Distributed Applications with CORBA

Dr Duncan Grisby
duncan@grisby.org
Outline

1. Introduction
2. A simple example
3. IDL and its mapping to Python
4. Object life cycle
5. Common requirements
6. Summary
About me

• BA and PhD at the University of Cambridge Computer Laboratory.

• Work for AT&T Laboratories Cambridge (www.uk.research.att.com).

• Working on CORBA systems — ways to make CORBA easier to use.

• Main author of omniORBpy
  – but this tutorial covers all the Python ORBs.
Introduction

1. What is a distributed system?
2. Why would we want one?
3. Distributed system technologies
4. What is CORBA?
5. CORBA ORBs for Python
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, ...
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.
- 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.
- All specifications are available for free.
Why use Python?

- All the normal reasons...
  - High level, clear syntax, interactive prompt, batteries included...

- Python is the only mainstream scripting language with a standard CORBA mapping.

- The CORBA to Python mapping is extremely simple:
  - C++ mapping specification: 166 pages
  - Java: 130 pages
  - Python: 16 pages!
Python ORBs

- **omniORBpy**
  - Based on C++ omniORB. Multi-threaded. Free (LGPL).
  - [www.omniorb.org/omniORBpy](http://www.omniorb.org/omniORBpy)

- **orbit-python**
  - Based on C ORBit. Single-threaded. Free (LGPL).
  - [projects.sault.org/orbit-python/](http://projects.sault.org/orbit-python/)

- **Fnorb**
  - Mostly Python, with a small amount of C. Multi-threaded. Newly open source (Python style). Back from dead?
  - [www.fnorb.com](http://www.fnorb.com)

- **ILU**
  - Based on C ILU. More than just CORBA. Open source. Dead?
A simple example

1. A picture
2. IDL, the Interface Definition Language
3. Client code
4. Server code
A classical object model
- the client sends request messages to the object; the object sends replies back.

The client does not care where the object is
- because the ORB deals with it.

The client knows what messages it can send, because the object has an interface
- specified in CORBA IDL…
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();
    };
}
Python client

```python
>>> import sys, CORBA, Snake
>>> orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
>>> adder = orb.string_to_object("corbaname:rir:#adder.obj")
>>> adder.accumulate(5)
5
>>> adder.accumulate(6)
11
>>> adder.accumulate(42)
53
>>> adder.reset()
>>> adder.accumulate(10)
10
```
import sys, CORBA, CosNaming, Snake, 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

orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
poa = orb.resolve_initial_references("RootPOA")
adderServant = Adder_i()
poa.activate_object(adderServant)
adderObjref = adderServant._this()

nameRoot = orb.resolve_initial_references("NameService")
nameRoot = nameRoot._narrow(CosNaming.NamingContext)
name = [CosNaming.NameComponent("adder", "obj")]
nameRoot.rebind(name, adderObjref)

poa._get_the_POAManager().activate()
orb.run()
IDL and its Python mapping

1. Practicalities
2. Simple types
3. Constants
4. Constructed types
5. Any
6. Interfaces
7. Objects by value
8. Repository identifiers
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

<table>
<thead>
<tr>
<th>IDL type</th>
<th>Meaning</th>
<th>Python mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>TRUE or FALSE</td>
<td>int</td>
</tr>
<tr>
<td>octet</td>
<td>8-bit unsigned</td>
<td>int</td>
</tr>
<tr>
<td>short</td>
<td>16-bit signed</td>
<td>int</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16-bit unsigned</td>
<td>int</td>
</tr>
<tr>
<td>long</td>
<td>32-bit signed</td>
<td>int</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32-bit unsigned</td>
<td>long</td>
</tr>
<tr>
<td>long long</td>
<td>64-bit signed</td>
<td>long</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>64-bit unsigned</td>
<td>long</td>
</tr>
<tr>
<td>float</td>
<td>32-bit IEEE float</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>64-bit IEEE float</td>
<td>float</td>
</tr>
<tr>
<td>long double</td>
<td>≥ 80-bit IEEE float</td>
<td>CORBA.long_double</td>
</tr>
</tbody>
</table>
# Textual types

<table>
<thead>
<tr>
<th>IDL</th>
<th>Meaning</th>
<th>Python</th>
</tr>
</thead>
</table>
| char  | 8-bit quantity, usually ISO 8859-1.  
  – or a character from any byte-oriented code set.  
  – or a single octet from a multi-byte code set.                                                                                           | string (length 1)           |
| string| String of char, usually ISO 8859-1.  
  – no embedded nulls.  
  – string<bound> is a bounded string.                                                                                                      | string                      |
| wchar | Wide character from some code set.  
  – must support UTF-16 (Unicode).  
  – but can support any code set.                                                                                                          | CORBA.wstring (length 1)    |
| wstring| String of wchar. Must support UTF-16.  
  – no embedded nulls.  
  – wstring<bound> is a bounded wide string.                                                                                             | CORBA.wstring               |
Confused yet?

Just ignore the complex bits. Use `char` and `string`, mapped to Python string, assuming ISO 8859-1 (Latin-1).

Pretend the table looks like…

<table>
<thead>
<tr>
<th>IDL</th>
<th>Meaning</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8-bit ISO 8859-1 character.</td>
<td>string (length 1)</td>
</tr>
<tr>
<td>string</td>
<td>String of ISO 8859-1 characters.</td>
<td>string</td>
</tr>
<tr>
<td></td>
<td>– no embedded nulls.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>– string&lt;bound&gt; is a <code>bounded</code> string.</td>
<td></td>
</tr>
<tr>
<td>wchar</td>
<td>Unicode character.</td>
<td>unicode (length 1)</td>
</tr>
<tr>
<td>wstring</td>
<td>Unicode string.</td>
<td>unicode</td>
</tr>
<tr>
<td></td>
<td>– no embedded nulls.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>– wstring&lt;bound&gt; is a <code>bounded</code> wstring.</td>
<td></td>
</tr>
</tbody>
</table>
Fixed point

\[
\text{fixed}<\text{digits, scale}> \quad 1 \leq \text{digits} \leq 31 \\
\quad 0 \leq \text{scale} \leq \text{digits}
\]

Fixed point value with \( \text{digits} \) total decimal digits, including \( \text{scale} \) decimal places.

**IDL**

* e.g. \( \text{fixed}<5,0> \) : range \( \pm 99999 \)
* \( \text{fixed}<5,2> \) : range \( \pm 999.99 \)
* \( \text{fixed}<5,5> \) : range \( \pm 0.99999 \)

**Python**

\[
f = \text{CORBA.fixed}(5, 2, 12345) \\
f.\text{value}() \rightarrow 12345L \\
f.\text{precision}() \rightarrow 5 \\
f.\text{decimals}() \rightarrow 2 \\
g = f * 2 \\
h = f + g \\
i = \text{MyFixed}(23456)
\]

123.45 \\
actual value \( \times 10^{\text{scale}} \) as a long \\
\( \text{digits} \) \\
\( \text{scale} \) \\
maths with integers... \\
... and between fixeds \\
where \text{MyFixed} is a typedef (see later).
module M {
    const long ONE = 1;
    const long TWO = ONE + ONE;
    const unsigned long mask = 0xff00ff00;
    const unsigned long value = 0x12345678 & mask;
    const double PI = 3.14;
    const char initial = 'D';
    const char null = '\0';
    const string title = "CORBA Tutorial";
    const string invalid = "\0 not allowed"; // Error!
    const fixed limit = 123.45d; // Not fixed<...>
    const boolean this_is_fun = TRUE;
};

>>> import M
>>> M.TWO
2
>>> M.initial
'D'
>>> M.title
'CORBA Tutorial'
>>> M.title = "not so constant"  # Oh dear!
module M {
    const long ONE = 1;
    module N {
        const long TWO = ONE + ONE;
    };
};
module O {
    const long THREE = M::ONE + M::N::TWO;
};
module M { // Reopen the module
    const long FOUR = O::THREE + ONE;
};
const long FIVE = 5; // Nasty global declaration

IDL

Python

>>> import M, M.N, O, _GlobalIDL
>>> M.ONE
1
>>> M.N.TWO
2
>>> _GlobalIDL.FIVE
5

Name not standardised!
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;
    }
};
```
Unions

- Consist of a *discriminator* and a *value*.
- Discriminator type can be integer, boolean, enum, char.

```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.
```
Unions

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

**IDL**

```
>>> u = M.MyUnion(2, 3.4)  # Initialise with discriminator/value
>>> u._d, u._v             # Access the discriminator and value.
(2, 3.4)
>>> u.b = 1                # Discriminator is now ≠ 1 or 2.
>>> u._d = 5               # Set it explicitly.
```

**Python**
Unions

- Multiple case labels permitted.

```idl
module M {
    enum Colour { red, green, blue };  
    union Another switch (Colour) {
        case red:
            case green: string message;
            case blue: short number;
    }
};
```
Unions

- Default case is optional.

**IDL**

```idl
module M {
    union Optional switch (boolean) {
        case TRUE: string message;
    }
}
```

**Python**

```python
>>> empty = M.Optional(0, None)
>>> full = M.Optional(message = "More boring text")
```
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)

• Creates a Python object with the typedef name, to be passed to various functions...
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;
}
```

```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**

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

**Python**

```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]]
```
Recursive types

- Structs and unions containing sequences of themselves.
- CORBA 2.4 introduced forward declarations:

```
module M {
    struct Tree;
    typedef sequence <Tree> TreeSeq;
    struct Tree {
        long data;
        TreeSeq children;
    };
};
```

- With CORBA 2.0–2.3, use an anonymous type:

```
module M {
    struct Tree {
        long data;
        sequence <Tree> children;
    };
};
```
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

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

Python

```python
raise M.Error()
raise M.Invalid("Tutorial too boring")
```
System Exceptions

- All CORBA operations can raise system exceptions.

```c
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:

```python
>>> 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
```

```python
>>> 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)
```
Operations

- Parameters and results are passed by value.
  - What about local calls?
    - omniORBpy — by value
    - orbit-python — by value
    - Fnorb — by reference
    - ILU — by reference
Operations

- No method overloading.

```java
interface I {
    void op(in string s);
    void op(in long l); // Illegal!
}
```
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!

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

IDL

>>> 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`.

```
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.

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

IDL
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.
Repository Identifiers

- All IDL declarations have a repository identifier.
- Used in the Interface Repository.
- Needed by functions like `CORBA.TypeCode()`.
- Usually of the form ‘IDL:M1/M2/Foo:1.0’.
- Can be modified with `#pragma version`, `#pragma ID`, and `#pragma prefix`.
- Find from Python with `CORBA.id(type name)`

```python
    tc = CORBA.TypeCode(CORBA.id(MyModule.MyStruct))
```
• IDL defines:
  – Interfaces of objects
  – Types which may be transmitted
  – Constants

• Forms the contract between client and server.

• Purely a declarative language.
Object life cycle

1. Example application
2. The CORBA object model
3. Objects, object references, servants and servers
4. The ORB
5. Client side
6. Server side: BOAs and POAs
Example application

- We will illustrate the rest of the tutorial with a simple application, a tic-tac-toe game.

- Design parameters:
  - 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.

- Comments to do with the example appear in green. (Not in the proceedings.)

- Full source code to the example available from www.omniORB.org/omniORBpy/tutorial/
Example application
CORBA Object model

- Remember the simple picture from the start? 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++.
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.
Terminology

Stubs and skeletons

- IDL is compiled to *stubs* and *skeletons* in the target language.
- The resulting files are often called just ‘stubs’, even if the skeletons are there too.
- Stubs implement object references.
- Skeletons support the incarnation of servants.
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.
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);
}
Object Request Broker

ORB ORB_init(inout arg_list argv,
in string orb_identifier);

– Initialises the ORB.
– Eats any command line arguments beginning ‘–ORB’.
– ORB identifier requests a particular ORB.
– In Python, CORBA.ORB_ID tells you the correct string.

>>> import CORBA, sys
>>> orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
Object Request Broker

```cpp
string object_to_string(in Object obj);

– Converts an object reference into a standard stringified form, e.g.

IOR:010000001900000049444a3a54757466f7269616c706c653a312e30
000000000100000000000000000000000000100102000200007000000006d79686678740000d04
00001c000000057617320697420776f72746820747970672068697320696e3f010000000000000008000000100000000545441

– IOR stands for Interoperable Object Reference

– meaning it can be understood by any compliant ORB.
Object Request Broker

Object string_to_object(in string str);

– Converts an IOR string back into an object reference.

– Alternatively, from CORBA 2.4, accepts URIs of the form:
  corbaloc::<host>:<port>/<key>
corbaloc:rir:<initial reference>
corbaname::<host>:<port>/<key>#<name>
corbaname:rir:<initial reference>#<name>
...


Object Request Broker

ObjectIdList list_initial_services();

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

– Provide access to various services. Some built-in, others administratively configured.

>>> orb.list_initial_services()
['NameService', 'RootPOA', 'POACurrent']
>>> orb.resolve_initial_references("NameService")
<CosNaming._objref_NamingContext instance at 8159b98>
>>> orb.resolve_initial_references("Foo")
Traceback (innermost last):
  ...
CORBA.ORB.InvalidName
Object Request Broker

boolean work_pending();
void perform_work();
void run();

– Allow the ORB to service incoming requests.

– run() is a blocking call
  – Supposed to be called from the main thread.
  – Not necessary with all ORBs (e.g. omniORB).

– work_pending() and perform_work() allow polled access.
  – Burns processor time.
  – Again, not necessary with omniORB etc.
Object Request Broker

void shutdown(in boolean wait_for_completion);

– Stops any threads blocked in run().

– If wait_for_completion is true, blocks until all incoming requests have finished.

– So it can’t be used inside an operation implementation!

void destroy();

– Frees up all resources associated with the ORB.
Portability notes

- `run()`, `work_pending()`, `perform_work()`,
  `shutdown()`, and `destroy()` were all added in CORBA 2.3.

- Older ORBs (Fnorb, ILU) use proprietary equivalents.

- Check your ORB’s documentation.
Client side

- Client receives an object reference, either from an ORB function, or a call to another object...
- ...and invokes operations on the object.
- Except that the type of the object may not be known...
  - e.g. expecting a reference with interface `I`, but receive `J` derived from `I`.
  - Make sure with `obj._narrow(I)`.
  - It is ORB dependent when you have to do this.
Narrowing

// A.idl
interface I {
    void example();
};
interface J {
    Object getObject();
    I getI();
};

// B.idl
interface K : I {
    void example2();
};

- Imagine a client has only seen A.idl

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
<th>Client sees</th>
</tr>
</thead>
<tbody>
<tr>
<td>getObject()</td>
<td>I</td>
<td>Probably I</td>
</tr>
<tr>
<td>getI()</td>
<td>I</td>
<td>Definitely I</td>
</tr>
<tr>
<td>getObject()</td>
<td>K</td>
<td>Definitely Object</td>
</tr>
<tr>
<td>getI()</td>
<td>K</td>
<td>Maybe I</td>
</tr>
</tbody>
</table>
Narrowing

- It is often tempting to miss out necessary narrows...

```python
nameRoot = orb.resolve_initial_references("NameService")
obj = nameRoot.resolve([CosNaming.NameComponent("Example", "obj")])
```

This fails if you meet a Naming service with some derived interface. Always use:

```python
nameRoot = orb.resolve_initial_references("NameService")
nameRoot = nameRoot._narrow(CosNaming.NamingContext)
if nameRoot is None:
    print "Invalid NameService reference!"
    sys.exit(1)

obj = nameRoot.resolve([CosNaming.NameComponent("Example", "obj")])
```
Narrowing

- Always narrow to the least derived interface possible. e.g. narrow to `NamingContext` rather than `NamingContextExt` if you do not need the extended features.

The following code will fail with many naming services, for no good reason:

```python
nameRoot = orb.resolve_initial_references("NameService")
nameRoot = nameRoot._narrow(CosNaming.NamingContextExt)
if nameRoot is None:
    print "Invalid NameService reference!"
    sys.exit(1)

obj = nameRoot.resolve([CosNaming.NameComponent("Example", "obj")])
```
Server side

- Objects must be registered with an object adapter.

- CORBA originally specified the BOA — Basic Object Adapter.
  - Too loosely specified for server code to be portable between ORBs.
  - See your ORB’s documentation if it has a BOA.

- Now have the POA — Portable Object Adapter.
  - Server code is portable between ORBs.
  - Specifies a very wide range of facilities.
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:
  - Single threaded or ORB-controlled threading.
  - 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 States

- A POA’s state is controlled with its POA manager:
  - Holding: Incoming requests are queued, up to a limit.
  - Active: Incoming requests are dispatched to the relevant objects.
  - Discarding: Incoming requests are met with TRANSIENT exceptions.
  - Inactive: The POA is about to be shut down. Cannot leave this state.

- More than one POA can be controlled by the same POA manager.
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);

};

}
# 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**

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

**Python**

```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
```
Common Requirements

1. Finding objects
2. Creating new objects
3. Transferring bulk data
4. Event notification
5. Session management
6. Garbage collection
7. General hints
Finding objects

- Naming service
  - Graph of name to object bindings.
  - Usually a tree.
  - Register the GameFactory here.

- Trading service
  - Find objects by their properties.
  - Must be careful not to get overwhelmed by genericity.

- Published IOR strings
  - Great for simple examples.
  - Unmaintainable for more than one or two references.
module CosNaming {
    struct NameComponent {
        string id;
        string kind;
    };
    typedef sequence<NameComponent> Name;
    ...
    interface NamingContext {
        ...
        void bind (in Name n, in Object obj) raises ...
        void rebind (in Name n, in Object obj) raises ...
        void bind_context (in Name n, in NamingContext nc) raises ...
        void rebind_context (in Name n, in NamingContext nc) raises ...
        Object resolve (in Name n) raises ...
        void unbind (in Name n) raises ...
        NamingContext new_context ();
        NamingContext bind_new_context (in Name n) raises ...
        void destroy () raises ...
        void list (in unsigned long how_many,
                    out BindingList bl, out BindingIterator bi);
    };
}
Creating new objects

• Factory pattern
  – Objects which have operations to create new objects.
  – e.g. the GameFactory:

```plaintext
defmodule TicTacToe {
  interface GameFactory {
    exception NameInUse {};
    Game newGame(in string name) raises (NameInUse);
    ...
  };
};
```
Factory pattern

class GameFactory_i (TicTacToe__POA.GameFactory):
    ...
    def newGame(self, name):
        # Create a POA for the game and its associated objects.
        # Default policies are suitable. Having one POA per game makes
        # it easy to deactivate all objects associated with a game.
        try:
            game_poa = self.poa.create_POA("Game-" + name, None, [])
        except PortableServer.POA.AdapterAlreadyExists:
            raise TicTacToe.GameFactory.NameInUse()

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

        # Activate it
        gid = game_poa.activate_object(gservant)

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

        # Activate the POA
        game_poa._get_the_POAManager().activate()
    ...
    # Return the object reference
    return gobj
• Sequences
  – Simple, but can’t cope with *really* large items.

• Iterator pattern. e.g. GameIterator:

```cpp
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();
};
```
Iterator pattern

class GameIterator_i (TicTacToe__POA.GameIterator):
    def __init__(self, factory, poa, games):
        self.factory, self.poa, self.games = factory, poa, games
        self.tick = 1 # Tick for time-out garbage collection

    def next_n(self, how_many):
        self.tick = 1
        front = self.games[:int(how_many)]
        self.games = self.games[int(how_many):]

        # Convert internal representation to GameInfo sequence
        ret = map(lambda g: TicTacToe.GameInfo(g[0], g[2]), front)

        if self.games:
            more = 1
        else:
            more = 0

        return (ret, more)

    def destroy(self):
        id = self.poa.servant_to_id(self)
        self.factory._removeIterator(id)
        self.poa.deactivate_object(id)
Event notification

- Event service
  - Provides an event channel connecting producers to consumers.
  - Unfiltered event delivery.
  - Push or pull transmission and reception.
- Notification service
  - Adds filtering to the Event service.
- Roll-your-own
  - Avoids use of generic interfaces.
  - Tends to become unmanageable with more than a few clients.
  - Used in the example to avoid dependencies.
Session management

- Often need to know *which* client is making a call.
- Give clients a ‘cookie’ to identify them.
  - Server must maintain a map from cookies to clients.
  - Relatively easy for clients to masquerade as others.

```java
interface Game {
    ...
    unsigned long watchGame  (in Spectator s, out GameState state);
    void                unwatchGame(in unsigned long cookie);
};
```
Session management

- Clients acquire a session object, and interact through that.
  - Client association is implicit in the target session object.
  - Harder (but not that hard) for clients to masquerade as others.

```java
interface Game {
  ...
  GameController joinGame(in Player p, out PlayerType t)
    raises (CannotJoin);
};
interface GameController {
  ...
  GameState play(in short x, in short y)
    raises (SquareOccupied, InvalidCoordinates, NotYourGo);
};
```
Garbage collection

• Hard!
  – No single approach satisfies all situations.
  – CORBA does not provide anything by default.
  – The decision about when an object is finished with is application specific.

• Can sometimes be avoided altogether
  – Periodically kill everything and restart the server.
  – Provide an API for manually clearing garbage.
  – Both methods used to clear dead Games.
Reference counting and pinging

- Superficially quite simple.
- Pings necessary to cope with malicious and buggy clients.
- Client pings server or server pings client?
- Does not scale well, since you can drown in pings.
- Reference cycles are a problem.
- Fits badly with persistent objects.
Garbage collection

- Evictor pattern
  - Impose a maximum on the number of active objects.
  - Activate objects on demand.
  - If maximum is reached, ‘evict’ an active object.
  - How do you pick the victim? LRU, FIFO, …
  - How do you pick the maximum?
  - Works well with database-backed persistence.
  - See Henning & Vinoski for details.
Garbage collection

• Timeouts
  – Object issues a ‘lease’ for a limited period of time.
  – Clients may explicitly or implicitly renew the lease.
  – Server is potentially more vulnerable to malicious clients than with evictor.
  – Better for transient objects.
  – Hard to do without threads.
  – Used to clean up GameIterators.
class IteratorScavenger (threading.Thread):
...
    def run(self):
        lock = self.factory.lock
        iterators = self.factory.iterators
        poa = self.factory.iterator_poa
        manager = poa._get_the_POAManager()

        while 1:
            time.sleep(SCAVENGER_INTERVAL)

            # Bonus points for spotting why we hold requests...
            manager.hold_requests(1)
            lock.acquire()

            for id, iter in iterators.items():
                if iter.tick == 1:
                    iter.tick = 0
                else:
                    del iterators[id]
                    poa.deactivate_object(id)
            del iter
            lock.release()
            manager.activate()
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 (i.e. ones which use Any) if possible.

- Write your code in Python!
Further resources

• ‘Advanced CORBA Programming with C++’, by Michi Henning and Steve Vinoski. Published by 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.

• Python language mapping,
  http://www.omg.org/technology/documents/formal/python_language_mapping.htm

• CORBA specifications,
  www.omg.org/technology/documents/formal/
Summary

• We have learnt (I hope)…
  – What CORBA is.
  – IDL, the Interface Definition Language.
  – The CORBA object model.
  – Approaches to some common application requirements.
  – Another way in which Python is cool.
module TicTacToe {

  // State of a game.
  enum PlayerType { Nobody, Nought, Cross };  
  typedef PlayerType GameState[3][3];  

  // Forward declaration of all interfaces.
  interface GameFactory;  
  interface GameIterator;  
  interface Game;  
  interface GameController;  
  interface Player;  
  interface Spectator;  

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

  interface GameFactory {  
    exception NameInUse {};  

    Game newGame(in string name) raises (NameInUse);  
    // Create a new game  

    GameInfoSeq listGames(in unsigned long how_many, out GameIterator iter);  
    // List the currently active games, returning a sequence with at  
    // most how_many elements. If there are more active games than  
    // that, the iterator is non-nil, permitting the rest of the games  
    // to be retrieved.  
  };
interface GameIterator {
    GameInfoSeq next_n(in unsigned long how_many, out boolean more);
    // Return the next sequence of games, up to a maximum of
    // how_many. If more is true, there are more games to list.
    void destroy();
    // Destroy the iterator object.
};

interface Game {
    readonly attribute string name;  // Name of this game.
    readonly attribute short players; // Number of players registered.
    readonly attribute GameState state; // Current state of the game.
    exception CannotJoin {};

    GameController joinGame(in Player p, out PlayerType t)
    raises (CannotJoin);
    // Join a new game, passing in a Player object reference. Returns
    // a GameController object reference used to play the game. The
    // out argument lets the player know whether they are noughts or
    // crosses.

    unsigned long watchGame (in Spectator s, out GameState state);
    void unwatchGame(in unsigned long cookie);
    // Register or unregister a spectator for the game. watchGame()
    // returns a cookie to be used to unregister. Note the potential
    // for unregistering other spectators. This should really use an
    // event or notification service.

    void kill();
    // Kill the game prematurely.
};

interface GameController {
exception SquareOccupied {}
exception InvalidCoordinates {}
exception NotYourGo {}

GameState play(in short x, in short y)
  raises (SquareOccupied, InvalidCoordinates, NotYourGo);
  // Place a piece at the specified coordinates. Returns the new
  // game state.
};

interface Player {
  void yourGo(in GameState state);
  // Tell the player it is their go, giving the current game state.

  void end(in GameState state, in PlayerType winner);
  // End of game. winner is Nobody if the game is tied.

  void gameAborted();
};

interface Spectator {
  void update(in GameState state);
  // Update the current state of the game.

  void end(in GameState state, in PlayerType winner);
  void gameAborted();
};
#!/usr/bin/env python

# gameServer.py

import sys, threading, time, Queue
import CORBA, PortableServer, CosNaming
import TicTacToe, TicTacToe__POA

SCAVENGER_INTERVAL = 30

class GameFactory_i (TicTacToe__POA.GameFactory):
    def __init__(self, poa):
        # Lists of games and iterators, and a lock to protect access
        # to them.
        self.games = []
        self.iterators = {}
        self.lock = threading.Lock()
        self.poa = poa

        # Create a POA for the GameIterators. Shares the POAManager of
        # this object. The POA uses the default policies of TRANSIENT,
        # SYSTEM_ID, UNIQUE_ID, RETAIN, NO_IMPLICIT_ACTIVATION,
        # USE_ACTIVE_OBJECT_MAP_ONLY, ORB_CTRL_MODEL.

        self.iterator_poa = poa.create_POA("IterPOA", None, [])
        self.iterator_poa._get_the_POAManager().activate()

        self.iterator_scavenger = IteratorScavenger(self)

        print "GameFactory_i created."

    def newGame(self, name):
        # Create a POA for the game and its associated objects.
        # Default policies are suitable. Having one POA per game makes
# it easy to deactivate all objects associated with a game.
try:
    game_poa = self.poa.create_POA("Game-" + name, None, [])
except PortableServer.POA.AdapterAlreadyExists:
    raise TicTacToe.GameFactory.NameInUse()

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

# Activate it
gid = game_poa.activate_object(gservant)

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

# Activate the POA
game_poa._get_the_POAManager().activate()

# Add to our list of games
self.lock.acquire()
self.games.append((name, gservant, gobj))
self.lock.release()

# Return the object reference
return gobj

def listGames(self, how_many):
    self.lock.acquire()
    front = self.games[:int(how_many)]
    rest = self.games[int(how_many):]
    self.lock.release()

    # Create list of GameInfo structures to return
    ret = map(lambda g: TicTacToe.GameInfo(g[0], g[2]), front)
# Create iterator if necessary
    if rest:
        iter = GameIterator_i(self, self.iterator_poa, rest)
        iid = self.iterator_poa.activate_object(iter)
        iobj = self.iterator_poa.id_to_reference(iid)
        self.lock.acquire()
        self.iterators[iid] = iter
        self.lock.release()
    else:
        iobj = None # Nil object reference

    return (ret, iobj)

def _removeGame(self, name):
    self.lock.acquire()
    for i in range(len(self.games)):
        if self.games[i][0] == name:
            del self.games[i]
            break
    self.lock.release()

def _removeIterator(self, iid):
    self.lock.acquire()
    del self.iterators[iid]
    self.lock.release()

class GameIterator_i (TicTacToe__POA.GameIterator):
    def __init__(self, factory, poa, games):
        self.factory = factory
        self.poa = poa
        self.games = games
        self.tick = 1 # Tick for time-out garbage collection
        print "GameIterator_i created."
def __del__(self):
    print "GameIterator_i deleted."

def next_n(self, how_many):
    self.tick = 1
    front = self.games[:int(how_many)]
    self.games = self.games[int(how_many):]

    # Convert internal representation to GameInfo sequence
    ret = map(lambda g: TicTacToe.GameInfo(g[0], g[2]), front)

    if self.games:
        more = 1
    else:
        more = 0

    return (ret, more)

def destroy(self):
    id = self.poa.servant_to_id(self)
    self.factory._removeIterator(id)
    self.poa.deactivate_object(id)

class IteratorScavenger (threading.Thread):
    def __init__(self, factory):
        threading.Thread.__init__(self)
        self.setDaemon(1)
        self.factory = factory
        self.start()

    def run(self):
        print "Iterator scavenger running..."

        lock = self.factory.lock
iterators = self.factory.iterators
poa = self.factory.iterator_poa
manager = poa._get_the_POAManager()

while 1:
    time.sleep(SCAVenger_INTERVAL)

    print "Scavenging dead iterators..."
    
    # Bonus points for spotting why we hold requests...
    manager.hold_requests(1)
    lock.acquire()

    for id, iter in iterators.items():
        if iter.tick == 1:
            iter.tick = 0
        else:
            del iterators[id]
            poa.deactivate_object(id)

            # This del drops the last reference to the iterator so
            # it can be collected immediately. Without it, the
            # Python servant object stays around until the next
            # time around the loop.
            del iter

    lock.release()
    manager.activate()

class Game_i (TicTacToe__POA.Game):
    def __init__(self, factory, name, poa):
        self.factory = factory
        self.name = name
        self.poa = poa
self.lock = threading.Lock()

n = TicTacToe.Nobody

self.players = 0
self.state = [[n,n,n],
              [n,n,n],
              [n,n,n]]

self.p_noughts = None
self.p_crosses = None
self.whose_go = TicTacToe.Nobody
self.spectators = []
self.spectatorNotifier = SpectatorNotifier(self.spectators, self.lock)

print "Game_i created."

def __del__(self):
    print "Game_i deleted."

def _get_name(self):
    return self.name

def _get_players(self):
    return self.players

def _get_state(self):
    return self.state

def joinGame(self, player):
    try:
        self.lock.acquire()
        if self.players == 2:
            raise TicTacToe.Game.CannotJoin()
if self.players == 0:
    ptype = TicTacToe.Nought
    self.p_noughts = player
else:
    ptype = TicTacToe.Cross
    self.p_crosses = player

    # Notify the noughts player that it’s their go
    try:
        self.whose_go = TicTacToe.Nought
        self.p_noughts.yourGo(self.state)
    except (CORBA.COMM_FAILURE, CORBA.OBJECT_NOT_EXIST), ex:
        print "Lost contact with player"
        self.kill()

    # Create a GameController
    gc = GameController_i(self, ptype)
    id = self.poa.activate_object(gc)
    gobj = self.poa.id_to_reference(id)

    self.players = self.players + 1

finally:
    self.lock.release()

return (gobj, ptype)

def watchGame(self, spectator):
    self.lock.acquire()
    cookie = len(self.spectators)
    self.spectators.append(spectator)
    self.lock.release()

    return cookie, self.state
def unwatchGame(self, cookie):
    cookie = int(cookie)
    self.lock.acquire()
    if len(self.spectators) > cookie:
        self.spectators[cookie] = None
    self.lock.release()

def kill(self):
    self.factory._removeGame(self.name)
    if self.p_noughts:
        try:
            self.p_noughts.gameAborted()
        except CORBA.SystemException, ex:
            print "System exception contacting noughts player"
    if self.p_crosses:
        try:
            self.p_crosses.gameAborted()
        except CORBA.SystemException, ex:
            print "System exception contacting crosses player"
    self.spectatorNotifier.gameAborted()
    self.poa.destroy(1,0)
    print "Game killed"

def _play(self, x, y, ptype):
    """Real implementation of GameController::play()"""
    if self.whose_go != ptype:
        raise TicTacToe.GameController.NotYourGo()
if x < 0 or x > 2 or y < 0 or y > 2:
    raise TicTacToe.GameController.InvalidCoordinates()

if self.state[x][y] != TicTacToe.Nobody:
    raise TicTacToe.GameController.SquareOccupied()

self.state[x][y] = ptype

w = self._checkForWinner()

try:
    if w is not None:
        print "Winner:" , w
        self.p_noughts.end(self.state, w)
        self.p_crosses.end(self.state, w)
        self.spectatorNotifier.end(self.state, w)

        # Kill ourselves
        self.factory._removeGame(self.name)
        self.poa.destroy(1,0)
    else:
        # Tell opponent it’s their go
        if ptype == TicTacToe.Nought:
            self.whose_go = TicTacToe.Cross
            self.p_crosses.yourGo(self.state)
        else:
            self.whose_go = TicTacToe.Nought
            self.p_noughts.yourGo(self.state)

        self.spectatorNotifier.update(self.state)

except (CORBA.COMM_FAILURE, CORBA.OBJECT_NOT_EXIST), ex:
    print "Lost contact with player!"
    self.kill()
```python
return self.state

def _checkForWinner(self):
    
    """If there is a winner, return the winning player’s type. If the game is a tie, return Nobody, otherwise return None."""

    # Rows
    for i in range(3):
        if self.state[i][0] == self.state[i][1] and 
        self.state[i][1] == self.state[i][2] and 
        self.state[i][0] != TicTacToe.Nobody:
            return self.state[i][0]

    # Columns
    for i in range(3):
        if self.state[0][i] == self.state[1][i] and 
        self.state[1][i] == self.state[2][i] and 
        self.state[0][i] != TicTacToe.Nobody:
            return self.state[0][i]

    # Top-left to bottom-right
    if self.state[0][0] == self.state[1][1] and 
    self.state[1][1] == self.state[2][2] and 
    self.state[0][0] != TicTacToe.Nobody:
        return self.state[0][0]

    # Bottom-left to top-right
    if self.state[0][2] == self.state[1][1] and 
    self.state[1][1] == self.state[2][0] and 
    self.state[0][2] != TicTacToe.Nobody:
        return self.state[0][2]

    # Return None if the game is not full
    for i in range(3):
        for j in range(3):
```

if self.state[i][j] == TicTacToe.Nobody:
    return None

    # It’s a draw
    return TicTacToe.Nobody

class SpectatorNotifier (threading.Thread):
    
    # This thread is used to notify all the spectators about changes
    # in the game state. Since there is only one thread, one errant
    # spectator can hold up all the others. A proper event or
    # notification service should make more effort to contact clients
    # concurrently. No matter what happens, the players can’t be held
    # up.
    #
    # The implementation uses a simple work queue, which could
    # potentially get backed-up. Ideally, items on the queue should be
    # thrown out if they have been waiting too long.

    def __init__(self, spectators, lock):
        threading.Thread.__init__(self)
        self.setDaemon(1)
        self.spectators = spectators
        self.lock = lock
        self.queue = Queue.Queue(0)
        self.start()

    def run(self):
        print "SpectatorNotifier running..."

        while 1:
            method, args = self.queue.get()

            print "Notifying:", method
try:
    self.lock.acquire()
    for i in range(len(self.spectators)):
        spec = self.spectators[i]
        if spec:
            try:
                apply(getattr(spec, method), args)

            except (CORBA.COMM_FAILURE, CORBA.OBJECT_NOT_EXIST), ex:
                print "Spectator lost"
                self.spectators[i] = None

    finally:
        self.lock.release()

def update(self, state):
    s = (state[0][:], state[1][:], state[2][:])
    self.queue.put(("update", (s,)))

def end(self, state, winner):
    self.queue.put(("end", (state, winner)))

def gameAborted(self):
    self.queue.put(("gameAborted", ()))

class GameController_i (TicTacToe__POA.GameController):

def __init__(self, game, ptype):
    self.game = game
    self.ptype = ptype
    print "GameController_i created."

def __del__(self):
    print "GameController_i deleted."
def play(self, x, y):
    return self.game._play(x, y, self.pdtype)

def main(argv):

    print "Game Server starting..."
    orb = CORBA.ORB_init(argv, CORBA.ORB_ID)
    poa = orb.resolve_initial_references("RootPOA")
    poa._get_the_POAManager().activate()
    gf_impl = GameFactory_i(poa)
    gf_id = poa.activate_object(gf_impl)
    gf_obj = poa.id_to_reference(gf_id)
    print orb.object_to_string(gf_obj)

    # Bind the GameFactory into the Naming service. This code is
    # paranoid about checking all the things which could go wrong.
    # Normally, you would assume something this fundamental works, and
    # just die with uncaught exceptions if it didn’t.
    try:
        nameRoot = orb.resolve_initial_references("NameService")
        nameRoot = nameRoot._narrow(CosNaming.NamingContext)
        if nameRoot is None:
            print "NameService narrow failed!"
            sys.exit(1)
    except CORBA.ORB.InvalidName, ex:
        # This should never happen, since "NameService" is always a
        # valid name, even if it hadn’t been configured.
        print "Got an InvalidName exception when resolving NameService!"
        sys.exit(1)
except CORBA.NO_RESOURCES, ex:
    print "No NameService configured!"
    sys.exit(1)

except CORBA.SystemException, ex:
    print "System exception trying to resolve and narrow NameService!"
    print ex
    sys.exit(1)

# Create a new context named "tutorial"
try:
    name = [CosNaming.NameComponent("tutorial", ")
    tutorialContext = nameRoot.bind_new_context(name)

except CosNaming.NamingContext.AlreadyBound, ex:
    # There is already a context named "tutorial", so we resolve
    # that.
    print 'Reusing "tutorial" naming context.'

    tutorialContext = nameRoot.resolve(name)
    tutorialContext = tutorialContext._narrow(CosNaming.NamingContext)

    if tutorialContext is None:
        # Oh dear -- the thing called "tutorial" isn’t a
        # NamingContext. We could replace it, but it’s safer to
        # bail out.
        print 'The name "tutorial" is already bound in the NameService.'
        sys.exit(1)

    # Bind the GameServer into the "tutorial" context. Use rebind() to
    # replace an existing entry is there is one.
    tutorialContext.rebind([CosNaming.NameComponent("GameFactory","")], gf_obj)

    print "GameFactory bound in NameService."
orb.run()

if __name__ == "__main__":
    main(sys.argv)
#!/usr/bin/env python

# gameClient.py

import sys, threading
import CORBA, PortableServer
import TicTacToe, TicTacToe__POA

from Tkinter import *

class GameBrowser :

    """This class implements a top-level user interface to the game player. It lists the games currently running in the GameFactory. The user can choose to create new games, and join, watch or kill existing games."""

    def __init__(self, orb, poa, gameFactory):
        self.orb = orb
        self.poa = poa
        self.gameFactory = gameFactory
        self.initGui()
        self.getGameList()
        print "GameBrowser initialised"

    def initGui(self):
        """Initialise the Tk objects for the GUI""

        self.master = Tk()
        self.master.title("Game Client")
        self.master.resizable(0,0)

        frame = Frame(self.master)

        # List box and scrollbar
listframe = Frame(frame)
scrollbar = Scrollbar(listframe, orient=VERTICAL)
self.listbox = Listbox(listframe, exportselection = 0,
    width = 30, height = 20,
    yscrollcommand = scrollbar.set)

scrollbar.config(command = self.listbox.yview)
self.listbox.pack(side=LEFT, fill=BOTH, expand=1)
scrollbar.pack(side=RIGHT, fill=Y)

self.listbox.bind("<ButtonRelease-1>", self.selectGame)

listframe.grid(row=0, column=0, rowspan=6)

# Padding
Frame(frame, width=20).grid(row=0, column=1, rowspan=6)

# Buttons
newbutton = Button(frame, text="New game", command=self.newGame)
joinbutton = Button(frame, text="Join game", command=self.joinGame)
watchbutton = Button(frame, text="Watch game", command=self.watchGame)
killbutton = Button(frame, text="Kill game", command=self.killGame)
updatebutton = Button(frame, text="Update list", command=self.update)
quitbutton = Button(frame, text="Quit", command=frame.quit)

newbutton .config(width=15)
joinbutton .config(width=15)
watchbutton .config(width=15)
killbutton .config(width=15)
updatebutton.config(width=15)
quitbutton .config(width=15)

self.newbutton = newbutton
newbutton.bind("<ButtonRelease-1>", self.setNewButtonPosition)
newbutton .grid(row=0, column=2)
joinbutton .grid(row=1, column=2)
watchbutton .grid(row=2, column=2)
killbutton .grid(row=3, column=2)
updatebutton.grid(row=4, column=2)
quitbutton .grid(row=5, column=2)

self.newGameDialogue = None

# Padding at bottom
Frame(frame, height=10).grid(row=6, columnspan=3)

# Status bar
self.statusbar = Label(self.master,
                        text="", bd=1, relief=SUNKEN, anchor=W)
sself.statusbar.pack(side=BOTTOM, fill=X)

frame.pack(side=TOP)

def getGameList(self):
    """Get the list of games from the GameFactory, and populate
the Listbox in the GUI""

    # To make life interesting, we get the game information
    # structures one at a time from the server. It would be far
    # more sensible to get them many at a time.

    self.gameList = []
    self.listbox.delete(0,END)

    try:
        seq, iterator = self.gameFactory.listGames(0)
    except CORBA.SystemException, ex:
        print "System exception contacting GameFactory:"
        print "", CORBA.id(ex), ex
return

if len(seq) > 0:
    print "listGames() did not return an empty sequence as it should"

if iterator is None:
    print "No games in the GameFactory"
    return

try:
    more = 1
    while more:
        seq, more = iterator.next_n(1)

        for info in seq:
            # seq should only ever have one element, but loop
            # to be safe
            self.gameList.append(info)
            self.listbox.insert(END,info.name)

    iterator.destroy()

except CORBA.SystemException, ex:
    print "System exception contacting GameIterator:"
    print "", CORBA.id(ex), ex

def statusMessage(self, msg):
    self.statusbar.config(text = msg)

def selectGame(self, evt):
    selection = self.listbox.curselection()

    if selection == (): return

    index = int(selection[0])
info = self.gameList[index]

try:
    players = info.obj._get_players()
    if players == 0:
        msg = "no players yet"
    elif players == 1:
        msg = "one player waiting"
    else:
        msg = "game in progress"

except CORBA.SystemException, ex:
    print "System exception contacting Game:"
    print "  ", CORBA.id(ex), ex
    msg = "error contacting Game object"

self.statusMessage("%s: %s" % (info.name, msg))

def setNewButtonPosition(self, evt):
    self._new_x = self.master.winfo_x() + self.newbutton.winfo_x() + evt.x
    self._new_y = self.master.winfo_y() + self.newbutton.winfo_y() + evt.y

def newGame(self):
    if self.newGameDialogue:
        self.newGameDialogue.destroy()

    self.newGameDialogue = toplevel = Toplevel(self.master)
    toplevel.transient()
    toplevel.title("New game...")
    toplevel.geometry("+%d+%d" % (self._new_x, self._new_y))
    Label(toplevel, text="Enter name for new game").pack()
    entry = Entry(toplevel)
    entry.pack()
entry.focus()

entry.bind("<Key-Return>", self.newGameEntered)

def newGameEntered(self, evt):
    name = evt.widget.get()
    self.newGameDialogue.destroy()
    self.newGameDialogue = None

    if name == "":
        self.statusMessage("You must give a non-empty name")
        return

    try:
        game = self.gameFactory.newGame(name)
    except TicTacToe.GameFactory.NameInUse:
        self.statusMessage("Game name in use")
        return
    except CORBA.SystemException, ex:
        print "System exception trying to create new game:"
        print "", CORBA.id(ex), ex
        self.statusMessage("System exception trying to create new game")
        return

    self.getGameList()

def joinGame(self):
    selection = self.listbox.curselection()
    if selection == (): return

    index = int(selection[0])
    info = self.gameList[index]
pi = Player_i(self.master, info.name)
id = poa.activate_object(pi)
po = poa.id_to_reference(id)
try:
    controller, type = info.obj.joinGame(po)
    if type == TicTacToe.Nought:
        stype = "noughts"
    else:
        stype = "crosses"

    pi.go(info.obj, controller, stype)
    self.statusMessage("%s: joined game as %s" % (info.name, stype))
except TicTacToe.Game.CannotJoin, ex:
    poa.deactivate_object(id)
    self.statusMessage("%s: cannot join game" % info.name)
except CORBA.SystemException, ex:
    poa.deactivate_object(id)
    print "System exception trying to join game:", CORBA.id(ex), ex
    self.statusMessage("%s: system exception contacting game" % \
                        info.name)
    self.getGameList()

def watchGame(self):
    selection = self.listBox.curselection()
    if selection == (): return

    index = int(selection[0])
    info = self.gameList[index]

    si = Spectator_i(self.master, info.name)
id = poa.activate_object(si)
so = poa.id_to_reference(id)
try:
    cookie, state = info.obj.watchGame(so)
    si.go(info.obj, cookie, state)

    self.statusMessage("Watching %s" % info.name)

except CORBA.SystemException, ex:
    poa.deactivate_object(id)
    print "System exception trying to watch game:", CORBA.id(ex), ex
    self.statusMessage("%s: system exception contacting game" % \
                         info.name)
    self.getGameList()

def update(self):
    self.getGameList()

def killGame(self):
    selection = self.listbox.curselection()
    if selection == (): return

    index = int(selection[0])
    info = self.gameList[index]

    try:
        info.obj.kill()
        msg = "killed"
    except CORBA.SystemException, ex:
        print "System exception trying to kill game:", CORBA.id(ex), ex
        msg = "error contacting object"

    self.statusMessage("%s: %s" % (info.name, msg))
self.getGameList()

class Player_i (TicTacToe__POA.Player):

    def __init__(self, master, name):
        self.master = master
        self.name = name
        print "Player_i created"

    def __del__(self):
        print "Player_i deleted"

    # CORBA methods
    def yourGo(self, state):
        self.drawState(state)
        self.statusMessage("Your go")

    def end(self, state, winner):
        self.drawState(state)
        if winner == TicTacToe.Nought:
            self.statusMessage("Noughts wins")
        elif winner == TicTacToe.Cross:
            self.statusMessage("Crosses wins")
        else:
            self.statusMessage("It's a draw")
        self.toplevel = None

    def gameAborted(self):
        self.statusMessage("Game aborted!")
        self.toplevel = None

    # Implementation details
    def go(self, game, controller, type):
        self.game = game
        self.controller = controller
self.type = type

self.toplevel = Toplevel(self.master)
self.toplevel.title("%s (%s)" % (self.name, type))

self.canvas = Canvas(self.toplevel, width=300, height=300)
self.canvas.pack()

self.canvas.create_line(100, 0, 100, 300, width=5)
self.canvas.create_line(200, 0, 200, 300, width=5)
self.canvas.create_line(0, 100, 300, 100, width=5)
self.canvas.create_line(0, 200, 300, 200, width=5)

self.canvas.bind("<ButtonRelease-1>", self.click)
self.toplevel.bind("<Destroy>", self.close)

self.statusbar = Label(self.toplevel,
    text="", bd=1, relief=SUNKEN, anchor=W)
self.statusbar.pack(side=BOTTOM, fill=X)

def statusMessage(self, msg):
    if self.toplevel:
        self.statusbar.config(text = msg)

def click(self, evt):
    x = evt.x / 100
    y = evt.y / 100
    try:
        self.statusMessage("Waiting for other player...")
        state = self.controller.play(x, y)
        self.drawState(state)

    except TicTacToe.GameController.SquareOccupied:
        self.statusMessage("Square already occupied")
except TicTacToe.GameController.NotYourGo:
    self.statusMessage("Not your go")

except TicTacToe.GameController.InvalidCoordinates:
    self.statusMessage("Eek! Invalid coordinates")

except CORBA.SystemException:
    print "System exception trying to contact GameController:
print "
    print "", CORBA.id(ex), ex
    self.statusMessage("System exception contacting GameController!")

def close(self, evt):
    if self.toplevel:
        self.toplevel = None
        try:
            self.game.kill()
        except CORBA.SystemException, ex:
            print "System exception trying to kill game:
print "
            print "", CORBA.id(ex), ex

        id = poa.servant_to_id(self)
        poa.deactivate_object(id)

    def drawNought(self, x, y):
        cx = x * 100 + 20
        cy = y * 100 + 20
        self.canvas.create_oval(cx, cy, cx+60, cy+60,
                              outline="darkgreen", width=5)

    def drawCross(self, x, y):
        cx = x * 100 + 30
        cy = y * 100 + 30
        self.canvas.create_line(cx, cy, cx+40, cy+40,
                                fill="darkred", width=5)
        self.canvas.create_line(cx, cy+40, cx+40, cy,
def drawState(self, state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == TicTacToe.Nought:
                self.drawNought(i, j)
            elif state[i][j] == TicTacToe.Cross:
                self.drawCross(i, j)

class Spectator_i (TicTacToe__POA.Spectator):

    def __init__(self, master, name):
        self.master = master
        self.name = name
        print "Spectator_i created"

    def __del__(self):
        print "Spectator_i deleted"

    # CORBA methods
    def update(self, state):
        self.drawState(state)

    def end(self, state, winner):
        self.drawState(state)
        if winner == TicTacToe.Nought:
            self.statusMessage("Noughts wins")
        elif winner == TicTacToe.Cross:
            self.statusMessage("Crosses wins")
        else:
            self.statusMessage("It’s a draw")
            self.toplevel = None

    def gameAborted(self):
```python
self.statusMessage("Game aborted!")
self.toplevel = None

# Implementation details

def go(self, game, cookie, state):
    self.game = game
    self.cookie = cookie

    self.toplevel = Toplevel(self.master)
    self.toplevel.title("Watching %s" % self.name)

    self.canvas = Canvas(self.toplevel, width=300, height=300)
    self.canvas.pack()

    self.canvas.create_line(100, 0, 100, 300, width=5)
    self.canvas.create_line(200, 0, 200, 300, width=5)
    self.canvas.create_line(0, 100, 300, 100, width=5)
    self.canvas.create_line(0, 200, 300, 200, width=5)

    self.toplevel.bind("<Destroy>", self.close)

    self.statusbar = Label(self.toplevel,
                           text="", bd=1, relief=SUNKEN, anchor=W)
    self.statusbar.pack(side=BOTTOM, fill=X)
    self.drawState(state)

def statusMessage(self, msg):
    self.statusbar.config(text = msg)

def close(self, evt):
    if self.toplevel:
        self.toplevel = None
        try:
            self.game.unwatchGame(self.cookie)
        except CORBA.SystemException, ex:
            pass
```

print "System exception trying to unwatch game:"
print " ", CORBA.id(ex), ex

id = poa.servant_to_id(self)
poa.deactivate_object(id)

def drawNought(self, x, y):
    cx = x * 100 + 20
    cy = y * 100 + 20
    self.canvas.create_oval(cx, cy, cx+60, cy+60,
        outline="darkgreen", width=5)

def drawCross(self, x, y):
    cx = x * 100 + 30
    cy = y * 100 + 30
    self.canvas.create_line(cx, cy, cx+40, cy+40,
        fill="darkred", width=5)
    self.canvas.create_line(cx, cy+40, cx+40, cy,
        fill="darkred", width=5)

def drawState(self, state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == TicTacToe.Nought:
                self.drawNought(i, j)
            elif state[i][j] == TicTacToe.Cross:
                self.drawCross(i, j)

orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
poa = orb.resolve_initial_references("RootPOA")
poa._get_the_POAManager().activate()

# Get the GameFactory reference using a corbaname URI. On a pre-CORBA
# 2.4 ORB, this would have to explicitly contact the naming service.
try:
    gameFactory = orb.string_to_object("corbaname:rir:#tutorial/GameFactory")
    gameFactory = gameFactory._narrow(TicTacToe.GameFactory)
except CORBA.BAD_PARAM, ex:
    # string_to_object throws BAD_PARAM if the name cannot be resolved
    print "Cannot find the GameFactory in the naming service."
    sys.exit(1)
except CORBA.SystemException, ex:
    # This might happen if the naming service is dead, or the narrow
    # tries to contact the object and it is not there.
    print "CORBA system exception trying to get the GameFactory reference:"
    print "  ", CORBA.id(ex), ex
    sys.exit(1)

# Start the game browser
browser = GameBrowser(orb, poa, gameFactory)

# Run the Tk mainloop in a separate thread

def tkloop():
    browser.master.mainloop()
    print "Shutting down the ORB..."
    orb.shutdown(0)

    threading.Thread(target=tkloop).start()

# Run the ORB main loop (not necessary with omniORBpy, but may be
# necessary with other ORBs. According to the CORBA specification,
# orb.run() must be given the main thread.
orb.run()