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.
  - `http://www.omg.org/technology/documents/formal/python_language_mapping.htm`
- The CORBA to Python mapping is extremely simple:
  - C++ mapping specification: 166 pages
  - Java: 130 pages
  - Python: 16 pages!

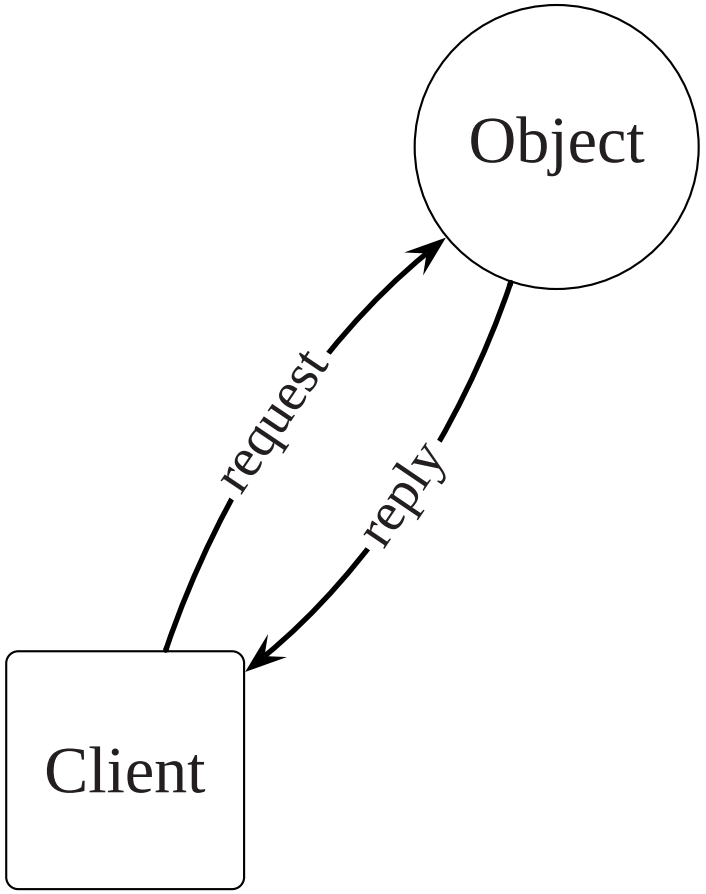
Python ORBs

- omniORBpy
  - Based on C++ omniORB. Multi-threaded. Free (LGPL).
  - `www.omniorb.org/omniORBpy`
- orbit-python
  - Based on C ORBit. Single-threaded. Free (LGPL).
  - `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`
- ILU
  - Based on C ILU. More than just CORBA. Open source. Dead?
  - `ftp.parc.xerox.com/pub/ilu/ilu.html`

A simple example

1. A picture
2. IDL, the Interface Definition Language
3. Client code
4. Server code

A picture



- 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

```
>>> 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
```

Python server

```
2  import sys, CORBA, CosNaming, Snake, Snake__POA
3
4  class Adder_i (Snake__POA.Adder):
5      def __init__(self):
6          self.value = 0
7
8      def accumulate(self, a):
9          self.value = self.value + a
10         return self.value
11
12     def reset(self):
13         self.value = 0
14
15 orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
16 poa = orb.resolve_initial_references("RootPOA")
17
18 adderServant = Adder_i()
19 poa.activate_object(adderServant)
20 adderObjref  = adderServant._this()
21
22 nameRoot = orb.resolve_initial_references("NameService")
23 nameRoot = nameRoot._narrow(CosNaming.NamingContext)
24 name     = [CosNaming.NameComponent("adder", "obj")]
25 nameRoot.rebind(name, adderObjref)
26
27 poa._get_the_POAManager().activate()
28 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

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 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. . .

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

Fixed point

`fixed<`*digits*`,`*scale*`>` $\quad 1 \leq digits \leq 31$, $0 \leq scale \leq digits$

IDL

Fixed point value with *digits* total decimal digits, including *scale* decimal places.

e.g. `fixed<5,0>` : range ± 99999
`fixed<5,2>` : range ± 999.99
`fixed<5,5>` : range ± 0.99999

Python

```
f = CORBA.fixed(5, 2, 12345)
f.value()     -> 12345L
f.precision() -> 5
f.decimals()  -> 2
g = f * 2
h = f + g
i = MyFixed(23456)
```

123.45
actual value $\times 10^{scale}$ as a long
digits
scale
maths with integers...
... and between fixeds
where `MyFixed` is a typedef (see later).

Constants

IDL

```
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;
};
```

Python

```
>>> import M
>>> M.TWO
2
>>> M.initial
'D'
>>> M.title
'CORBA Tutorial'
>>> M.title = "not so constant"  # Oh dear!
```

Modules

IDL

```
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
```

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;
  };
};
```

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.

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

IDL

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

Python

```
>>> 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.
```

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;
  };
};
```

Python

```
>>> u = M.Another(M.blue, 5)  #  Constructor giving discriminator, value.
>>> u.message = "Hi"          #  Discriminator now red or green.
```

Unions

- Default case is optional.

IDL

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

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;
};
```

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]]
```

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

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

Python

```
raise M.Error()
raise M.Invalid("Tutorial 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)
```

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.

```
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!

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.

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*`)`

```
tc = CORBA.TypeCode(CORBA.id(MyModule.MyStruct))
```

IDL: Summary

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

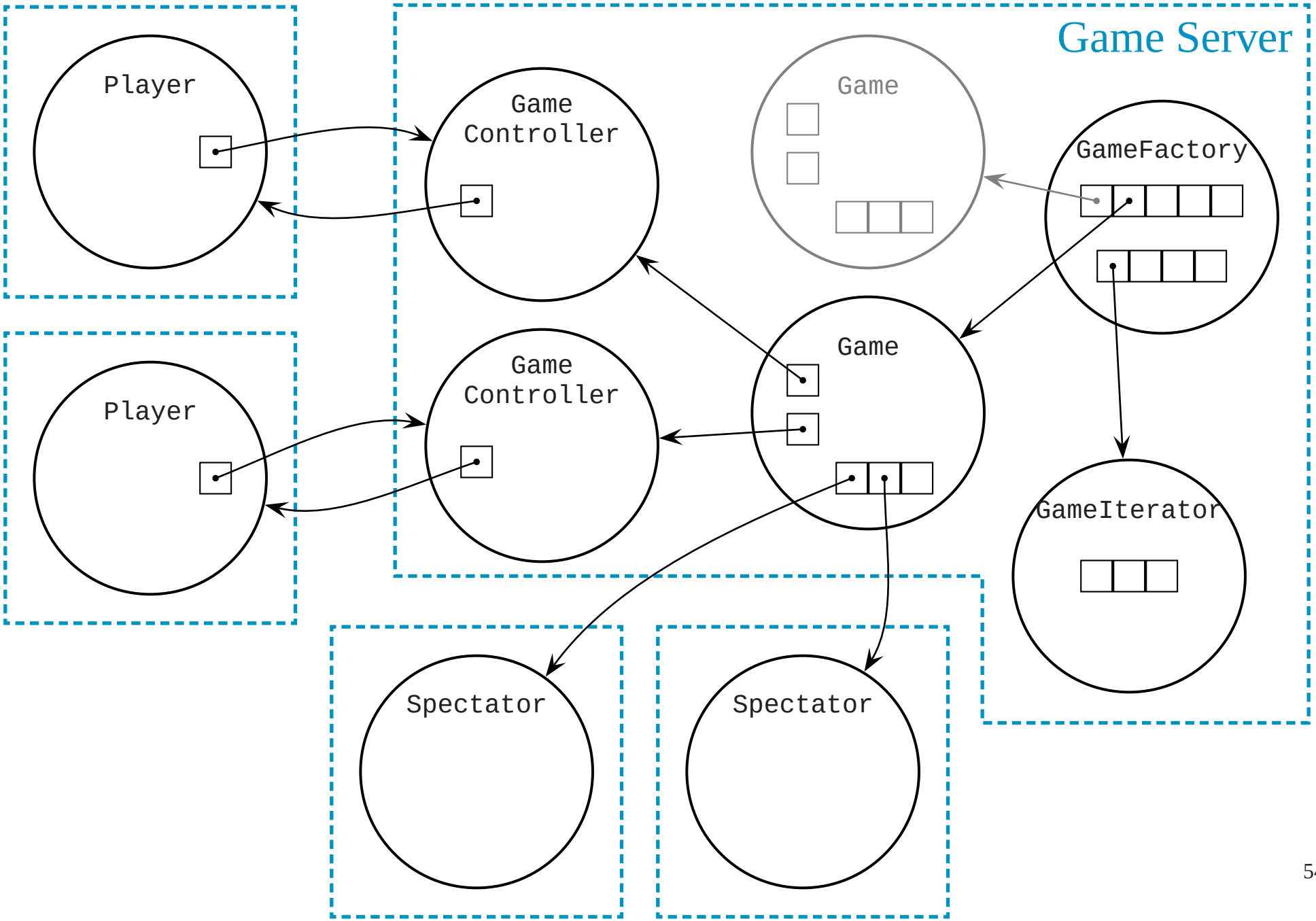
Object life cycle

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

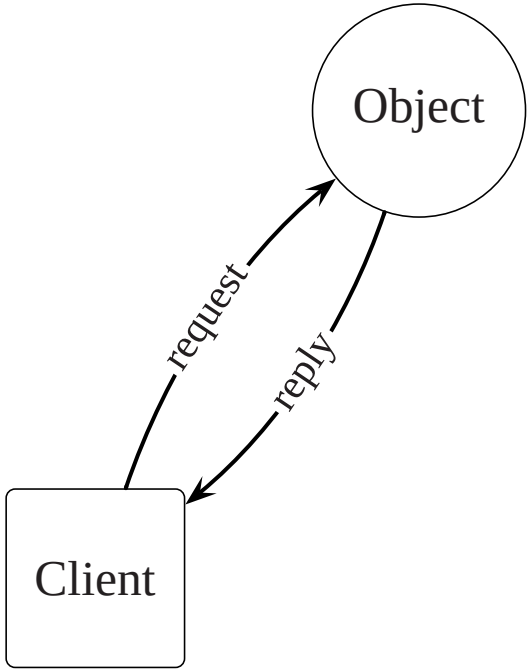
Example application

- 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++.

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.

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.

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);
};
```

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

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

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

```
IOR:010000001900000049444c3a5475746f7269616c2f4578616d706c653a312e30
000000000100000000000000480000000101020007000006d79686f73740000d204
00001c0000005761732069742076f72746820747970696e67207468697320696e3f
010000000000000008000000100000000545441
```

- 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`

Function	Returns	Client sees
`getObject()`	`I`	Probably `I`
`getI()`	`I`	Definitely `I`
`getObject()`	`K`	Definitely `Object`
`getI()`	`K`	Maybe `I`

Narrowing

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

```
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:

```
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:

```
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.

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
```

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.

Naming service

```
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:

```
module 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
```

Bulk data

- Sequences
  - Simple, but can't cope with *really* large items.
- Iterator pattern. e.g. GameIterator:

```
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.

```
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.

```
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.

Garbage collection

- 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.

Garbage collection

```
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.

```
// IDL for a rather over-engineered distributed noughts and crosses game

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.
  };

```

```
36  interface GameIterator {
37    GameInfoSeq next_n(in unsigned long how_many, out boolean more);
38    // Return the next sequence of games, up to a maximum of
39    // how_many. If more is true, there are more games to list.
40
41    void destroy();
42    // Destroy the iterator object.
43  };
44
45  interface Game {
46    readonly attribute string    name;    // Name of this game.
47    readonly attribute short     players; // Number of players registered.
48    readonly attribute GameState state;   // Current state of the game.
49
50    exception CannotJoin {};
51
52    GameController joinGame(in Player p, out PlayerType t)
53      raises (CannotJoin);
54    // Join a new game, passing in a Player object reference. Returns
55    // a GameController object reference used to play the game. The
56    // out argument lets the player know whether they are noughts or
57    // crosses.
58
59    unsigned long watchGame  (in Spectator s, out GameState state);
60    void          unwatchGame(in unsigned long cookie);
61    // Register or unregister a spectator for the game. watchGame()
62    // returns a cookie to be used to unregister. Note the potential
63    // for unregistering other spectators. This should really use an
64    // event or notification service.
65
66    void kill();
67    // Kill the game prematurely.
68  };
69
70  interface GameController {
```

```
71      exception SquareOccupied {};
72      exception InvalidCoordinates {};
73      exception NotYourGo {};
74
75      GameState play(in short x, in short y)
76        raises (SquareOccupied, InvalidCoordinates, NotYourGo);
77      // Place a piece at the specified coordinates. Returns the new
78      // game state.
79    };
80
81    interface Player {
82      void yourGo(in GameState state);
83      // Tell the player it is their go, giving the current game state.
84
85      void end(in GameState state, in PlayerType winner);
86      // End of game. winner is Nobody if the game is tied.
87
88      void gameAborted();
89    };
90
91    interface Spectator {
92      void update(in GameState state);
93      // Update the current state of the game.
94
95      void end(in GameState state, in PlayerType winner);
96      void gameAborted();
97    };
98  };
```

```
#!/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)
```

```
71
72          # Create iterator if necessary
73          if rest:
74              iter = GameIterator_i(self, self.iterator_poa, rest)
75              iid  = self.iterator_poa.activate_object(iter)
76              iobj = self.iterator_poa.id_to_reference(iid)
77              self.lock.acquire()
78              self.iterators[iid] = iter
79              self.lock.release()
80          else:
81              iobj = None # Nil object reference
82
83          return (ret, iobj)
84
85      def _removeGame(self, name):
86          self.lock.acquire()
87          for i in range(len(self.games)):
88              if self.games[i][0] == name:
89                  del self.games[i]
90                  break
91          self.lock.release()
92
93      def _removeIterator(self, iid):
94          self.lock.acquire()
95          del self.iterators[iid]
96          self.lock.release()
97
98  class GameIterator_i (TicTacToe__POA.GameIterator):
99
100     def __init__(self, factory, poa, games):
101         self.factory = factory
102         self.poa     = poa
103         self.games   = games
104         self.tick    = 1 # Tick for time-out garbage collection
105         print "GameIterator_i created."
```

```
106
107     def __del__(self):
108         print "GameIterator_i deleted."
109
110     def next_n(self, how_many):
111         self.tick  = 1
112         front      = self.games[:int(how_many)]
113         self.games = self.games[int(how_many):]
114
115         # Convert internal representation to GameInfo sequence
116         ret = map(lambda g: TicTacToe.GameInfo(g[0], g[2]), front)
117
118         if self.games:
119             more = 1
120         else:
121             more = 0
122
123         return (ret, more)
124
125     def destroy(self):
126         id = self.poa.servant_to_id(self)
127         self.factory._removeIterator(id)
128         self.poa.deactivate_object(id)
129
130 class IteratorScavenger (threading.Thread):
131     def __init__(self, factory):
132         threading.Thread.__init__(self)
133         self.setDaemon(1)
134         self.factory = factory
135         self.start()
136
137     def run(self):
138         print "Iterator scavenger running..."
139
140         lock      = self.factory.lock
```

```
141         iterators = self.factory.iterators
142         poa       = self.factory.iterator_poa
143         manager   = poa._get_the_POAManager()
144 
145         while 1:
146             time.sleep(SCAVENGER_INTERVAL)
147 
148             print "Scavenging dead iterators..."
149 
150             # Bonus points for spotting why we hold requests...
151             manager.hold_requests(1)
152             lock.acquire()
153 
154             for id, iter in iterators.items():
155                 if iter.tick == 1:
156                     iter.tick = 0
157                 else:
158                     del iterators[id]
159                     poa.deactivate_object(id)
160 
161                 # This del drops the last reference to the iterator so
162                 # it can be collected immediately. Without it, the
163                 # Python servant object stays around until the next
164                 # time around the loop.
165                 del iter
166 
167             lock.release()
168             manager.activate()
169 
170 class Game_i (TicTacToe__POA.Game):
171 
172     def __init__(self, factory, name, poa):
173         self.factory = factory
174         self.name    = name
175         self.poa     = poa
```

```
176         self.lock    = threading.Lock()
177 
178         n = TicTacToe.Nobody
179 
180         self.players = 0
181         self.state   = [[n,n,n],
182                         [n,n,n],
183                         [n,n,n]]
184 
185         self.p_noughts         = None
186         self.p_crosses         = None
187         self.whose_go          = TicTacToe.Nobody
188         self.spectators        = []
189         self.spectatorNotifier = SpectatorNotifier(self.spectators, self.lock)
190 
191         print "Game_i created."
192 
193     def __del__(self):
194         print "Game_i deleted."
195 
196     def _get_name(self):
197         return self.name
198 
199     def _get_players(self):
200         return self.players
201 
202     def _get_state(self):
203         return self.state
204 
205     def joinGame(self, player):
206         try:
207             self.lock.acquire()
208 
209             if self.players == 2:
210                 raise TicTacToe.Game.CannotJoin()
```

```
211
212             if self.players == 0:
213                 ptype = TicTacToe.Nought
214                 self.p_noughts = player
215             else:
216                 ptype = TicTacToe.Cross
217                 self.p_crosses = player
218
219                 # Notify the noughts player that it's their go
220                 try:
221                     self.whose_go = TicTacToe.Nought
222                     self.p_noughts.yourGo(self.state)
223                 except (CORBA.COMM_FAILURE, CORBA.OBJECT_NOT_EXIST), ex:
224                     print "Lost contact with player"
225                     self.kill()
226
227             # Create a GameController
228             gc   = GameController_i(self, ptype)
229             id   = self.poa.activate_object(gc)
230             gobj = self.poa.id_to_reference(id)
231
232             self.players = self.players + 1
233
234         finally:
235             self.lock.release()
236
237         return (gobj, ptype)
238
239     def watchGame(self, spectator):
240         self.lock.acquire()
241         cookie = len(self.spectators)
242         self.spectators.append(spectator)
243         self.lock.release()
244         return cookie, self.state
245
```

```
    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()
```

```
316         return self.state
317
318     def _checkForWinner(self):
319         """If there is a winner, return the winning player's type. If
320         the game is a tie, return Nobody, otherwise return None."""
321
322         # Rows
323         for i in range(3):
324             if self.state[i][0] == self.state[i][1] and \
325                self.state[i][1] == self.state[i][2] and \
326                self.state[i][0] != TicTacToe.Nobody:
327                 return self.state[i][0]
328
329         # Columns
330         for i in range(3):
331             if self.state[0][i] == self.state[1][i] and \
332                self.state[1][i] == self.state[2][i] and \
333                self.state[0][i] != TicTacToe.Nobody:
334                 return self.state[0][i]
335
336         # Top-left to bottom-right
337         if self.state[0][0] == self.state[1][1] and \
338            self.state[1][1] == self.state[2][2] and \
339            self.state[0][0] != TicTacToe.Nobody:
340             return self.state[0][0]
341
342         # Bottom-left to top-right
343         if self.state[0][2] == self.state[1][1] and \
344            self.state[1][1] == self.state[2][0] and \
345            self.state[0][2] != TicTacToe.Nobody:
346             return self.state[0][2]
347
348         # Return None if the game is not full
349         for i in range(3):
350             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."

```

```
421     def play(self, x, y):
422         return self.game._play(x, y, self.ptype)
423 
424 def main(argv):
425 
426     print "Game Server starting..."
427 
428     orb = CORBA.ORB_init(argv, CORBA.ORB_ID)
429     poa = orb.resolve_initial_references("RootPOA")
430 
431     poa._get_the_POAManager().activate()
432 
433     gf_impl = GameFactory_i(poa)
434     gf_id   = poa.activate_object(gf_impl)
435     gf_obj  = poa.id_to_reference(gf_id)
436 
437     print orb.object_to_string(gf_obj)
438 
439     # Bind the GameFactory into the Naming service. This code is
440     # paranoid about checking all the things which could go wrong.
441     # Normally, you would assume something this fundamental works, and
442     # just die with uncaught exceptions if it didn't.
443     try:
444         nameRoot = orb.resolve_initial_references("NameService")
445         nameRoot = nameRoot._narrow(CosNaming.NamingContext)
446         if nameRoot is None:
447             print "NameService narrow failed!"
448             sys.exit(1)
449 
450     except CORBA.ORB.InvalidName, ex:
451         # This should never happen, since "NameService" is always a
452         # valid name, even if it hadn't been configured.
453 
454         print "Got an InvalidName exception when resolving NameService!"
455         sys.exit(1)
```

```
456
457     except CORBA.NO_RESOURCES, ex:
458         print "No NameService configured!"
459         sys.exit(1)
460
461     except CORBA.SystemException, ex:
462         print "System exception trying to resolve and narrow NameService!"
463         print ex
464         sys.exit(1)
465
466     # Create a new context named "tutorial"
467     try:
468         name = [CosNaming.NameComponent("tutorial", "")]
469         tutorialContext = nameRoot.bind_new_context(name)
470
471     except CosNaming.NamingContext.AlreadyBound, ex:
472         # There is already a context named "tutorial", so we resolve
473         # that.
474         print 'Reusing "tutorial" naming context.'
475
476         tutorialContext = nameRoot.resolve(name)
477         tutorialContext = tutorialContext._narrow(CosNaming.NamingContext)
478
479         if tutorialContext is None:
480             # Oh dear -- the thing called "tutorial" isn't a
481             # NamingContext. We could replace it, but it's safer to
482             # bail out.
483             print 'The name "tutorial" is already bound in the NameService.'
484             sys.exit(1)
485
486     # Bind the GameServer into the "tutorial" context. Use rebind() to
487     # replace an existing entry is there is one.
488     tutorialContext.rebind([CosNaming.NameComponent("GameFactory","")], gf_obj)
489
490     print "GameFactory bound in NameService."
```

```
491
492     orb.run()
493
494 if __name__ == "__main__":
495     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)
        self.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
```

```
106             return
107
108         if len(seq) > 0:
109             print "listGames() did not return an empty sequence as it should"
110
111         if iterator is None:
112             print "No games in the GameFactory"
113             return
114
115         try:
116             more = 1
117             while more:
118                 seq, more = iterator.next_n(1)
119
120                 for info in seq:
121                     # seq should only ever have one element, but loop
122                     # to be safe
123                     self.gameList.append(info)
124                     self.listbox.insert(END,info.name)
125
126             iterator.destroy()
127
128         except CORBA.SystemException, ex:
129             print "System exception contacting GameIterator:"
130             print "  ", CORBA.id(ex), ex
131
132     def statusMessage(self, msg):
133         self.statusbar.config(text = msg)
134
135     def selectGame(self, evt):
136         selection = self.listbox.curselection()
137
138         if selection == (): return
139
140         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:"
            print "  ", 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)
```

```
246         so = poa.id_to_reference(id)
247         try:
248             cookie, state = info.obj.watchGame(so)
249             si.go(info.obj, cookie, state)
250 
251             self.statusMessage("Watching %s" % info.name)
252 
253         except CORBA.SystemException, ex:
254             poa.deactivate_object(id)
255             print "System exception trying to watch game:"
256             print "  ", CORBA.id(ex), ex
257             self.statusMessage("%s: system exception contacting game" % \
258                                info.name)
259             self.getGameList()
260 
261     def update(self):
262         self.getGameList()
263 
264     def killGame(self):
265         selection = self.listbox.curselection()
266         if selection == (): return
267 
268         index = int(selection[0])
269         info  = self.gameList[index]
270 
271         try:
272             info.obj.kill()
273             msg = "killed"
274 
275         except CORBA.SystemException, ex:
276             print "System exception trying to kill game:"
277             print "  ", CORBA.id(ex), ex
278             msg = "error contacting object"
279 
280         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
```

```
316         self.type       = type
317 
318         self.toplevel   = Toplevel(self.master)
319         self.toplevel.title("%s (%s)" % (self.name, type))
320 
321         self.canvas = Canvas(self.toplevel, width=300, height=300)
322         self.canvas.pack()
323 
324         self.canvas.create_line(100,   0, 100, 300, width=5)
325         self.canvas.create_line(200,   0, 200, 300, width=5)
326         self.canvas.create_line(0,   100, 300, 100, width=5)
327         self.canvas.create_line(0,   200, 300, 200, width=5)
328 
329         self.canvas.bind("<ButtonRelease-1>", self.click)
330         self.toplevel.bind("<Destroy>", self.close)
331 
332         self.statusbar = Label(self.toplevel,
333                                text="", bd=1, relief=SUNKEN, anchor=W)
334         self.statusbar.pack(side=BOTTOM, fill=X)
335 
336     def statusMessage(self, msg):
337         if self.toplevel:
338             self.statusbar.config(text = msg)
339 
340     def click(self, evt):
341         x = evt.x / 100
342         y = evt.y / 100
343         try:
344             self.statusMessage("Waiting for other player...")
345             state = self.controller.play(x, y)
346             self.drawState(state)
347 
348         except TicTacToe.GameController.SquareOccupied:
349             self.statusMessage("Square already occupied")
350 
```

```
        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 "  ", 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 "  ", 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)

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):
```

```
421         self.statusMessage("Game aborted!")
422         self.toplevel = None
423 
424     # Implementation details
425     def go(self, game, cookie, state):
426         self.game   = game
427         self.cookie = cookie
428 
429         self.toplevel = Toplevel(self.master)
430         self.toplevel.title("Watching %s" % self.name)
431 
432         self.canvas = Canvas(self.toplevel, width=300, height=300)
433         self.canvas.pack()
434 
435         self.canvas.create_line(100,   0, 100, 300, width=5)
436         self.canvas.create_line(200,   0, 200, 300, width=5)
437         self.canvas.create_line(0,   100, 300, 100, width=5)
438         self.canvas.create_line(0,   200, 300, 200, width=5)
439 
440         self.toplevel.bind("<Destroy>", self.close)
441 
442         self.statusbar = Label(self.toplevel,
443                                text="", bd=1, relief=SUNKEN, anchor=W)
444         self.statusbar.pack(side=BOTTOM, fill=X)
445         self.drawState(state)
446 
447     def statusMessage(self, msg):
448         self.statusbar.config(text = msg)
449 
450     def close(self, evt):
451         if self.toplevel:
452             self.toplevel = None
453             try:
454                 self.game.unwatchGame(self.cookie)
455             except CORBA.SystemException, ex:
```

```
                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()
```