmlns:plink="http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
 xmlns:xop="http://www.w3.org/2004/08/xop/include"
 xmlns:senc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:s12env="http://www.w3.org/2003/05/soap-envelope/"
 xmlns:s12enc="http://www.w3.org/2003/05/soap-encoding/"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:senv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <senv:Body>
    <tns:get_utc_timeResponse>
      <tns:get_utc_timeResult>
        2012-03-06T17:43:30.894466
      </tns:get_utc_timeResult>
    </tns:get_utc_timeResponse>
  </senv:Body>
</senv:Envelope>
```

Or, just XML:

```
from rpclib.application import Application
from rpclib.protocol.xml import XmlObject

xml = Application([DateTimeService],
        tns='rpclib.examples.multiprot',
        in_protocol=HttpRpc(),
        out_protocol=XmlObject()
    )
```

Or, just XML:

```
$ curl http://localhost:9910/get_utc_time \
                                    | tidy -xml -indent

<?xml version='1.0' encoding='utf-8'?>
<ns0:get_utc_timeResponse
xmlns:ns0="rpclib.examples.multiple_protocols">
  <ns0:get_utc_timeResult>
    2012-03-06T17:49:08.922501
  </ns0:get_utc_timeResult>
</ns0:get_utc_timeResponse>
```

Or, HTML:

```
from rpclib.application import Application
from rpclib.protocol.xml import HtmlMicroFormat

html = Application([DateTimeService],
        tns='rpclib.examples.multiprot',
        in_protocol=HttpRpc(),
        out_protocol=HtmlMicroFormat()
    )
```

Or, HTML:

```
$ curl http://localhost:9910/get_utc_time \
                                    | tidy -xml -indent

<div class="get_utc_timeResponse">
  <div class="get_utc_timeResult">
    2012-03-06T17:52:50.234246
  </div>
</div>
```

etc…

Rpclib also makes it easy to implement custom protocols.

Let's implement an output protocol that renders the datetime value as an analog clock.

(without going into much detail ☺)

To do that, we need to implement the `serialize` and `create_out_string` functions in a `ProtocolBase` child.

```
from rpclib.protocol import ProtocolBase

class SvgClock(ProtocolBase):
  mime_type = 'image/svg+xml'
```

```
from rpclib.protocol import ProtocolBase

class SvgClock(ProtocolBase):
  mime_type = 'image/svg+xml'

  def serialize(self, ctx, message):
    d = ctx.out_object[0] # the return value
```

```
from rpclib.protocol import ProtocolBase

class SvgClock(ProtocolBase):
  mime_type = 'image/svg+xml'

  def serialize(self, ctx, message):
    d = ctx.out_object[0] # the return value

    # (some math and boilerplate suppressed)
```

```
from rpclib.protocol import ProtocolBase

class SvgClock(ProtocolBase):
  mime_type = 'image/svg+xml'

  def serialize(self, ctx, message):
    d = ctx.out_object[0] # the return value

    # (some math and boilerplate suppressed)

    # clock is a svg file parsed as lxml Element
    ctx.out_document = clock
```

```
from rpclib.protocol import ProtocolBase

class SvgClock(ProtocolBase):
  mime_type = 'image/svg+xml'

  def serialize(self, ctx, message):
    d = ctx.out_object[0] # the return value

    # (some math and boilerplate suppressed)

    # clock is a svg file parsed as lxml Element
    ctx.out_document = clock

  def create_out_string(self, ctx, charset=None):
    ctx.out_string = [
        etree.tostring(ctx.out_document)
    ]
```

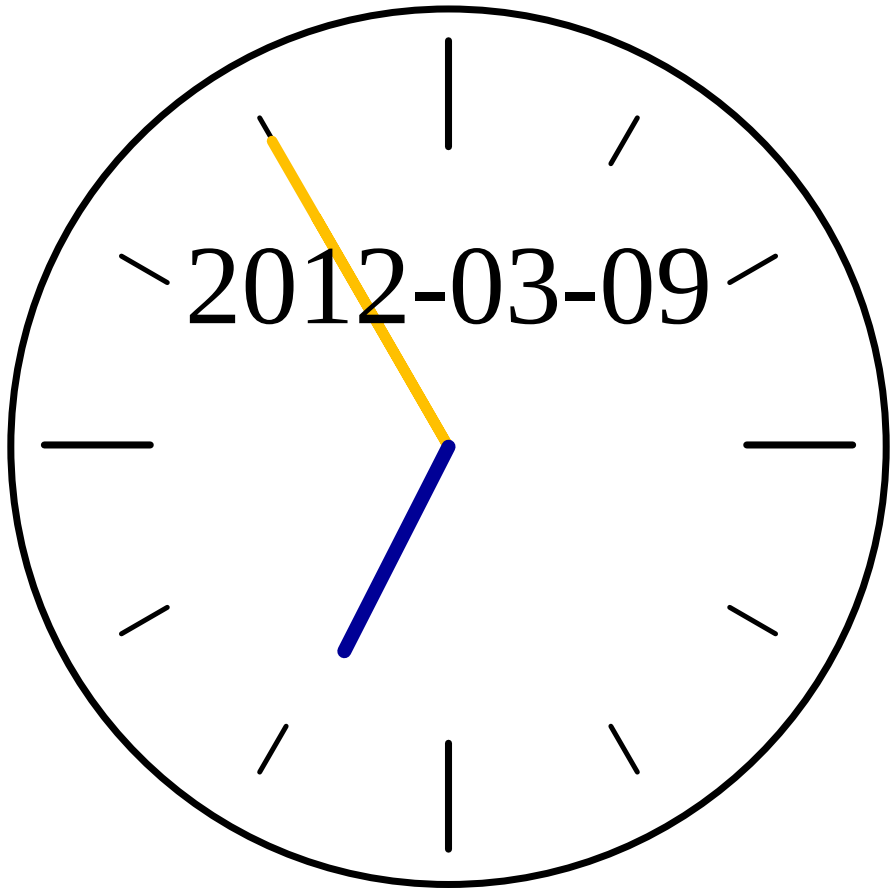
The custom SVG protocol:

```
from rpclib.application import Application

svg = Application([DateTimeService],
        tns='rpclib.examples.multiprot',
        in_protocol=HttpRpc(),
        out_protocol=SvgClock()
    )
```

The custom SVG protocol:

```
$ curl http://localhost:9910/get_utc_time \
                                    > utc_time.svg
```



So, what's missing?

Protocols: JSON! ProtoBuf! XmlRpc!
HTML! (The whole document)

Transports: SMTP! Files! BitTorrent!

(and many other things! see the ROADMAP.rst in the source repo.)

Additional Information:

github.com/arskom/rpclib

This example and the presentation are in:
examples/multiple_protocols