Distributed Applications

with

Python

Dr Duncan Grisby
duncan@grisby.org

Part one

Technologies

Outline

1. Introduction
2. A simple example
3. XML-RPC details
4. CORBA details
5. Comparisons and summary

About me

- BA and PhD at the University of Cambridge Computer Laboratory.
- Worked at AT&T Laboratories Cambridge before its closure in April 2002.
- Founder of Apasphere Ltd. 

  - Interested in contract offers. . .
- Main author of omniORBpy
  - but I'm trying very hard to be unbiased.

Introduction

1. What is a distributed system?
2. Why would we want one?
3. Distributed system technologies
4. XML-RPC
5. SOAP
6. CORBA

What is a distributed system?

- A system in which not all parts run in the same address space. . .
  - and normally across more than one computer.
- Complex
  - concurrency
  - latency
  - nasty failure modes
  - . . .

So why bother?

- There’s more than one computer in the world.
- They solve some real problems
  - Distributed users
  - Load balancing
  - Fault tolerance
  - Distributed computation
  - …
- It’s a challenge.

Technologies

- Sockets
- RPC
  - Sun RPC, DCE, **XML-RPC**, **SOAP**
- Single language distributed objects
  - Java RMI, DOPY, Pyro
- Cross-language distributed objects
  - DCOM, **CORBA**
- Message-oriented middleware, mobile agents, tuple spaces, …

RPC — Remote Procedure Call

- Model networked interactions as procedure calls.
  - Natural model for many kinds of application.
  - Totally inappropriate for some things.
- Considered at least as early as 1976
  - White, J.E., *A high-level framework for network-based resource sharing*, Proceedings of the National Computer Conference, June 1976.
- Requires: server addressing model, transport protocol, data type *marshalling*.

Object Oriented RPC

- Obvious extension of RPC to support objects.
  - Exactly analogous to the difference between procedural and object oriented programming.
- In a remote method call, choice of object is implicit in the *object reference*.
- Object references are first class data types: they can be sent as method arguments.
- Requires: object addressing model, transport protocol, marshalling.

What is XML-RPC?

- `www.xmlrpc.com`
- Very simple RPC protocol
  - HTTP for server addressing and transport protocol.
  - XML messages for data type marshalling.
  - Limited range of simple types.
- Stable specification
  - Perhaps too stable.
- Implementations in many languages.
- Fork from an early version of SOAP…

What is SOAP?

- It depends who you ask!
  - Started life as an RPC protocol using HTTP/XML.
  - Moving away from that, towards a general message framing scheme.
- As of SOAP 1.2, no longer stands for ‘Simple Object Access Protocol’.
- `www.w3c.org/2002/ws/`
- A plethora of related specifications:
  - XML Schema, WSDL, UDDI, . . .
- Specification and implementations in flux.

Schemas, WSDL and UDDI

- XML Schema
  - `www.w3.org/XML/Schema`
  - Used in SOAP to define types.
- WSDL — Web Services Description Language
  - `www.w3.org/TR/wsdl`
  - Wraps up information about types, messages and operations supported by a service, and where to find the service.
- UDDI — Universal Description, Discovery and Integration
  - `www.uddi.org`
  - Framework for describing, finding services.

What is CORBA?

Common Object Request Broker Architecture.

- i.e. a common architecture for object request brokers.
- A framework for building *object oriented* distributed systems.
- Cross-platform, language neutral.
- Defines an object model, standard language mappings, . . .
- An extensive open standard, defined by the Object Management Group.
  - `www.omg.org`

Object Management Group

- Founded in 1989.
- The world's largest software consortium with around 800 member companies.
- Only provides *specifications*, not implementations.
- As well as CORBA core, specifies:
  - Services: naming, trading, security, . . .
  - Domains: telecoms, health-care, finance, . . .
  - UML: Unified Modelling Language.
  - MDA: Model Driven Architecture.
- All specifications are available for free.

Python XML-RPC

- xmlrpclib
  - `www.pythonware.com/products/xmlrpc/`
  - Part of Python standard library since 2.2.
  - Very Pythonic and easy-to-use.

Python SOAP

- SOAP.py
  - `pywebsvcs.sourceforge.net`
  - Similar in style to xmlrpclib.
  - Not actively maintained.
- ZSI, Zolera SOAP Infrastructure
  - `pywebsvcs.sourceforge.net` again.
  - Most flexible and powerful option.
  - Currently not particularly Pythonic.

Python SOAP cont'd

- SOAPy
  - `soapy.sourceforge.net`
  - Supports WSDL, XML Schema
  - Client side only.
- 4Suite SOAP
  - `www.4suite.org`
  - Part of 4Suite Server.
  - From the 'SOAP as message framing' camp.
  - No RPC.

Python CORBA

- omniORBpy
  - `omniorb.sourceforge.net`
  - Based on C++ omniORB. Multi-threaded.
  - Most complete and standards-compliant.
- orbit-python
  - `orbit-python.sault.org`
  - Based on C ORBit. Single-threaded.
- Fnorb
  - `www.fnorb.org`
  - Pure Python (recent development).
  - Dead for a long time.
  - Newly open source (Python style).

A simple example

1. Specification
2. XML-RPC implementation
3. SOAP implementation
4. CORBA implementation
5. Comparison

Specifi cation

- We want an ‘adder’ service with operations:
  - `add`: add two integers.
  - `add_many`: take a list of integers and return their sum.
  - `accumulate`: add a single argument to a running total, return the new total.
  - `reset`: reset the running total to zero.

XML-RPC server

```
#!/usr/bin/env python
import operator, xmlrpclib, SimpleXMLRPCServer

class Adder_impl:
    def __init__(self):
        self.value = 0

    def add(self, a, b):
        return a + b

    def add_many(self, a_list):
        return reduce(operator.add, a_list, 0)

    def accumulate(self, a):
        self.value += a
        return self.value

    def reset(self):
        self.value = 0
        return xmlrpclib.True

adder  = Adder_impl()
server = SimpleXMLRPCServer.SimpleXMLRPCServer(("", 8000))
server.register_instance(adder)
server.serve_forever()
```

XML-RPC client

```
>>> import xmlrpclib
>>> adder = xmlrpclib.Server("http://server.host.name:8000/")
>>> adder.add(123, 456)
579
>>> adder.add("Hello ", "world")
’Hello world’
>>> adder.add_many([1,2,3,4,5])
15
>>> adder.add_many(range(100))
4950
>>> adder.accumulate(5)
5
>>> adder.accumulate(7)
12
>>> adder.reset()
<Boolean True at 819a97c>
>>> adder.accumulate(10)
10
>>> adder.accumulate(2.5)
12.5
```

XML-RPC request

```
POST / HTTP/1.0
Host: pineapple:8000
User-Agent: xmlrpclib.py/1.0b4 (by www.pythonware.com)
Content-Type: text/xml
Content-Length: 191

<?xml version=’1.0’?>
<methodCall>
<methodName>add</methodName>
<params>
<param>
<value><int>123</int></value>
</param>
<param>
<value><int>456</int></value>
</param>
</params>
</methodCall>
```

XML-RPC response

```
HTTP/1.0 200 OK
Server: BaseHTTP/0.2 Python/2.2c1
Date: Thu, 28 Feb 2002 10:47:05 GMT
Content-type: text/xml
Content-length: 123

<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><int>579</int></value>
</param>
</params>
</methodResponse>
```

XML-RPC notes

- We didn’t have to tell XML-RPC the names of the functions, or their argument types.
  - Dynamic dispatch/typing just like Python.
  - Not necessarily a good thing in a distributed system. . .
- XML-RPC has no equivalent of `None`.
  - `reset()` has to return something.

SOAP server (SOAP.py)

```
#!/usr/bin/env python
import operator, SOAP

class Adder_impl:
    def __init__(self):
        self.value = 0

    def add(self, a, b):
        return a + b

    def add_many(self, a_list):
        return reduce(operator.add, a_list, 0)

    def accumulate(self, a):
        self.value += a
        return self.value

    def reset(self):
        self.value = 0

adder  = Adder_impl()
server = SOAP.SOAPServer(("", 8000))
server.registerObject(adder)
server.serve_forever()
```

SOAP client

```
>>> import SOAP
>>> adder = SOAP.SOAPProxy("http://server.host.name:8000/")
>>> adder.add(123, 456)
579
>>> adder.add("Hello ", "world")
’Hello world’
>>> adder.add_many([1,2,3,4,5])
15
>>> adder.add_many(range(100))
4950
>>> adder.accumulate(5)
5
>>> adder.accumulate(7)
12
>>> adder.reset()
>>> adder.accumulate(10)
10
>>> adder.accumulate(2.5)
12.5
```

SOAP request

```
POST / HTTP/1.0
Host: pineapple:8000
User-agent: SOAP.py 0.9.7 (actzero.com)
Content-type: text/xml; charset="UTF-8"
Content-length: 492
SOAPAction: ""

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xm
lsoap.org/soap/encoding/" xmlns:SOAP-ENC="http://schemas.xml
soap.org/soap/encoding/" xmlns:xsi="http://www.w3.org/1999/X
MLSchema-instance" xmlns:SOAP-ENV="http://schemas.xmlsoap.or
g/soap/envelope/" xmlns:xsd="http://www.w3.org/1999/XMLSchem
a">
<SOAP-ENV:Body>
<add SOAP-ENC:root="1">
<v1 xsi:type="xsd:int">123</v1>
<v2 xsi:type="xsd:int">456</v2>
</add>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP response

```
HTTP/1.0 200 OK
Server: <a href="http://www.actzero.com/solution.html">SOAP.
py 0.9.7</a> (Python 2.2c1)
Date: Thu, 28 Feb 2002 11:07:38 GMT
Content-type: text/xml; charset="UTF-8"
Content-length: 484

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xm
lsoap.org/soap/encoding/" xmlns:SOAP-ENC="http://schemas.xml
soap.org/soap/encoding/" xmlns:xsi="http://www.w3.org/1999/X
MLSchema-instance" xmlns:SOAP-ENV="http://schemas.xmlsoap.or
g/soap/envelope/" xmlns:xsd="http://www.w3.org/1999/XMLSchem
a">
<SOAP-ENV:Body>
<addResponse SOAP-ENC:root="1">
<Result xsi:type="xsd:int">579</Result>
</addResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP notes

- Dynamic dispatch/typing like XML-RPC.
- WSDL would allow us to specify function names and types.
  - Except that none of the Python SOAP implementations support it fully.
- SOAP *does* have the equivalent of `None`.
- The SOAP encoding is much bigger and more complex than the XML-RPC encoding.

CORBA interface

- Types and interfaces must be defined.
  - CORBA Interface Definition Language, IDL.
  - Serves as formal documentation for the service, too.
  - Can be avoided if there's a *really* good reason.

```
1  module Snake {
2    interface Adder {
3      typedef sequence<long> LongSeq;
4
5      long add(in long a, in long b);
6      long add_many(in LongSeq a_list);
7      long accumulate(in long a);
8      void reset();
9    };
10 };
```

CORBA server

```
#!/usr/bin/env python
import sys, operator, CORBA, Snake__POA

class Adder_impl(Snake__POA.Adder):
    def __init__(self):
        self.value = 0

    def add(self, a, b):
        return a + b

    def add_many(self, a_list):
        return reduce(operator.add, a_list, 0)

    def accumulate(self, a):
        self.value += a
        return self.value

    def reset(self):
        self.value = 0

orb = CORBA.ORB_init(sys.argv)
poa = orb.resolve_initial_references("RootPOA")
obj = Adder_impl()._this()
print orb.object_to_string(obj)
poa._get_the_POAManager().activate()
orb.run()
```

CORBA client

```
>>> import CORBA, Snake
>>> orb = CORBA.ORB_init()
>>> obj = orb.string_to_object("IOR:0100...")
>>> adder = obj._narrow(Snake.Adder)
>>> adder.add(123, 456)
579
>>> adder.add("Hello ", "world")
Traceback (most recent call last): ...
CORBA.BAD_PARAM: Minor: BAD_PARAM_WrongPythonType, COMPLETED_NO.
>>> adder.add_many([1,2,3,4,5])
15
>>> adder.add_many(range(100))
4950
>>> adder.accumulate(5)
5
>>> adder.accumulate(7)
12
>>> adder.reset()
>>> adder.accumulate(10)
10
```

CORBA request/response

- CORBA uses an efficient binary format.

  Request:

```
4749 4f50 0102 0100 3400 0000 0600 0000 GIOP....4.......
0300 0000 0000 0000 0e00 0000 fe25 177e .............%.~
3c00 0032 7500 0000 0000 0000 0400 0000 <..2u............
6164 6400 0000 0000 7b00 0000 c801 0000 add.....{.......
```

  Response:

```
4749 4f50 0102 0101 1000 0000 0600 0000 GIOP............
0000 0000 0000 0000 4302 0000           ........C...
```

- Tools like Ethereal (`www.ethereal.com`) will pick it apart if you need to know what it means.

XML-RPC details

1. Types
2. Faults
3. Clients and servers
4. Extensions

XML-RPC types

- Boolean
  - `xmlrpclib.True` or `xmlrpclib.False`
- Integers
  - Python int type.
- Floating point
  - Python float type.
  - Beware rounding errors!
- Strings
  - Python string type.
  - ASCII only.

XML-RPC types

- Array
  - Python sequence type (list, tuple) containing ‘conformable’ values.
- Struct
  - Python dictionary with string keys, ‘conformable’ values.
- Date
  - `xmlrpclib.DateTime` instance.
  - Construct with seconds since epoch, time tuple, ISO 8601 string.
- Binary
  - `xmlrpclib.Binary` instance.
  - Construct with string, read from `data`.

XML-RPC faults

- Any server function can raise `xmlrpclib.Fault` to indicate an error.
  - Constructor takes integer fault code and a human-readable fault string.
  - Access with `faultCode` and `faultString`.
  - Uncaught Python exceptions in server functions are turned into Faults.
- The system may also raise `xmlrpclib.ProtocolError` if the call failed for some HTTP/TCP reason.

XML-RPC clients

- Clients create a proxy to a server:

```
proxy = xmlrpclib.ServerProxy("http://host.name:[port][/path]")
```

- Method names may contain dots:

```
a = proxy.foo()
b = proxy.bar.baz.wibble()
```

- https accepted if your Python has SSL support:

```
proxy = xmlrpclib.ServerProxy("https://host.name:[port][/path]")
```

XML-RPC servers

- SimpleXMLRPCServer included in Python 2.2:

```
server = SimpleXMLRPCServer.SimpleXMLRPCServer(("", port))
```

  - Usually specify empty string as host name. Use specific interface name/address to restrict calls to a particular interface.

- Register an instance

```
instance = MyServerClass()
server.register_instance(instance)
```

  - All of instance's methods available (except those prefixed with '_').
  - Sub-instances for dotted method names.
  - Only one instance can be registered.

XML-RPC servers

- Instance with a dispatch method:

```
class MyServer:
    def _dispatch(method, params):
        print "The method name was", method
        # Do something to implement the method...
```

- Register separate functions:

```
server.register_function(pow)

def doit(a, b): return a - b
server.register_function(doit, "subtract")
```

XML-RPC extensions

- `www.xmlrpc.com/directory/1568/services/xmlrpcExtensions`
- `system.listMethods`
  - return list of available functions.
- `system.methodSignature`
  - return the signature of the specified method, as a list of strings.
- `system.methodHelp`
  - return a help string for the specified method.
- `system.multiCall`
  - call a list of methods in sequence, returning all the results.

CORBA details

1. IDL and its Python mapping
2. CORBA object model
3. Object Request Broker
4. Portable Object Adapter

Interface Definition Language

- IDL forms a ‘contract’ between the client and object.
- Mapped to the target language by an *IDL compiler*.
- Strong typing.
- Influenced by C++ (braces and semicolons — sorry!).

```
module Snake {
  interface Adder {
    long accumulate(in long a);
    void reset();
  };
};
```

IDL Facilities

- All types and interfaces are specified in IDL.
- Base types:
  - integers, floating point, strings, wide strings.
- Constructed types:
  - enumerations, sequences, arrays, structures, discriminated unions, fixed point, interfaces.
- Interfaces:
  - operations, attributes, exceptions.
- Dynamic types:
  - Any, TypeCode.

IDL Example

```
module Example {

  struct Person {
    string name;
    unsigned short age;
  };

  enum DwellingKind { house, flat, cottage, castle };

  struct Dwelling {
    DwellingKind kind;
    Person owner;
    unsigned long number_of_rooms;
  };

  interface Auction {
    readonly attribute Dwelling lot;
    readonly attribute float high_bid;
    boolean bid(in Person who, in float amount);
  };

  interface AuctionHouse {
    Auction SellDwelling(in Dwelling to_sell, in float reserve);
  };
};
```

IDL to Python

- Standard Python language mapping:
  - `www.omg.org/technology/documents/formal/python_language_mapping.htm`
- Map IDL to Python with an *IDL compiler*. . .

```
$ omniidl -bpython example.idl
```

- Use the mapped types from Python. . .

```
>>> import Example
>>> fred = Example.Person("Fred Bloggs", 42)
>>> residence = Example.Dwelling(Example.cottage, fred, 3)
>>> residence.number_of_rooms
3
>>> auctioneer = # Get AuctionHouse object from somewhere
>>> auction = auctioneer.SellDwelling(residence, 1000.0)
>>> auction.bid(Example.Person("Joe Smith", 28), 2000.0)
>>> auction._get_high_bid()
2000.0
```

ORB and POA

- The Object Request Broker (ORB) holds everything together.
  - Not a stand-alone process—library code in all CORBA applications.
  - Provides basis for network-transparency, object model, etc.
- The Portable Object Adapter (POA) supports server code.
  - Supports activation of *servants*—i.e. implementation objects.
  - On-demand activation, default servants, flexible servant locators.

Standard CORBA services

- Naming
  - Tree-based hierarchy of named objects.
  - Supports federation.
- Notification
  - Asynchronous event filtering, notification.
- Interface repository
  - Run-time type discovery.
- Security
  - Encryption, authentication, authorisation, non-repudiation. . .
- Object trading, Transaction, Concurrency, Persistence, Time, . . .

Part two

Solving real problems

Common Problems

1. Finding services/objects
2. Transferring bulk data
3. Event notification
4. State and session management

Finding things

- Low-tech
  - Hard-coded URIs.
  - Write URIs / CORBA IORs to a file.
- Look-up by name
  - CORBA Naming service.
  - UDDI.
  - Ad-hoc name service.
- Look-up by properties
  - CORBA Trader service.
  - UDDI.
  - How do you know how to use it once you've got it?

Bulk data

- Lists / sequences
  - Simple, but can't cope with *really* large items.
- Iterator pattern in CORBA.

```
struct GameInfo { string name; Game obj; };
typedef sequence <GameInfo> GameInfoSeq;

interface GameFactory {
  ...
  GameInfoSeq listGames(in unsigned long how_many,
                        out GameIterator iter);
};
interface GameIterator {
  GameInfoSeq next_n(in unsigned long how_many,
                     out boolean more);
  void destroy();
};
```

- Socket transfer, FTP, etc.

Event notifi cation

- Blocking calls
  - Return when event occurs.
  - Interacts badly with timeouts.
- Callbacks
  - Service calls client when event occurs.
  - Firewall issues (CORBA bidir GIOP).
  - Tricky with web services.
- CORBA Event / Notification services
  - Push or pull transmission and reception.
  - Event filtering.
  - Manage scalability issues.
- MOM: IBM MQSeries, MSMQ, . . .

State and session management

- How do you create and track server-side state?
  - Don’t if you can help it!
  - CORBA Factory pattern.
  - RPC uses explicit cookies to identify state.
- How do you get rid of state?
  - Distributed garbage collection is *hard*!
  - No complete solution.
  - Must think about it on a per-application basis.
  - Reference counting and pinging, evictor pattern, timeouts, …

Conclusion

1. Comparisons
2. My recommendations
3. General hints
4. Further resources

Comparisons

- Like Python itself, XML-RPC and SOAP use dynamic typing.
  - Good for fast prototyping. . .
  - . . . but can you *really* trust your clients?
  - Distribution turns a debugging issue into a security issue.
  - Robust code has to check types everywhere.
- CORBA uses static interfaces and typing.
  - Have to specify interfaces in advance.
  - CORBA runtime checks types for you.
  - You have to document the interfaces anyway.
  - `Any` provides dynamic typing if you need it.

Comparisons

- XML-RPC and SOAP only specify transfer syntax.
  - Different implementations use different APIs.
  - Not an issue with Python XML-RPC since everyone uses xmlrpclib.
  - Definitely an issue with SOAP.
- CORBA has standard language mappings and object model.
  - Python source code is portable between different Python ORBs.
  - Object model and API is the same for all languages.

Comparisons

- XML-RPC and SOAP are *procedural*
  - Addressing on a per-server basis.
  - No implicit state in function calls.
  - Using explicit state in all calls can become tricky.
- CORBA is *object-oriented*
  - Object references are first-class data types.
  - Application entities can be modelled as objects.
  - Managing large numbers of objects can be tricky.

Comparisons

- CORBA uses a compact binary format for transmission.
  - Efficient use of bandwidth.
  - Easy to generate and parse.
- XML-RPC and SOAP use XML text.
  - Egregious waste of bandwidth.
  - Easy-ish to generate, computationally expensive to parse.
  - ‘Easy’ for a human to read
    - not this human!
- CORBA is 10–100 times more compact, 100–500 times faster.

My recommendations

- Use XML-RPC if
  - your requirements are *really* simple.
  - performance is not a big issue.
- Use CORBA if
  - object orientation and complex types are important.
  - interoperability is important.
  - performance is important.
  - CORBA’s services solve many of your problems.

My recommendations

- Use SOAP if
  - you like tracking a moving ‘standard’ :-)
  - you want to be buzzword-compliant.
- Use sockets if
  - you need to stream binary data.
  - you can’t afford *any* infrastructure.
- Use something else if
  - it fits neatly with your application.
- Use a combination of things if
  - it makes sense to do so.

General hints

- Design for distribution.
  - Think carefully about latency.
  - Often better to send data which may not be needed than to have fine-grained interfaces.
- Use exceptions wisely (if the platforms provides them).
- Avoid generic interfaces (e.g. ones which use CORBA Any) if possible.
- Don’t forget security requirements!
- Write your code in Python!

Further resources

- ‘Programming Web Services with XML-RPC’, by Simon St. Laurent, Joe Johnston and Edd Dumbill. O’Reilly.
- ‘Advanced CORBA Programming with C++’, by Michi Henning and Steve Vinoski. Addison-Wesley.
  - Don’t be put off by the C++ in the title — most of the content is applicable to any language.
  - Besides, it’s fun to see how much harder things are for C++ users.

CORBA resources

- Python CORBA tutorial

  `www.grisby.org/presentations/py10code.html`

- CORBA IDL to Python language mapping,

  `www.omg.org/technology/documents/formal/python_language_mapping.htm`

- CORBA specifications,

  `www.omg.org/technology/documents/`

Conclusion

- There are a lot of options out there.
- Despite the web services hype, CORBA and other established technologies are the best solution to many real-world problems.
- The value of web services is not as a replacement for CORBA, but an addition.
- Web services proponents could learn a lot from CORBA, if only they looked.