Distributed Applications

with

Python

Dr Duncan Grisby
duncan@grisby.org

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.
- Recent work has been on CORBA systems — ways to make CORBA easier to use.
- Main author of omniORBpy
  - but I’m trying very hard to be unbiased.
- AT&T Laboratories Cambridge is closing at the end of April.
  - Are you hiring?

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
  - `www.omniorb.org/omniORBpy`
  - 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`
  - Mostly Python, with a small amount of C.
  - 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

Specification

- 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.

CORBA notes

- CORBA objects are addressed using an IOR, Interoperable Object Reference.
  - `orb.object_to_string()` converts an IOR to a string form:

    ```
    IOR:010000001400000049444c3a536e616b652f41646465723a312e3000
    0100000000000000400000000101000110000006d792e7365637265742e
    7365727665720000d20420000000576173206974207 6f72746820747970
    696e67207468617420494f5220696e3f
    ```

  - Applications almost never deal with IORs directly.
  - Object references are normally received from other objects, like the Naming service.
- The `_narrow()` call checked that the object really was an Adder.
  - Often no need to narrow.

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.

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.Server("http://some.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.Server("https://some.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

IDL practicalities

- IDL files must end with `.idl` (although in most circumstances it doesn't matter).
- Written in ISO 8859-1 (Latin-1). Identifiers must be ASCII.
- Files are run through the C++ pre-processor
  - `#include`, `#define`, `//`, `/* */`, etc.
- Processed with an *IDL compiler*, e.g. omniidl, fnidl.
  - Resulting in *stubs* and *skeletons*.
- Case sensitive, but different capitalisations collide.
  - e.g. `attribute string String;` is invalid.
- Scoping rules similar (but not identical) to C++.

Simple types

IDL type	Meaning		Python mapping
`boolean`	TRUE or FALSE		int
`octet`	8-bit	unsigned	int
`short`	16-bit	signed	int
`unsigned short`	16-bit	unsigned	int
`long`	32-bit	signed	int
`unsigned long`	32-bit	unsigned	long
`long long`	64-bit	signed	long
`unsigned long long`	64-bit	unsigned	long
`float`	32-bit	IEEE float	float
`double`	64-bit	IEEE float	float
`long double`	$\geq$80-bit	IEEE float	CORBA.long_double

Textual types

IDL	Meaning	Python
`char`	8-bit ISO 8859-1 character.	string (length 1)
`string`	String of ISO 8859-1 characters. – no embedded nulls. – `string<`*bound*`>` is a *bounded* string.	string
`wchar`	Unicode character.	unicode (length 1)
`wstring`	Unicode string. – no embedded nulls. – `wstring<`*bound*`>` is a *bounded* wstring.	unicode

- In fact, any code set can be used, not just ISO 8859-1 and Unicode.
- ORBs negotiate translation between code sets.

Enumerations

- Simple list of identifiers.
- Only operation is comparison between values.
- Do not create a new naming scope!

IDL

```
module M {
  enum colour { red, green, blue, orange };
  enum sex { male, female };
  enum fruit { apple, pear, orange };  // Clash! orange redefined!
  const colour nice = red;
  const colour silly = male;  // Error!
};
```

Python

```
>>> choice = M.red   # Not M.colour.red
>>> choice == M.red
1
>>> choice == M.green
0
>>> choice == M.male
0
```

Structures

- Same idea as a C struct.
- Form a new naming scope.
- Structs can be nested.

IDL

```
module M {
  struct Person {
    string name;
    unsigned short age;
  };
};
```

Python

```
>>> me = M.Person("Duncan", 27)
>>> me.name
'Duncan'
>>> me.age = me.age + 1
```

Unions

- Consist of a *discriminator* and a *value*.
- Discriminator type can be integer, boolean, enum, char.
- More options than shown here.

IDL

```
module M {
  union MyUnion switch (long) {
    case 1:  string  s;
    case 2:  double  d;
    default: boolean b;
  };
};
```

Python

```
>>> u = M.MyUnion(s = "Hello")
>>> u.s
'Hello'
>>> u.d          # Raises a CORBA.BAD_PARAM exception.
>>> u.d = 3.4    # OK. Discriminator is now 2.
>>> u.b = 1      # Discriminator is now ≠ 1 or 2.
```

Typedefs

- Create an alias to a type.

```
module M {
  typedef float Temperature;
  struct Reading {
    Temperature min;
    Temperature max;
  };
  typedef Reading MyReading;
};
```

- Just use the aliased type from Python.

```
>>> r = M.Reading(1.2, 3.4)
>>> s = M.MyReading(5.6, 7.8)
```

Sequences

- Variable length list of elements.
- Bounded or unbounded.
- Must be declared with `typedef`.

IDL

```
module M {
  typedef sequence<long>              LongSeq;
  typedef sequence<long,5>            BoundedLongSeq;
  typedef sequence<octet>             OctetSeq;
  typedef sequence<sequence<short> >  NestedSeq;
};
```

Note the space

Python

```
>>> ls = [1,2,3,4,5]    #  Valid as a LongSeq or BoundedLongSeq.
>>> ls = [1,2,3,4,5,6]  #  Too long for BoundedLongSeq.
>>> ls = (1,2,3,4,5)    #  Tuples are valid too.
>>> os = "abc\0\1\2"    #  octet and char map to Python string for speed.
>>> ns = [[1,2],[]]     #  Valid NestedSeq.
```

Arrays

- Fixed length list of elements.
- Must be declared with `typedef`.

IDL

```
module M {
  typedef long  LongArray[5];
  typedef char  CharArray[6];
  typedef short TwoDArray[3][2];
};
```

Python

```
>>> la = [1,2,3,4,5]  # Valid LongArray.
>>> la = (1,2,3,4,5)  # Valid LongArray.
>>> ca = "ABCDEF"     # octet and char map to string again.
>>> ta = [[1,2],[3,4],[5,6]]
```

Exceptions

- Used to indicate an error condition.
- Almost the same as structures
  - Except that they can be empty.
- Not actually types
  - They cannot be used anywhere other than a `raises` clause.

IDL

```
module M {
  exception Error {};
  exception Invalid {
    string reason;
  };
};
```

Python

```
raise M.Error()
raise M.Invalid("Presentation too boring")
```

System Exceptions

- All CORBA operations can raise system exceptions.

```
module CORBA {
  enum completion_status {
    COMPLETED_YES,
    COMPLETED_NO,
    COMPLETED_MAYBE
  };
  exception name {
    unsigned long     minor;
    completion_status completed;
  };
};
```

- `BAD_PARAM`, `COMM_FAILURE`, `OBJECT_NOT_EXIST`, `...`
- Minor codes might tell you something useful:

```
>>> obj.echoString(123)
Traceback (innermost last):
...
omniORB.CORBA.BAD_PARAM: Minor: BAD_PARAM_WrongPythonType, COMPLETED_NO.
```

TypeCode and Any

- An Any can contain data with any IDL-declared type.
- A TypeCode tells you (and the ORB) everything there is to know about a type.

IDL

```
module M {
  struct Event {
    long number;
    any  data;
  };
};
```

Python

```
>>> a = CORBA.Any(CORBA.TC_long, 1234)
>>> a.value()
1234
>>> a.typecode().kind()
CORBA.tk_long
>>> a = CORBA.Any(CORBA.TypeCode("IDL:M/MyStruct:1.0"), s)
>>> a.typecode().kind()
CORBA.tk_struct
```

Interfaces

- Define the interface of a (potentially) remote object.
- Can contain
  - type declarations
  - exception declarations
  - constant definitions
  - operations
  - attributes
- Support multiple inheritance.
- Create a valid IDL type.

Operations

- Parameters may be `in`, `out`, or `inout`.
- Single return value or `void`.
- Operations with more than one result value return a tuple.

IDL

```
interface I {
  void op1();
  void op2(in string s, in    long l);
  void op3(in string s, out   long l);
  long op4(in string s, in    long l);
  long op5(in string s, inout long l);
};
```

Python

```
>>> o.op1()
>>> o.op2("Hello", 1234)
>>> l = o.op3("Hello")
>>> r = o.op4("Hello")
>>> r, l = o.op5("Hello", 2345)
```

Exceptions

- Exceptions are declared with a `raises` clause.
- System exceptions are implicit, and must not be declared.

IDL

```
module M {
  interface I {
    exception NotPermitted { string reason; };
    exception NoSuchFile {};
    void deleteFile(in string name) raises (NotPermitted, NoSuchFile);
  };
};
```

Python

```
try:
    o.deleteFile("example.txt")
    print "Deleted OK"
except M.I.NotPermitted, ex:
    print "Not permitted because:", ex.reason
except M.I.NoSuchFile:
    print "File does not exist"
```

Oneway

- Operations may be declared `oneway`.
- Best effort delivery — may never arrive!
- Client will *probably* not block.
- No return value, `out` or `inout` parameters.
- No user exceptions.
- Client may still receive system exceptions.

IDL

```
interface I {
  oneway void eventHint(in any evt);
};
```

Python

```
a = CORBA.Any(CORBA.TypeCode("IDL:Mouse/Position:1.0"),
              Mouse.Position(100, 200))
o.eventHint(a) # Don't care if the event is lost
```

Attributes

- **Not** the same as Python attributes.
- Shorthand for a get/set pair of operations.
- Server may implement them however it likes.
- Cannot raise user exceptions.
- Use with care!

IDL

```
interface VolumeControl {
  attribute float level;
  readonly attribute string name;
};
```

Python

```
>>> o._get_level()
1.234
>>> o._set_level(2.345)
>>> o._get_name()
'left speaker'
>>> o._set_name("right speaker")
AttributeError: _set_name
```

Inheritance

- Interfaces may be derived from any number of other interfaces.
- Operations and attributes cannot be redefined.

IDL

```
interface A {
  void opA();
};
interface B {
  void opB();
};
interface C : A, B {
  void opC(); // OK
  void opA(); // Error: clash with inherited operation
};
```

Object references

- Interfaces declare first-class types.
- Objects are passed by reference.
  - Or, more correctly, object *references* are passed by value.

IDL

```
interface Game {
  ...
};
interface GameFactory {
  Game newGame();
};
```

Python

```
>>> gf   = #  get a GameFactory reference from somewhere...
>>> game = gf.newGame()
```

Object references

- A *nil* object reference is represented by Python `None`.
- Derived interfaces can be used where a base interface is specified.
- The implicit base of all interfaces is `Object`.

IDL

```
interface A { ... };
interface B : A { ... };
interface C {
  void one(in A an_A);            // Accepts A or B
  void two(in Object an_Object); // Accepts A, B, or C
};
```

Forward declarations

- Used to create cyclic dependencies between interfaces.
- Full definition must be available.
  - Some IDL compilers require that it is in the same file.

IDL

```
interface I;
interface J {
  attribute I the_I;
};
interface I {
  attribute J the_J;
};
```

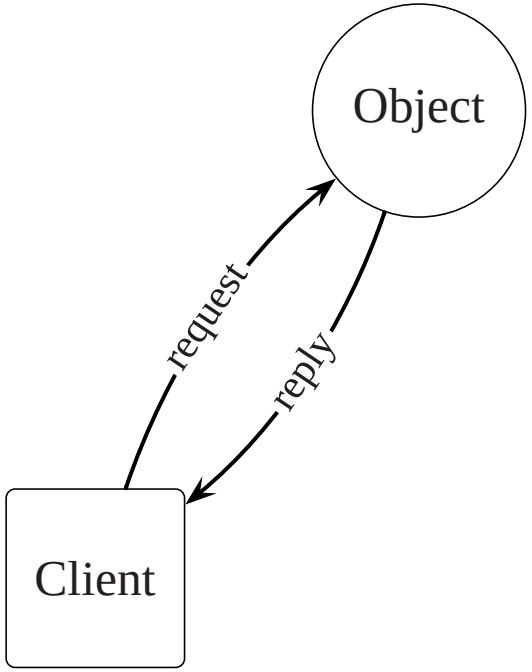
Objects by value

- CORBA 2.3 added `valuetype` for objects passed by value, rather than by reference.
- Like structs with single inheritance.
- Supports transmission or arbitrary graphs.
- Objects can have behaviour as well as state.
- Lots of nastiness:
  - IDL no longer forms the only contract between client and server.
  - Mobile code security issues.
  - Issues with the on-the-wire format.
- Not supported by any Python ORB yet.

IDL: Summary

- IDL defines:
  - Interfaces of objects
  - Types which may be transmitted
  - Constants
- Forms the contract between client and server.
- Purely a declarative language.

CORBA Object model



- What exactly is an ‘Object’?
- Often, a CORBA object is simply a programming language object which is remotely accessible.
- In general, an object’s existence may be independent of:
  - Clients holding references
  - References elsewhere
  - Operation invocations
  - Implementation objects (servants)
  - Server processes

Terminology

Object reference

- A handle identifying an object.
- Contains sufficient information to locate the object.
- The object may not exist
  - at the moment
  - ever.
- Refers to a single object.
- An object may have many references to it.
- Analogous to a pointer in C++.

Terminology

Servant

- A programming language entity *incarnating* one or more CORBA objects.
- Provides a concrete target for a CORBA object.
- Not a one-to-one mapping between CORBA objects and servants
  - A servant may incarnate more than one object simultaneously.
  - Servants can be instantiated on demand.
- Servants live within a *server* process.

Terminology

Client and Server

- A *client* is an entity which issues requests on an object.
- A *server* is a process which may support one or more servants.
- Both are rôles, not fixed designations
  - A program can act as a client one moment, server the next
  - or both concurrently.

Object Request Broker

- The ORB brokers requests between objects.
- Responsible for
  - object reference management
  - connection management
  - operation invocation
  - marshalling
  - . . .
- Public API specified in *pseudo*-IDL.
  - Like real IDL, but not necessarily following the language mapping rules.
- Not a stand-alone process—library code in all CORBA applications.

Object Request Broker

```
module CORBA { // Pseudo IDL
  interface ORB {
    string object_to_string(in Object obj);
    Object string_to_object(in string str);

    typedef string ObjectId;
    typedef sequence <ObjectId> ObjectIdList;
    exception InvalidName {};

    ObjectIdList list_initial_services();
    Object resolve_initial_references(in ObjectId identifier)
      raises (InvalidName);

    boolean work_pending();
    void    perform_work();
    void    run();
    void    shutdown(in boolean wait_for_completion);
    void    destroy();
    ...
  };
  ORB ORB_init(inout arg_list argv, in string orb_identifier);
};
```

Portable Object Adapter

- Objects are created within POAs.
- Within a POA, an object is identified with an *object id*.
- Objects can be *activated* and *deactivated*.
- A servant *incarnates* an activated object.
- When an object is deactivated, the associated servant is *etherealized*.
- There can be a many-to-one mapping between objects and servants.
  - i.e. a single servant can incarnate multiple objects within a POA.
  - or even within multiple POAs.

POA Policies

- The behaviour of a POA is determined by its *policies*:
  - Threading model.
  - Transient or persistent object life-span.
  - One id per servant or multiple ids.
  - User-provided object ids, or system-provided ids.
  - Use an active object map, default servant, servant locator, or servant activator.
  - Allow implicit activation or not.

Transient / Persistent Objects

- To clients, object references are opaque.
  - So they cannot tell anything about the object's life cycle.
- Servers classify objects as *transient* or *persistent*.
- Transient objects
  - Do not exist past the life of the server process.
  - Good for callbacks, session management, etc.
- Persistent objects
  - Can exist past the life of a server process.
  - Good for long-lived services.
  - The POA does not persist the state for you!

POA Interface

```
module PortableServer {
  ...
  native Servant;
  ...
  interface POA {
    ...
    ObjectId activate_object(in Servant p_servant)
      raises (ServantAlreadyActive, WrongPolicy);

    void     activate_object_with_id(in ObjectId id, in Servant p_servant)
      raises (ServantAlreadyActive, ObjectAlreadyActive, WrongPolicy);

    void     deactivate_object(in ObjectId oid)
      raises (ObjectNotActive, WrongPolicy);

    Object   create_reference(in CORBA::RepositoryId intf)
      raises (WrongPolicy);

    Object   create_reference_with_id(in ObjectId oid,
                                      in CORBA::RepositoryId intf)
      raises (WrongPolicy);

    ...
```

POA Interface

```
    ...

    ObjectId servant_to_id(in Servant p_servant)
      raises (ServantNotActive, WrongPolicy);

    Object   servant_to_reference(in Servant p_servant)
      raises (ServantNotActive, WrongPolicy);

    Servant  reference_to_servant(in Object reference)
      raises(ObjectNotActive, WrongAdapter, WrongPolicy);

    ObjectId reference_to_id(in Object reference)
      raises (WrongAdapter, WrongPolicy);

    Servant  id_to_servant(in ObjectId oid)
      raises (ObjectNotActive, WrongPolicy);

    Object   id_to_reference(in ObjectId oid)
      raises (ObjectNotActive, WrongPolicy);
  };
};
```

POA use

```
# Create Game servant object
gservant = Game_i(self, name, game_poa)

# Activate it
gid = game_poa.activate_object(gservant)

# The POA now holds a reference to the servant.
del gservant

# Get the object reference
gobj = game_poa.id_to_reference(gid)

...

# Deactivate the object. Deletes the servant object,
# since the POA held the only reference to it.
game_poa.deactivate_object(gid)
```

Servant definition

- To activate an object, you have to provide a Python *servant* object.
- The servant's class must be derived from the servant *skeleton* class.
- For interface `I` in module `M`, the skeleton class is `M__POA.I` (with two underscores).
  - Only the top-level module name is suffixed: the skeleton class for `M::N::I` is `M__POA.N.I`.
- The servant class must provide implementations of all the IDL-defined operations, with the correct argument types.

Servant definition

IDL

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

Python

```
import Snake__POA

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

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

    def reset(self):
        self.value = 0

servant = Adder_i()
poa.activate_object(servant)
```

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, . . .

Conclusion

1. My recommendations
2. General hints
3. Further resources
4. A big example for the keen

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.
- 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.

Further resources

- Python CORBA tutorial (expanded version of this presentation)

  `www.omniorb.org/omniORBpy/tutorial/`

- CORBA IDL to Python language mapping,

  `http://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 is 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.

Example CORBA application

- This example demonstrates many design patterns used in real CORBA applications.
- A noughts-and-crosses game:
  - A single server, supporting any number of games.
  - Two players per game (obviously), plus any number of spectators.
  - Clients do not know the rules of the game.
- Terribly over-engineered for what it is.
- Full source code to the example available from `www.omniorb.org/omniORBpy/tutorial/`

Example application