

Taller HispaLinux 2003: Servicios Web (Web Services) en Python

Autor: Ernesto Revilla – erny@sicem.biz

Revisiones: Lorenzo Gil Sánchez, lgs@sicem.biz

Versión: 0.2, borrador

Fecha: 18-08-03

Contenido

Introducción	3
Ejemplo base: Un sistema de pedido electrónico	4
XML-RPC (XML Remote Procedure Call).....	5
Historia breve	5
Descripción de la tecnología	6
Tipos de datos los argumentos.....	7
Tratamientos de excepciones.....	7
Tratamiento de la URL.....	7
Objetos que responden a solicitudes XML-RPC	8
XML-RPC y unicode	8
Ejemplo de Servidor simple: multiplicador de números:.....	8
Ejemplo de Servidor: Pedidos como Servicio Web	9
Ejemplo de Cliente: Usando el Servicio de Pedidos	10
Introspección	11
Servidor simple de XML-RPC con inspección	11
Ejemplo de uso de introspección.....	12
XML-RPC con Webware	13
XML-RPC con Zope	14
Ejemplo simple de XML-RPC con Zope	15
Zope, XML-RPC y seguridad.....	15
SOAP: Simple Object Access Protocol.....	19
Introducción.....	19
Descripción de la tecnología	19
Espacios de nombres (namespaces) XML	21
Paso de parámetros en SOAP	21
Tipos de datos	21
SOAPpy.....	22
Tipos de datos simples	23
Tipos de datos complejos	24
Fuentes	24

XML	24
XML-RPC	24
SOAP.....	24

1. WSDL (Web Service Description Language)
 - a. Descripción
 - b. WSDL con SOAPpy
 - c. Ejemplo
2. UDDI
 - a. UDDI4Py (IBM)
3. Seguridad
 - a. Conexiones seguras
 - b. Firmas digitales de documentos XML
4. CORBA
5. Envío de documentos mediante SOAP
6. Estándares para el envío de documentos de negocio
 - a. ebXML
 - b. cXML
 - c. UBL

Introducción

En los últimos tiempos, Internet ha tenido un auge espectacular. En concreto, la tecnología Web ha demostrado ser extremadamente simple y flexible para publicar y encontrar información, ayudando a satisfacer nuestras necesidades de comunicación.

Mientras que en la etapa inicial de la Web, se ha pretendido publicar información destinada a usuarios finales, en esta segunda etapa el objetivo es que los programas se comunican entre sí para la automatización de tareas. Esto se conoce por **Webs semánticas**¹. Ejemplos son buscadores que automáticamente clasifican los contenidos Web para un posterior acceso optimizado por personas, buscadores de servicios especializados en regiones geográficas, programas que realizan automáticamente compras según precios, condiciones de calidad y plazos de entrega, etc.

Definición²: Un **servicio Web** es una **colección de funciones** que están empaquetadas como **una sola entidad y publicada para ser usado por otros programas**.

En principio, la problemática a resolver es la siguiente:

- **Interoperabilidad**
Deseamos que se puedan consumir y ofrecer servicios desde cualquier ordenador, lenguaje o entorno de programación
- **Fiabilidad**
Un servicio de intercambio de órganos para trasplantes ha de tener una alta disponibilidad. ¿Qué pasa si falla? ¿Pueden crearse sistemas tolerantes a fallos?
- **Seguridad**
Si nuestro banco ofrece servicios Web para realizar transferencias, ¿cómo se asegura que se trata de una persona o un sistema autorizado?
- **Comportamiento transaccional**
El encargo de un artículo a medida requiere posiblemente comprobar las existencias y en encargo al un proveedor del material que falta. Esta operación compleja puede fallar, por ejemplo, si ya no se fabrica algunos de los componentes necesarios, con lo que la operación en su totalidad sólo tiene éxito si todas la operaciones individuales lo tienen.
- **Escalabilidad**
Si se desborda un servicio, ¿es fácil agregar más hardware y software para satisfacer las necesidades?
- **Publicación y descubrimiento**
¿Cómo se sabe qué servicios existen y cómo se usan?
- **Seguimiento**
¿Cómo se puede ver qué es lo que ha pasado, quién lo ha accedido, cuántos recursos ha consumido, etc.?

La gran popularidad de las tecnologías Web, sin embargo, ha resultado en estándares y especificaciones que se duplican en funcionalidad, dificultan la interoperabilidad e ignoran en mayor medida problemas que ya estaban resueltos anteriormente.

Así, por ejemplo, XML-RPC y SOAP duplican cierta funcionalidad de RPC y CORBA, sin embargo, no tratan transacciones y otros aspectos más difíciles de resolver³. Por otro lado, WSDL

¹ Berners-Lee, Tim: <http://www.w3.org/DesignIssues/Semantic.html>

² Glass, Graham: <http://www-106.ibm.com/developerworks/webservices/library/ws-peer1.html>

(Web Service Description Language) podía haberse basado en RDF (Resource Description Framework), un marco para dar semántica a los contenidos Web⁴.

El resultado es que hay múltiples tecnologías intentando resolver más o menos el mismo problema, en otras palabras, estamos reinventando la rueda una y otra vez.

Este documento no pretende de ninguna manera valorar o comparar las tecnologías disponibles, sino sólo mostrar la facilidad que ofrece Python y una serie de librerías y productos para crear y consumir servicios Web.

Ejemplo base: Un sistema de pedido electrónico

Para los ejemplos posteriores consideramos un sistema simplificado de pedidos. Disponemos de una lista de artículos, cada uno con una referencia única y su número de unidades en existencias. Adicionalmente, registramos las unidades que han sido reservados para servir pedidos ya aceptados.

```
class ErrorAplicacion(Exception): pass # para posteriores errores de aplicacion

class Artículo:
    def __init__(self, ref, existenciasIniciales):
        self.ref = ref
        self.existencias = existenciasIniciales
        self.udsReservadas = 0 # para pedidos aceptados

# creamos una colección de artículos
articulos={}
for ref, existenciasIniciales in [ ('camisa', 10), ('pantalon', 20) ]:
    articulos[ref] = Artículo(ref, existenciasIniciales)
```

Por otro lado, disponemos de una lista de clientes, cada uno con su nombre y un código único.

```
class ErrorCodigoClienteYaEnUso(ErrorAplicacion): pass

class ColeccionClientes(dict):

    def anyadir(self, cliente): # que dé error si agregamos un cliente
        if cliente.codigo in self: # con código ya existente.
            raise ErrorCodigoClienteYaEnUso
        else:
            dict.__setitem__(self, cliente.codigo, cliente)

class Cliente:
    contador = 1
    def __init__(self, nombre, codigo=None):
        if codigo:
            self.codigo = codigo
        else:
            self.codigo = "%5d" % self.__class__.contador # podría ser self.contador
            self.__class__.contador += 1 # incrementamos variable de clase

# vamos a crear algunos clientes
clientes = ColeccionClientes()
for nombre, codigo in [ ('Paco', 'Paco'), ('Pepe', None), ('Erny', None) ]:
    cliente = Cliente(nombre, codigo)
    clientes.anyadir(cliente)
```

³ Grisby, Duncan: <http://www.grisby.org/presentations/euro2002.pdf>,
<http://www.grisby.org/presentations/accu2003.pdf>,

⁴ Ogbuji, Uche: <http://www.adtmag.com/article.asp?id=6004>

Ahora, nuestro sistema simplificado de pedidos, que inicialmente no contiene ningún tipo de lógica de comunicación:

```
class ErrorArticuloNoValido(ErrorAplicacion): pass
class ErrorNoHayStock(ErrorAplicacion): pass
class ErrorClienteNoValido(ErrorAplicacion): pass

pedidos=[] # lista donde guardamos los pedidos
class Pedido:
    contadorPedidos=0

    def __init__(self, codigoCliente):
        if not codigoCliente in clientes:
            raise ErrorClienteNoValido
        self.cliente = clientes[codigoCliente]
        self.estado = 'N' # No servido
        self.articulos = [] # lista de tuplas (articulo, uds)

    def anyadirArticulo(self, ref, uds):
        try:
            articulo=articulos[ref]
        except KeyError:
            raise ArticuloNoValido, "El articulo %s no es valido." % ref
        udsDisponibles = articulo.existencias - articulo.udsReservadas
        if udsDisponibles < uds:
            raise ErrorNoHayStock, "El encargo del articulo %s sobrepasa " \
                + "las existencias actuales (%s)." % (ref, unidadesDisponibles)
        articulo.udsReservadas += uds
        self.articulos.append( (articulo, uds) )

    def guardar(self):
        self.__class__. contadorPedidos += 1
        self.num = "%5d" % contadorPedidos
        pedidos.append(self)
```

XML-RPC (XML Remote Procedure Call)

Historia breve

La primera implementación de llamadas a procedimientos remotos (RPC) fue presentada formalmente en 1984⁵ y la primera especificación fue donada por SUN y acogida por IETF en Abril de 1988⁶. Así, la intercomunicación de programas en entornos heterogéneos es ya relativamente antigua y existe abundante material introductorio⁷.

XML-RPC⁸ surgió en 1998 como parte del producto Frontier de UserLand (<http://frontier.userland.com>), un sistema de gestión de contenidos Web, y fue consecuencia de la necesidad de intercomunicar programas y de la popularidad Web y XML. La idea era aplanar las diferencias entre CORBA, DCOM (Microsoft) y Apple Events⁹ usando XML (como lenguaje para los mensajes de intercambio) y HTTP (para transportar los mensajes) como puente, con el intento de reemplazar RPC y XDR (eXternal Data Representation). Cabe destacar la falta de apoyo de Microsoft a CORBA y la falta de implementaciones económicas o libres en esta época, de manera que XML-RPC puede considerarse resultado de las circunstancias.

⁵ Birrell, Adrew D.; Nelson, Bruce Jay: Implementing Remote Procedure Call, ACM Trans. On Computer Systems, 1984, pp 39-54

⁶ Sun Microsystems, Inc: <http://rfc.sunsite.dk/rfc/rfc1050.html>

⁷ <http://gsync.escet.urjc.es/docencia/asignaturas/redes-II/transparencias/rpc/rpc.html>

⁸ <http://www.xmlrpc.com/>

⁹ Winer, Dave: [http://frontier.userland.com/stories/storyReader\\$1124](http://frontier.userland.com/stories/storyReader$1124)

Descripción de la tecnología

Básicamente, se trata de codificar la llamada a un procedimiento mediante XML. Para ellos necesitamos:

- especificar el nombre de la función a invocar
- codificar los argumentos mediante una expresión independiente de lenguaje de programación y hardware
- tratar errores (excepciones)
- mandar el mensaje
- recuperar y traducir a la inversa el mensaje de respuesta

Veamos un ejemplo¹⁰ en Python:

```
>>> from xmlrpclib import Server
>>> server=ServerProxy("http://betty.userland.com")
>>> server.examples.getStateName(10)
Georgia
```

produce el siguiente mensaje **HTTP-Post**:

```
POST /RPC2 HTTP/1.0
Host: betty.userland.com
User-Agent: xmlrpclib.py/1.0.0 (by www.pythonware.com)
Content-Type: text/xml
Content-length: 161
```

```
<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><int>10</int></value>
    </param>
  </params>
</methodCall>
```

Recibimos la siguiente respuesta del servidor:

```
HTTP/1.1 200 OK
Content-Length: 136
Content-Type: text/xml
Date: Date: Thu, 31 Jul 2003 12:25:40 GMT
Server: Server: UserLand Frontier/9.0-WinNT
Age: 0
```

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value>Georgia</value>
    </param>
  </params>
</methodResponse>
```

¹⁰ <http://xmlrpc.com/spec>

Tipos de datos los argumentos

Dado que los argumentos tiene que convertirse a un formato legible en el otro lado de canal de comunicación, sólo de admiten argumentos de tipos simples:

- enteros de 4 bytes con signo (int)
- boléanos (boolean)
- cadenas de caracteres (string)
- números flotantes de precisión doble (double)
- fecha / hora en formato ISO8601
- datos binarios codificados en formato base64 (base64)

Además, se permiten tipos de datos compuestos:

- Estructuras (struct) de datos simples, como por ejemplo "persona", con persona.nombre="Pepe", persona.apellidos="Perez", persona.edad=35, ...
- Vectores (array) de datos simples, que no tiene que ser del mismo tipo, como por ejemplo, [1,2, "hola", -2.344]
- Anidaciones de los anteriores de cualquier profundidad, es decir, una estructura puede contener a su vez estructuras y vectores, lo mismo que los vectores.

Tratamientos de excepciones

Pueden producirse diferentes errores en las llamadas a funciones remotas:

- método no encontrado
- error en la función llamada
- error de transporte (time-out, host no encontrado, protocolo incorrecto)
- problemas con la conversión de los argumentos

Cuando se produzca un error o una excepción en procedimiento remoto, se lanza una excepción. Además de las excepciones de la capa de red, hay otras específicas de xmlrpc-lib:

- IOError, unsupported XML-RPC protocol: Todas las especificaciones del objeto ServerProxy tienen que empezar por "http://" o "https://".
- ProtocolError: El servidor probablemente no es un servidor XML-RPC, o no entiende alguna parte del mensaje.
- ResponseError: Paquete de respuesta no válido.
- TypeError, OverflowError: Algunos de los argumentos no puede ser convertido al formato requerido por XML-RPC
- Fault: Error en el procedimiento, demasiados argumentos, procedimiento no existente, etc.

Para ver la especificación y detalles de implementación se recomienda la lectura de <http://www.xmlrpc.com/spec> y del archivo código fuente ./Lib/xmlrpc-lib.py .

Tratamiento de la URL

La URL se tratamiento de la misma manera que en solicitudes Web y es del formato:

<protocol>://<servidor>[:<puerto>][ruta]

La parte de protocolo de URL puede ser:

- HTTP: transporte normal, HTTP 1.0
- HTTPS: transporte seguro, si disponible en la versión Python y las librerías necesarias están instaladas

Si no se indica el puerto, se usa el defecto 80 para el transporte HTTP y 443 para HTTPS.

Si la ruta no está especificada, xmlrpclib.Server agrega automáticamente la ruta /RPC2. Por lo tanto, no es lo mismo poner **http://servidor** que **http://servidor/**. Nótese la barra final.

Objetos que responden a solicitudes XML-RPC

La clase SimpleXMLRPCServer de la librería estándar de Python 2.2 y posterior ofrece dos funciones para definir qué objetos pueden responder a llamadas XML-RPC:

- `register_function`: permite especificar cualquier función para responder. El primer argumento es una función. El segundo argumento, que es opcional, permite especificar un nombre alternativo que se usa para publicar la función. Puede llamarse esta función las veces que se desee para enumerar todas las funciones que se ofrecen dentro de un servicio Web.
- `register_instance`: permite registrar una única instancia de cualquier clase que trata las llamadas recibidas. Prevalecen las funciones registradas con `register_function`, independientemente del orden del uso de estas dos funciones. Si esta clase define un método `_dispatch`, todas las llamadas irán a través de este procedimiento. La signatura es: `_dispatch(self, methodName, args)`, donde `args` es una lista. El valor de retorno será devuelto al procedimiento llamante. No se permite el uso de argumentos nombrados con `_dispatch`.

En caso de no usar `_dispatch`, si el nombre del método contiene puntos, se intentará ubicar los atributos de la instancia que corresponden por su nombre, por ejemplo, `operaciones.calculos.sumar(x,y)` buscaría por el atributo 'operaciones', dentro de éste por un atributo con 'calculos' y finalmente la función `sumar`. Si no se encuentra algún atributo, ocurrirá una excepción.

XML-RPC y unicode

XML-RPC trata correctamente el envío y la recepción de cadenas de caracteres fuera del rango ASCII (0-127). Para ello es importante que la cadena sea realmente de tipo unicode. Aunque Python nos permita usar cadenas tipo 'Martínez', la llamada a un procedimiento remoto que un tipo de dato como éste da ProtocolError. Al usar `u'Martínez'` o más explícitamente `unicode('Martínez', 'latin-1')` los datos se envían al servidor usando la codificación de XML por defecto UTF-8 dentro de la etiqueta `<string>`. Al recibir un dato de tipo cadena de caracteres (string) que no puede ser codificado como ASCII, la librería devuelve automáticamente el dato como unicode. Para tratar con otros sistemas que no usan UTF-8 por defecto, `xmlrpclib.Server` permite especificar otra codificación, como 'latin-1'.

Ejemplo de Servidor simple: multiplicador de números:

Servidor:

```
def multiplica(x,y):
    return x*y

def divide(x,y):
    return x/y

from SimpleXMLRPCServer import SimpleXMLRPCServer
s = SimpleXMLRPCServer(("localhost",8001))
s.register_function(multiplica)
```

```
s.register_function(divide)
s.serve_forever()
```

Cliente:

```
>>> from xmlrpclib import Server
>>> s=ServerProxy("http://localhost:8001")
>>> s.multiplica(10,30)
300

>>> s.multiplica("ax",5)
'axaxaxaxax'

>>> s.multiplica("ax","kk")
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "C:\prg\Python22\lib\xmlrpclib.py", line 821, in __call__
    return self.__send(self.__name, args)
  File "C:\prg\Python22\lib\xmlrpclib.py", line 975, in __request
    verbose=self.__verbose
  File "C:\prg\Python22\lib\xmlrpclib.py", line 853, in request
    return self.parse_response(h.getfile())
  File "C:\prg\Python22\lib\xmlrpclib.py", line 896, in parse_response
    return u.close()
  File "C:\prg\Python22\lib\xmlrpclib.py", line 571, in close
    raise apply(Fault, (), self._stack[0])
Fault: <Fault 1: "exceptions.TypeError:unsupported operand type(s) for *: 'str' and 'str'">

>>> s.divide(5,2)
2

>>> s.divide(5.0,2)
2.5

>>> s.divide(2,0)
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "C:\prg\Python22\lib\xmlrpclib.py", line 821, in __call__
    return self.__send(self.__name, args)
  File "C:\prg\Python22\lib\xmlrpclib.py", line 975, in __request
    verbose=self.__verbose
  File "C:\prg\Python22\lib\xmlrpclib.py", line 853, in request
    return self.parse_response(h.getfile())
  File "C:\prg\Python22\lib\xmlrpclib.py", line 896, in parse_response
    return u.close()
  File "C:\prg\Python22\lib\xmlrpclib.py", line 571, in close
    raise apply(Fault, (), self._stack[0])
Fault: <Fault 1: 'exceptions.ZeroDivisionError:integer division or modulo by zero'>
```

Ejemplo de Servidor: Pedidos como Servicio Web

Nuestro cometido es ahora definir un servicio Web que permite dar de alta pedidos, comprobar su estado (de entrega) o cancelarlo, siempre que no haya sido despachado todavía. Aquí va el código para la parte del servidor:

```
class ErrorPedidoNoPuedeSerCancelado(ErrorAplicacion): pass

class ServicioPedidos:
    def __init__(self):

    def crearPedido(self, articulos):
        pedido = Pedido() # crear un nuevo pedido

        for cdg, uds in articulos: # recibimos una lista de tuplas
            pedido.anyadirArticulo(cdg, uds) # agregar artículo por artículo
        pedido.guardar() # guardar pedido
```

```

        return pedido.num                # devolver nº de pedido

def comprobarEstado(self, numPedido):
    return pedidos[numPedido].estado

def cancelarPedido(self, num):
    if pedidos[num].estado=='N':
        del pedidos[num]
    else:
        raise PedidoNoPuedeSerCancelado

ps=PedidoService()
# cosas específicas para la comunicación de XML-RPC
from SimpleXMLRPCServer import SimpleXMLRPCServer
server=SimpleXMLRPCServer(('localhost',8080))
server.register_instance(ps)
server.serve_forever()

```

Ejemplo de Cliente: Usando el Servicio de Pedidos

Aquí va un ejemplo de código cliente para el :

```

from xmlrpclib import Server
s = ServerProxy("http://localhost:8080")
# crear un nuevo pedido
numPedido = s.crearPedido("Paco","ac",[ ('a',10), ('b',5) ])
# si queremos comprobar el estado
print "Estado del pedido %s es %s" % (numPedido, s.comprobarEstado(numPedido))
# y si queremos cancelarlo
s.cancelarPedido(numPedido)

# vamos a mandar un pedido con contraseña incorrecta
s.crearPedido('Paco', 'kk', [ ('a',20) ] )

```

Obtenemos la siguiente traza:

```

Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "C:\prg\Python22\lib\xmlrpclib.py", line 821, in __call__
    return self.__send(self.__name, args)
  File "C:\prg\Python22\lib\xmlrpclib.py", line 975, in __request
    verbose=self.__verbose
  File "C:\prg\Python22\lib\xmlrpclib.py", line 853, in request
    return self.parse_response(h.getfile())
  File "C:\prg\Python22\lib\xmlrpclib.py", line 896, in parse_response
    return u.close()
  File "C:\prg\Python22\lib\xmlrpclib.py", line 571, in close
    raise apply(Fault, (), self._stack[0])
Fault: <Fault 1: '__main__.ErrorPasswordIncorrecto:'>

```

O pedimos una cantidad más allá de las existencias disponibles:

```
s.crearPedido('Paco','ac',[ ('a', 100) ] )
```

y obtenemos la siguiente traza:

```

Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "C:\prg\Python22\lib\xmlrpclib.py", line 821, in __call__
    return self.__send(self.__name, args)
  File "C:\prg\Python22\lib\xmlrpclib.py", line 975, in __request
    verbose=self.__verbose
  File "C:\prg\Python22\lib\xmlrpclib.py", line 853, in request
    return self.parse_response(h.getfile())
  File "C:\prg\Python22\lib\xmlrpclib.py", line 896, in parse_response
    return u.close()

```

```
File "C:\prg\Python22\lib\xmlrpc\lib.py", line 571, in close
    raise apply(Fault, (), self._stack[0])
Fault: <Fault 1: '__main__.ErrorNoHayExistencias:El encargo del articulo a sobrepasa las
existencias disponibles (10).>
```

Introspección

La especificación original de XML-RPC no ofrece ninguna posibilidad de descubrir qué funciones están disponibles, cómo se usan y obtener cualquier otro tipo de ayuda. Para cubrir esta necesidad, se propusieron tres nuevas funciones¹¹:

- **listMethods()**: devuelve una lista de cadenas de caracteres, una para cada nombre de función disponible para ser llamada a través de XML-RPC
- **methodSignature(nombre)**: devuelve una lista de posibles firmas para el nombre de la función especificada como argumento, es decir, los tipos de los argumentos, como por ejemplo: [[string, int, int], [array, int, int]].
- **methodHelp(nombre)**: devuelve la documentación para el nombre de función especificada mediante el argumento, si disponible. La especificación indica que la cadena puede contener etiquetas HTML.

Dado que el servidor simple XML-RPC de la librería no contiene esta funcionalidad, hemos de descargar e instalar el archivo `xmlrpc_registry.py`, referenciado desde la documentación de Python¹² como 'XML-RPC Hacks page'¹³.

Servidor simple de XML-RPC con inspección

Con esto podemos construir el siguiente servidor XML-RPC ampliado que redefine `register_function` y `register_instance` para contemplar la introspección:

```
# Servidor XML-RPC simple con introspección.
# requiere xmlrpc_registry.py http://xmlrpc-c.sourceforge.net/hacks/xmlrpc_registry.py
# HispaLinux Sept. 2003, erny@sicem.biz
# Es seguro usar: from IntrospectiveXMLRPCServer import *

import SimpleXMLRPCServer
import xmlrpc_registry
from xmlrpc_registry import INT, BOOLEAN, DOUBLE, STRING, DATETIME, BASE64, ARRAY, STRUCT

class IntrospectiveXMLRPCRequestHandler(SimpleXMLRPCServer.SimpleXMLRPCRequestHandler):
    """Para cada solicitud se crea una nueva instancia de esta clase.
    Véase: SocketServer.TCPServer"""
    def _dispatch(self, methodName, params):
        """Este método es llamado siempre."""
        return self.server.registry.dispatch_call(methodName, params)

class IntrospectiveXMLRPCServer(SimpleXMLRPCServer.SimpleXMLRPCServer):
    """Extendemos SimpleXMLRPCServer, para que register_function y
    register_method agreguen automáticamente firmas y doc strings.
    Como no podemos determinar la firma de las funciones en tiempo
    de ejecución, lo tenemos que poner manualmente, por ejemplo:
    def AClass:
        def metodo1(self, param1, param2):
            ...lo que sea...
            metodo1._signature=[ [STRING, STRING, INT] ]
    """
    def __init__(self, addr,
```

¹¹ <http://www.python.org/doc/current/lib/serverproxy-objects.html>

¹² <http://www.python.org/doc/current/lib/module-xmlrpc-lib.html>

¹³ <http://xmlrpc-c.sourceforge.net/hacks.php>

```

        requestHandlerClass=IntrospectiveXMLRPCRequestHandler, logRequests=1):
SimpleXMLRPCServer.SimpleXMLRPCServer.__init__(self, addr,
        requestHandler=requestHandlerClass, logRequests=logRequests)
# crear registro de funciones local.
self.registry=xmlrpc_registry.Registry()

def register_function(self, function, functionName=None,
        functionSignature=None, functionHelp=None):
    """ registrar función, signatura (si disponible) y doc string. """
    if not functionName: functionName = function.func_name
    if not functionSignature:
        try:
            functionSignature=function._signature
        except:
            functionSignature='undef'
    if not functionHelp: functionHelp=function.__doc__ or ""
    self.registry.add_method(functionName, function, functionSignature, functionHelp)

def register_instance(self, instance):
    """ register all todos los métodos de la instancia, signaturas y doc strings. """
    import types
    for methodName in self.methodList(instance, includePrivate=0):
        method = getattr(instance,methodName)
        try:
            methodSignature=method._signature
        except:
            methodSignature='undef'
        methodHelp=method.__doc__ or ""
        self.registry.add_method(methodName, method, methodSignature, methodHelp)

def methodList(instance, includePrivate=1): # metodo static
    """Devuelve una lista de métodos de la instancia.
    Se consideran metodos privados todos los que empiezan por _"""
    import types
    methodNames = []
    for symbol in dir(instance.__class__):
        if symbol[0]!="_" or includePrivate:
            if isinstance(getattr(instance,symbol),
                (types.MethodType, types.FunctionType)):
                methodNames.append(symbol)
    return methodNames
methodList=staticmethod(methodList)

if __name__=="__main__":
    def test():
        def echo(testArg):
            """ Simple echo service for debugging. returns the same argument. """
            return testArg
        echo._signature=[[STRING, STRING],[INT, INT]]

        class Class:
            """ Doc String of Class. """
            def sayHello(self, who):
                """ returns Hello <who>. """
                return "Hello %s" % who
            sayHello._signature=[[STRING, STRING]]

        instance=Class()
        s=IntrospectiveXMLRPCServer(('localhost',8001))
        s.register_instance(instance)
        s.register_function(echo)
        s.serve_forever()
    test()

```

Ejemplo de uso de introspección

Este fragmento de código probado con el intérprete de comando muestra el uso de la funcionalidad de introspección:

```
>>> from xmlrpclib import Server
>>> s=ServerProxy("http://localhost:8001")
>>> s.system.listMethods()
['echo', 'sayHello', 'system.listMethods', 'system.methodHelp', 'system.methodSignature',
'system.multicall']
>>> s.system.methodHelp("echo")
' Simple echo service for debugging. returns the same argument. '
>>> s.system.methodSignature("echo")
[['string', 'string'], ['int', 'int']]
>>> s.echo("Hola")
'Hola'
```

XML-RPC con Webware

Webware for Python¹⁴ es un conjunto de componentes para el desarrollo de aplicaciones basadas en Web.¹⁵ El componente más importante es WebKit que permite ejecutar Servlets en un entorno de servidor multi-hebra. Desgraciadamente, los procedimientos de instalación¹⁶ y las intrucciones de desarrollo¹⁷ no son triviales y están fuera del ámbito de este documento.

La clase fundamental que normalmente se usa para escribir aplicaciones Web es **HTTPServlet** o alguno de sus subclases, como **Page**. La persona que quiere crear un Servlet deriva de alguna de las anteriores clases y redefine alguno de los métodos que escriben el contenido tales como **writeBody**, **writeHTML** o **writeContent**.

WebKit dispone de soporte para XML-RPC a través de la clase **XMLRPCServlet**. Veamos un el ejemplo que viene con WebKit traducido al español y almacenado en el archivo **EjemploXMLRPC.py** del directorio Webware/WebKit/Examples, adicionalmente supongamos que el servidor Web ha sido configurado en el puerto 80 de manera que el contenido de Webware se accede a través de la ruta de /wk/... .

```
from WebKit.XMLRPCServlet import XMLRPCServlet

class EjemploXMLRPC (XMLRPCServlet):
    def exposedMethods(self):
        return ['multiplica', 'suma']

    def multiplica(self, x, y):
        return x * y

    def suma(self, x, y):
        return x + y

    def divide(self, *args):
        return reduce(operator.div, args)
```

En principio, todos los métodos de un XMLRPCServlet son privados. El método `exposeMethods` devuelve una lista de todos los métodos que se publican como procedimientos remotos. En el caso anterior, sólo **multiplica** y **suma** son métodos públicos.

Para probarlo arrancamos el intérprete interactivo de Python:

```
>>> from xmlrpclib import Server
>>> s=ServerProxy("http://localhost/wk/Examples/EjemploXMLRPC")
>>> s.multiplica(2,5)
10
>>> s.divide(10,2)
Traceback (most recent call last):
```

¹⁴ <http://webware.sourceforge.net/>

¹⁵ <http://webware.sourceforge.net/Webware-0.8.1/Docs/Overview.html>

¹⁶ <http://webware.sourceforge.net/Webware-0.8.1/WebKit/Docs/InstallGuide.html>

¹⁷ <http://webware.sourceforge.net/Webware-0.8.1/WebKit/Docs/UsersGuide.html>

```

File "<interactive input>", line 1, in ?
File "C:\prg\Python22\lib\xmlrpplib.py", line 821, in __call__
    return self.__send(self.__name, args)
File "C:\prg\Python22\lib\xmlrpplib.py", line 975, in __request
    verbose=self.__verbose
File "C:\prg\Python22\lib\xmlrpplib.py", line 853, in request
    return self.parse_response(h.getfile())
File "C:\prg\Python22\lib\xmlrpplib.py", line 896, in parse_response
    return u.close()
File "C:\prg\Python22\lib\xmlrpplib.py", line 571, in close
    raise apply(Fault, (), self._stack[0])
Fault: <Fault 1: 'Traceback (most recent call last):
File ".\WebKit\XMLRPCServlet.py", line 40, in respondToPost
response = self.call(method, *params)
File ".\WebKit\RPCServlet.py", line 15, in call
raise NotImplementedError, methodName
NotImplementedError: divide'>

```

Las diferencias con respecto al servidor simple de XML-RPC que viene con la librería estándar de Python 2.2 son:

- los servlets se ejecutan en un entorno multi-hebra, es decir, varias llamadas XML-RPC pueden ser respondidas a la vez.
- hay que especificar explícitamente cuáles de los métodos de un servlet se publican, en contraste del registro de una instancia que hace disponible, en principio, el acceso a todos sus métodos.
- registro de errores a un archivo para posterior revisión
- es necesario especificar la ruta completa (contexto/directorio/./Servlet).
- podría usarse la gestión de sesiones de WebKit, y la funcionalidad disponible de los otros componentes de Webware como UserKit o MiddleKit.

Para cambiar el comportamiento de las llamadas a todos los métodos de un servlet, como por ejemplo, recibir como primer elemento un identificador de sesión o una contraseña y agregar una comprobación de seguridad al principio de cada llamada, puede reescribirse el método **call**:

```

def call(self, idSesion, nombreMetodo, *args, **kwargs):
    if self.sesionEsValida(idSesion):
        return XMLRPCServlet.call(self, nombreMetodo, *args, **kwargs)
    else:
        raise SesionNoValida

```

XML-RPC con Zope

Zope es un servidor de aplicaciones muy popular, especialmente en el campo de la gestión de contenidos, dado su carácter gratuito y la facilidad para crear y gestionar contenido dinámico.

Su principio de funcionamiento es muy simple: traduce el acceso mediante rutas (de tipo URL) en llamadas a objetos. Por ejemplo, /carpeta1/carpeta2/documento se traduciría en algo como carpeta1.carpeta2.documento(), que posiblemente a su vez efectúa llamadas a otros objetos, para devolver un texto HTML. Cada método puede devolver texto, números o cualquier otro tipo de dato, como listas, etc. Este mecanismo hace que prácticamente todas la funcionalidad de los objetos de Zope puedan accederse usando solicitudes HTTP.

Dado que solicitudes XML-RPC son básicamente llamadas HTTP-Post, no es de extrañar que Zope también ofrece este tipo de acceso a sus objetos¹⁸.

¹⁸ <http://www.zope.org/>

Ejemplo simple de XML-RPC con Zope

Supongamos que tenemos Zope corriendo en el puerto 80 de 'localhost' y el siguiente script (Python) en la carpeta raíz con acceso público:

Nombre script: sumar

Lista parámetros: numeros

Título: sumar todos los numeros de una lista

Código:

```
suma=0
for num in numeros: suma+=num
return suma
```

Ejemplo de uso:

```
>>> from xmlrpclib import Server
>>> s=ServerProxy("http://localhost/")
>>> s.sumar([1, 2, 3])
6
>>> # probamos algunos metodos de la API de Zope
>>> s.sumar.title_or_id()
sumar todos los numeros de una lista
>>> s.sumar.getPhysicalPath()
['', 'sumar']
>>> s.sumar.document_src()
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "C:\prg\Python22\lib\xmlrpclib.py", line 821, in __call__
    return self.__send(self.__name, args)
  File "C:\prg\Python22\lib\xmlrpclib.py", line 975, in __request
    verbose=self.__verbose
  File "C:\prg\Python22\lib\xmlrpclib.py", line 848, in request
    headers
ProtocolError: <ProtocolError for servidor/: 401 Unauthorized>
```

Zope, XML-RPC y seguridad

Como se ve en el ejemplo anterior, la mayoría de los objetos y métodos en Zope están habitualmente protegidos. Vamos a verlo un poco más detalladamente:

```
>>> s=Servidor("http://localhost/", verbose=1)
>>> s.sumar.document_src()
connect: (servidor, 80)
send: 'POST / HTTP/1.0\r\n'
send: 'Host: servidor\r\n'
send: 'User-Agent: xmlrpclib.py/1.0.0 (by www.pythonware.com)\r\n'
send: 'Content-Type: text/xml\r\n'
send: 'Content-Length: 112\r\n'
send: '\r\n'
send: "<?xml
version='1.0'?>\n<methodCall>\n<methodName>sumar.document_src</methodName>\n<params>\n</p
arams>\n</methodCall>\n"
reply: 'HTTP/1.0 401 Unauthorized\r\n'
header: Server: Zope/(Zope 2.6.1 (binary release, python 2.1, win32-x86), python 2.1.3,
win32) ZServer/1.1b1
header: Date: Wed, 06 Aug 2003 18:20:00 GMT
header: WWW-Authenticate: basic realm="Zope"
header: Bobo-Exception-File: P:\Plone\Zope\lib\python\ZPublisher\HTTPResponse.py
header: Bobo-Exception-Type: Unauthorized
header: Content-Type: text/html
header: Connection: close
header: Bobo-Exception-Value: <strong>You are not authorized to access this
resource.</strong>
header: Etag:
header: Content-Length: 64
```

```

header: Bobo-Exception-Line: 667
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "C:\prg\Python22\lib\xmlrpplib.py", line 821, in __call__
    return self.__send(self.__name, args)
  File "C:\prg\Python22\lib\xmlrpplib.py", line 975, in __request
    verbose=self.__verbose
  File "C:\prg\Python22\lib\xmlrpplib.py", line 848, in request
    headers
ProtocolError: <ProtocolError for servidor/: 401 Unauthorized>

```

Vemos como Zope manda un conjunto de cabeceras, tal como el tipo de autenticación requerida (WWW-Authenticate). Para poder acceder a estos recursos, es necesario poder especificar nombre de usuario y contraseña. Dado que Zope usa el mismo mecanismo que las solicitudes HTTP normales, tenemos que mandar la siguiente línea de cabecera:

Authorization: Basic <nombre usuario>:<contraseña>

donde la parte **<nombre usuario>:<contraseña>** tiene que ir codificada en base64. La codificación es la parte más fácil:

```

from base64 import encodestring
function codificar(usuario, password):
    return encodestring("%s:%s" % (usuario, password))[:-1]

```

Más difícil parece modificar el código de *xmlrpplib* para incluir esta nueva cabecera. Para ello tenemos que redefinir la clase *Transport* que en su método **request** tiene las siguientes líneas:

```

...
    self.send_request(h, handler, request_body)
    self.send_host(h, host)
    self.send_user_agent(h)
    self.send_content(h, request_body)
...

```

Cada uno de **send_host**, **send_user_agent**, **send_content** escribe cabeceras (putheader) para mandarlo al servidor remoto. **send_content** adicionalmente termina la sección de cabeceras (endheaders) insertando una línea en blanco entre la sección de cabeceras y el propio mensaje xml que contiene la llamada y los argumentos. Si no queremos reescribir el método **request** de la clase *Transport* sólo podemos emplear **send_host** o **send_user_agent** para incluir la información requerida.

El siguiente código contenido en el archivo **xmlrpplib2.py**, ubicado en algún sitio donde Python lo pueda acceder (por ejemplo en el subdirectorio lib), redefine las clases originales de manera que se tengan en cuenta nombres de usuario y contraseñas:

```

# librería xmlrpplib con envío de cabeceras para seguridad
# HispaLinux Sept. 2003, erny@sicem.biz

from xmlrpplib import *
import base64

_Transport=Transport          # renombramos las clases originales
_SafeTransport=SafeTransport
_ServerProxy=ServerProxy

class TransportMixin:         # abstracción del código común de Transport y SafeTransport
    def __init__(self, username="", password=""):
        # (Safe)Transport no tiene método __init__, no requiere inicialización.
        self.username=username
        self.password=password

    def send_host(self, connection, host):
        self.super.send_host(self, connection, host) # llamada a la superclase.
        if self.username:
            userpass64=base64.encodestring("%s:%s" % (self.username, self.password))
            # dado que base64.encodestring agrega un caracter nueva línea al final,
            # se lo quitamos
            userpass64=userpass64[:-1]

```

```

        connection.putheader("Authorization","Basic %s" % userpass64)

# nuestras nuevas clases de transporte
class Transport(TransportMixin,_Transport):
    def __init__(self, username="", password=""):
        self.super=_Transport
        TransportMixin.__init__(self, username, password)

class SafeTransport(TransportMixin,_SafeTransport):
    def __init__(self, username="", password=""):
        self.super=_SafeTransport
        TransportMixin.__init__(self, username, password)

class ServerProxy(_ServerProxy):
    # reescribimos totalmente el metodo __init__ porque la falta de granularidad
    # no permite hacer cambios puntuales. Sólo hemos de proporcionar usuario
    # y password si estos han sidos especificados.
    def __init__(self, uri, username="", password="",
                 transport=None, encoding=None, verbose=0):
        # establish a "logical" server connection
        # get the url
        import urllib
        type, uri = urllib.splitttype(uri)
        if type not in ("http", "https"):
            raise IOError, "unsupported XML-RPC protocol"
        self.__host, self.__handler = urllib.splithost(uri)
        if not self.__handler:
            self.__handler = "/RPC2"

        if transport is None:
            if type == "https":
                transport = SafeTransport(username, password)
            else:
                transport = Transport(username, password)
        self.__transport = transport

        self.__encoding = encoding
        self.__verbose = verbose

Server = ServerProxy

```

De nuevo probamos la conexión con Zope, esta vez usando nuestro nuevo módulo xmlrpc1ib2, y suponnendo la existencia del usuario **usuario** con contraseña **password** que está autorizado para modificar el *script sumar*:

```

>>> from xmlrpc1ib2 import ServerProxy
>>> s = ServerProxy("http://localhost/", "usuario", "password")
>>> s.sumar.document_src()
'## Script (Python) "sumar"\n##bind container=container\n##bind context=context\n##bind
namespace=\n##bind script=script\n##bind
subpath=traverse_subpath\n##parameters=numeros\n##title=Sumar todos los
numeros\n##\nsuma=0\nfor num in numeros: suma+=num\nreturn suma\n'
>>> print _
## Script (Python) "sumar"
##bind container=container
##bind context=context
##bind namespace=
##bind script=script
##bind subpath=traverse_subpath
##parameters=numeros
##title=Sumar todos los numeros
##
suma=0
for num in numeros: suma+=num
return suma

```

Falta: HTTPS, lado cliente, lado servidor!

Desde el lado del cliente, los mecanismos para realizar llamadas mediante HTTPS son idénticas excepto que empiezan por `https://...` y que como primer argumento del objeto `ServerProxy` puede especificarse una tupla (`nombreHost`, `diccionario-certificados X-509`¹⁹).

¹⁹ <http://ist.uwaterloo.ca/security/ssl-pki/>

SOAP: Simple Object Access Protocol

Introducción

SOAP construye encima de XML-RPC. SOAP agrega a su precesor²⁰:

- tipos de datos definidos por la aplicación, a partir de tipos de datos básicos
- mensajes complejos con sobre (envelope), cabecera (header) y cuerpo.
- independencia del protocolo de transporte

Existen muchos debates acerca de la idoneidad de SOAP. La cuestión es si una aplicación determinada realmente necesita esta nueva funcionalidad o simplemente desea usar simples llamadas a procedimientos remotos. Con respecto a CORBA, SOAP tampoco resuelve la problemática de transacciones, limpieza de objetos huérfanos o transporte de identificadores de objetos.

Un punto de referencia para descargar implementaciones de SOAP para Python es <http://pywebsvcs.sourceforge.net/>, haciendo disponible diferentes versiones de SOAP con o sin soporte de WSDL (que se verá más adelante).

Descripción de la tecnología

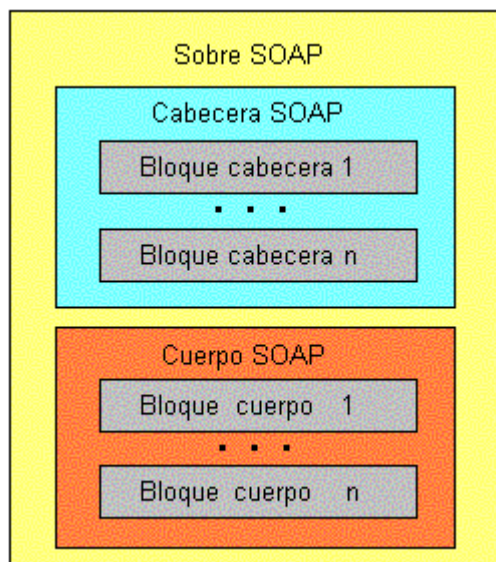
Al igual que XML-RPC, SOAP usa mensajes en formato XML. Sin embargo, además de HTTP y HTTPS también se consideran otros protocolos de transporte tales como SMTP, y la estructura de los mensajes es más compleja porque SOAP está pensado para otros escenarios además de las llamadas a procedimientos remotos.

En general, es posible mandar un mensaje SOAP desde un *nodo emisor* a un *nodo receptor* pasando por *nodos intermedios*, que procesan parte del mensaje para agregar información adicional. En caso de un buscador, por ejemplo, los nodos intermedios podrían realizar un análisis morfológico de la palabras y dar prioridad a los términos de búsqueda, mientras que otro nodo realizaría la propia búsqueda.

Esto significa que los nodos intermedios tienen que poder procesar parte del mensaje. Por esta razón, los mensajes SOAP han sido divididos en cabecera, parte procesada por todos los intermediarios, y cuerpo, parte destinado sólo al receptor final. El uso de la cabecera es opcional, bajando los requisitos para usos simples.

²⁰ <http://www-106.ibm.com/developerworks/xml/library/x-soapbx2/?open&l=136,t=gr,p=bullet>

La estructura resultante es la siguiente²¹:



Ejemplo de un mensaje SOAP:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/">
  <env:Header>
    <p:cabecerapedido xmlns:p="http://soap.ejemplo.org/pedido_cabecera"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
      env:mustUnderstand="true">
      <p:ref>2003/00058</p:ref>
      <p:fecha>2003-09-09</p:fecha>
    </p:cabecerapedido>
    <c:cliente xmlns:c="http://soap.ejemplo.com/clientes"
      env:role="http://www.w3.org/2003/05/soap-envelope/role/next"
      env:mustUnderstand="true">
      <c:nombre>Ernesto Revilla</c:nombre>
    </c:cliente>
  </env:Header>
  <env:Body>
    <p:cuerpopedido xmlns:p="http://soap.ejemplo.org/pedido_cuerpo">
      <p:fechaEntrega>2003-09-11</p:fechaEntrega>
      <p:direccionEnvio>
        <p:domicilio>c/ Alhóndiga 24</p:domicilio>
        <p:municipio>Granada</p:municipio>
        <p:cdgPostal>18003</p:cdgPostal>
      </p:direccionEnvio>
      <p:lineas>
        <p:linea>
          <p:refArticulo>5A630F</p:refArticulo>
          <p:unidades>22.5</p:unidades>
        </p:linea>
        <p:linea>
          <p:refArticulo>00053</p:refArticulo>
          <p:unidades>10</p:unidades>
        </p:linea>
      </p:lineas>
    </p:cuerpopedido>
  </env:Body>
</env:Envelope>
```

²¹ <http://www.w3.org/TR/2003/REC-soap12-part0-20030624/>

Espacios de nombres (namespaces) XML²²

El concepto de los espacios de nombres en XML es parecido al de Python y su cometido principal es evitar colisiones en los nombres de las etiquetas y los atributos. El escenario es un poco diferente: se trata que un mismo documento puede ser procesado por diferentes programas, donde cada software puede agregar, modificar o leer información específica. Mientras que los espacios de nombres de Python son dinámicos en el sentido que se puede agregar o redefinir cualquier identificador, los de XML normalmente no son modificables.

Definición: Un **espacio de nombres XML** es una colección de nombres identificado mediante una referencia URI (Uniform Resource Identifier = Identificador uniforme de recurso)²³ y usado en documentos XML para tipos de elementos (etiquetas) y nombres de atributos. Los espacios de nombres XML difieren de los espacios de nombres usados habitualmente en lenguajes de programación en que la versión para XML tiene una estructura interna y no es, matemáticamente hablado, un conjunto.

En el ejemplo anterior vemos como la segunda etiqueta especifica el espacio de nombres **env** mediante: **xmlns:env="http://www.w3.org/2003/05/soap-envelope"** . A partir de este momento, **env** que ligado al espacio de nombres indicado mediante el URI (<http://www.w3.org/2003/05/soap-envelope>), es decir, todas las etiquetas y atributos que empiecen por 'env:' han de estar definidos en el documento de especificación de espacio de nombres.

La estructura interna a la que se refiere la definición se corresponde a una definición de tipo que en XML se realiza mediante esquemas (Schema)²⁴. Este es un fragmento de la definición del espacio de nombres para SOAP:

```
<xs:element name="Envelope" type="tns:Envelope" />
<xs:complexType name="Envelope" >
  <xs:sequence>
    <xs:element ref="tns:Header" minOccurs="0" />
    <xs:element ref="tns:Body" minOccurs="1" />
  </xs:sequence>
  <xs:anyAttribute namespace="##other" processContents="lax" />
</xs:complexType>
```

En el fragmento puede verse cómo se define el tipo **Envelope** como secuencia de **Header** opcional y **Body** obligatorio. Más sobre esquemas XML vemos a continuación, en la definición de tipos complejos.

Paso de parámetros en SOAP

Con SOAP es necesario pasar los parámetros en el mismo orden que la signatura del método y con los nombres que le correspondan. Algunas veces, esto puede dar problemas porque los diccionarios de Python no necesariamente mantienen el orden en que se ha especificado sus entradas. Esto es una incompatibilidad entre Python y SOAP lo que en la práctica significa que puede haber implementaciones que no procesan correctamente las solicitudes de los clientes SOAP escritos con Python.

Tipos de datos

En principio, SOAP permite todos los tipos de datos especificados en el esquema XML asociado. Existen actualmente dos versiones, 1999 y 2001, con ligeras diferencias. La conversión de los tipos de Python a los tipos de SOAP / XML-Schema depende de la librería usada. Los conversión de los tipos más simples, como números enteros, números de punto flotante y cadenas

²² <http://www.w3.org/TR/1999/REC-xml-names-19990114/Overview.html>

²³ <http://www.faqs.org/rfcs/rfc2396.html>

²⁴ <http://www.w3.org/TR/xmlschema-0/>

de caracteres, incluso la de listas, es trivial. Pero para usar tipos de datos de XML-Schema es necesario el uso de funciones auxiliares.

SOAPpy

Esta biblioteca dispone de la funcionalidad necesaria para escribir servidores y clientes de servicios Web usando el protocolo SOAP. Para acceder a un servidor SOAP usamos la clase SOAPProxy (en vez de ServerProxy). En el ejemplo, accedemos a un servicio que nos proporciona la temperatura en Fahrenheit de una ciudad especificada mediante código postal

```
>>> from SOAPpy import SOAPProxy
>>> s = SOAPProxy("http://services.xmethods.net/soap/servlet/rpcrouter")
>>> s.getTemp('92612') # Obtener temperatura de Irvine / California
<Fault SOAP-ENV:Server.BadTargetObjectURI: Unable to determine object id from call: is
the method element namespaced?>
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
  File "P:\PYTHON22\Lib\site-packages\SOAPpy\Client.py", line 362, in __call__
    return self.__r_call(*args, **kw)
  File "P:\PYTHON22\Lib\site-packages\SOAPpy\Client.py", line 384, in __r_call
    self.__hd, self.__ma)
  File "P:\PYTHON22\Lib\site-packages\SOAPpy\Client.py", line 306, in __call
    raise p
faultType: <Fault SOAP-ENV:Server.BadTargetObjectURI: Unable to determine object id from
call: is the method element namespaced?>
```

Oops. Ha dado error, pero el servidor nos da una pista (' Unable to determine object id from call: is the method element namespaced?'). Tenemos que especificar el espacio de nombres 'urn:xmethods-Temperature':

```
>>> s = SOAPProxy("http://services.xmethods.net/soap/servlet/rpcrouter",
... namespace="urn:xmethods-Temperature")
>>> s.getTemp('92612') # Obtener temperatura de Irvine / California
68.0
>>> # lo convertimos a celsius:
>>> ( _ - 32) / 1.8
20.0
```

Para ver exáctamente qué información se manda y se recibe, activamos el indicador de depuración:

```
>>> from SOAPpy import Config
>>> Config.debug = 1
>>> s.getTemp('92612') # Obtener temperatura de Irvine / California
*** Outgoing HTTP headers *****
POST /soap/servlet/rpcrouter HTTP/1.0
Host: services.xmethods.net
User-agent: SOAPpy 0.10.2 (http://pywebsvcs.sf.net)
Content-type: text/xml; charset="UTF-8"
Content-length: 518
SOAPAction: ""
*****
*** Outgoing SOAP *****
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:getTemp xmlns:ns1="urn:xmethods-Temperature" SOAP-ENC:root="1">
      <_1 xsi:type="xsd:string">92612</_1>
    </ns1:getTemp>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*****
```

```

*** Incoming HTTP headers *****
HTTP/1.? 200 OK
Date: Tue, 19 Aug 2003 17:09:07 GMT
Server: Enhydra-MultiServer/3.1.1b1
Status: 200
Content-Type: text/xml; charset=utf-8
Servlet-Engine: Enhydra Application Server/3.1.1b1 (JSP 1.1; Servlet 2.2; Java 1.3.1_04;
Linux 2.4.7-10smp i386; java.vendor=Sun Microsystems Inc.)
Content-Length: 465
Set-Cookie: JSESSIONID=OEDR6EM0GwYH_sBTZsl6IV7U;Path=/soap
X-Cache: MISS from www.xmethods.net
Age: 6
*****
*** Incoming SOAP *****
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:getTempResponse xmlns:ns1="urn:xmethods-Temperature"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:float">73.0</return>
    </ns1:getTempResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
*****
73.0
>>> # la temperatura ha subido mientras estaba escribiendo esto.

```

Los atributos del elemento **SOAP-ENV:Envelope** son un poco confusos. Parece que el texto `SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"` es un poco redundante porque suponemos que los sobres siempre se codifican de esta manera. (De hecho SOAP 1.2 no permite especificar `encodingStyle` en el sobre.) Mucho más claro aparece en la respuesta. Dado que no se trata de escribir una librería capaz de enviar mensajes SOAP sino de usarla, no entramos en mayor debate del mensaje sino destacamos sólo lo importante:

- tanto en el envío como en la recepción **se especifican los tipos de datos** usados
- **ns1** se utiliza genéricamente como el espacio de nombres del método
- se llama a **getTemp** y se le pasa un parámetro sin indicar un nombre
- la respuesta es enviada como **getTempResponse**
- el valor es devuelto mediante la etiqueta **return**

Tipos de datos simples

La traducción de cadenas de caracteres, enteros y números de punto flotante es automática. Para utilizar otros tipos nativos de SOAP esta librería dispone de un submódulo `Types` para este propósito:

```

>>> from SOAPpy import Types
>>> miNum = Types.unsignedIntType(55)
>>> miNum
<SOAPpy.Types.unsignedIntType at 17907688>
>>> miNum == 55
0
>>> int(miNum) == 55
Traceback (most recent call last):
  File "<interactive input>", line 1, in ?
AttributeError: unsignedIntType instance has no attribute '__int__'

```

Una vez que hemos convertido el argumento a un tipo SOAP ya no podemos realizar comparaciones. La expresión `Types.unsignedIntType(55) == Types.unsignedIntType(55)` devuelve 0. Podemos acceder a la expresión original utilizada para construir el tipo mediante el miembro `_data`.

Tipos de datos complejos

Dado que SOAP permite el uso de tipos de datos complejos a través de la definición de esquemas, SOAPpy usa diccionarios de objetos donde cada elemento puede ser a su vez un dato complejo (diccionario) o simple. Esto permite crear mensajes como el que hemos visto al principio del capítulo:

```
from SOAPpy import Types
direccion = Types.structType( data = { 'domicilio': u'c/ Alhóndiga 24', 'municipio':
'Granada',
'cdgPostal': '18003', '_typeName': 'direccionEnvio' } )
lineas = [ {'refArticulo': 'pantalon', 'unidades': 20}, {'refArticulo': 'camisa',
'unidades': 15} ]
lineasSoap = []
for linea in lineas:
    linea['_typeName'] = 'linea'
    lineasSoap.append( Types.structType( data = linea ) )
lineasSoap = Types.arrayType( data = lineasSoap )
po = Types.structType( data = {"numPedido": "1234", "fecha": Types.dateTimeType(),
"direccionEnvio": dirEnvio, "lineas": lineasSoap} )
```

Fuentes

XML

- Jones, Chritsopher A.; Drake Jr., Fred L.: Python & XML, O'Reilly 2002, pp. 1-45

XML-RPC

- Python 2.2 Library Reference: xmlrpclib, SimpleXMLRPCServer
- www.xmlrpc.com

SOAP

- Jones, Chritsopher A.; Drake Jr., Fred L.: Python & XML, O'Reilly 2002, pp. 204-225

Notas para seguir el documento

Documentos de intercambio para e-business:

- * ebXML: Electronic Business XML
- * xCBL: XML Common Business Library, <http://www.xcbl.org/>
- * cXML: commerce XML, <http://www.cxml.org/>
- * UBL: Universal Business Language, <http://www.oasis-open.org/committees/ubl/>

Servicios Web:

- * XML-RPC
- * SOAP
- * WDSL
- * UDDI
- * XML Schemas

Otros:

- * RPC, XDR (eXternal Data Representation)
- * CORBA
- * RDF
- * UML -> XMI
- * OMG's Model Driven Architecture (MDA) -> XMI, intercambio de modelos
- * Pyro

Ejemplos de Servicios Web:

- * Autenticación
- * comprobación de crédito / tarjeta de crédito
- * valores de la bolsa
- * pedidos

Servicios Web complejos pueden llamar a servicios Web simples para obtener un resultado

Ventajas:

- * construye encima de estándares populares:
 - * http
 - * xml (SOAP, WDSL, UDDI)
- * mensajes legibles

- * fácil de usar???
- * hace falta relativamente poca infraestructura para ponerlo a funcionar
- * implementaciones económicas o libres

Desventajas:

- * incompatibilidades entre mensajes SOAP
- * inmaduro, es muy reciente
- * lento
- * pocas implementaciones económicas o libres

Definición Web Service (Servicio Web):

- * conjunto de funciones que se pueden llamar remotamente mediante tecnologías Web (HTTP, XML) y que tratan un tema determinado.
- * Ejemplo: Pedidos electrónicos. Funciones: dar de alta, comprobar estado, cancelar

Entornos:

- * xmlrpclib y SimpleXMLRPCServer (XML-RPC)
- * Webware (XML-RPC)
- * CherryPy(XML-RPC)
- * Zope (XML-RPC)
- * SOAPpy (SOAP)
- * Siena (?)

Aunque la temática de la computación distribuida es compleja (fiabilidad, seguridad, transacciones, etc.) empezamos ignorando inicialmente esta temática, con un ejemplo simple: XML-RPC