Common Object Request Broker Architecture
(CORBA)

RPC Problems:

1) Have to name server machine, can't move service.
2) C, not C++, not language independent

CORBA Goals:

1) Simplify development of distributed applications.

2) Decouple machine—service relationship
(allow services to move)

3) Provide a foundation for distributed object cooperation.

4) Provide for OO programmers what RPC provides for C.

Advantages over RPC
1) Object abstractions used instead of procedural abstractions.
2) Inheritance supported.

Other Differences
Supports multi-threaded services

Components

1) Object Request Broker.
2) Interoperability Specification
3) Interface definition language
4) Program language mappings for IDL
5) Static invocation interface
6) Dynamic invocation interface
7) Dynamic skeleton interface
8) Basic object adapter
9) Interface and implementation repositories

Object Request Broker

Provides communication for exchanging requests between local and remote objects.
Clients may bind to a service by using a broker or by explicitly naming the server.

Interoperability Specification (GIOP, IIOP)

Specifies protocols for communication to ORB

Several clients can share a connection to an ORB.

ORBs can communicate with each other.

Common Data Representation (CDR), (like XDR)

Objects (Servers) can be located dynamically,
objects can migrate to other machines transparently

Seven message types:
Client – `Request`, `CancelRequest`, `LocateRequest`
Server – `Reply`, `LocateReply`, `CloseConnection`
Both – `MessageError`

IDL

Needed–something to define interfaces to remote objects (similar to what the `.x` file does for RPCs.

Solution–Interface Definition Language (similar to C++)

Goals:

Platform independence.

Enforced modularity.

Easy to program.

Language independent.

Object oriented
–specify interfaces that have operations and attributes.
support inheritance

Language mappings: C, C++, Smalltalk, COBOL, Modula 3, DCE, Java.

More IDL

Supported features:

Modules, interfaces, operations, attributes, inheritance, basic types (double, long, char), arrays, sequence, struct, enum, union, typedef, consts, exceptions

C++ differences: no data members, no pointers, no overloading, no `int`, no templates, tagged unions only

parameter passing modes, different string type, different sequence type, different exception interface, oneway call semantics

no constructors/destructors, no control constructs (like the `.x` file–no programming)

Static Invocation Interface– Methods are specified in advance by the IDL.

Dynamic Invocation Interface– Objects and methods can be specified at runtime.
Meta-data is stored in "Interface Repository"
May query an objects interface (what methods do you support) at runtime.

Object Access Methodology

1) Client calls object proxy
(similar `cif` stub)

2) Proxy calls object broker, finds out where the server object lives. (Clients may bind to a service using the broker or by explicitly naming the server.)

3) Proxy calls object, gets response.

4) Proxy returns response to client.

Object Reference: a “handle”
(Similar to the RPC handle)
use it only for passing to the proxy.
Binding client to a target object:
Request an object reference (handle) from a server
The object reference serves as the local proxy.
Invoke methods on the proxy.
Warning: do not use this pointer to access the object;

The ORB activates (starts) any idle service when any method of that service is invoked.

Call-backs can be set up. (Notifier service)
A client can request to be called when an event occurs.

The Echo Example

Goal: Echo a string.

Programming steps:

1. define the object interface in IDL
2. use the IDL compiler to generate the stub
3. build the object implementation
4. build the client implementation

The Echo IDL

File: `echo.idl`

```
#ifndef __ECHO_IDL__
#define __ECHO_IDL__
interface Echo {
   struct Line {
     char data[20];
   };
   Line echoString(in Line mesg);
};
#endif
```

The "`ifndef`" is so the same idl compile doesn't include this twice (not necessary in a simple project).

The IDL says the `Echo` object has a method called `echoString` that takes a string and returns a string.

The OmniORB IDL compiler

```
omniidl -bcxx echo.idl
```

Produces:
`echo.hh` – C++ header file
`echoSK.cc` – C++ source (stub) file (SKeleton Class)
`echoDynSK.cc` – another stub (we don't use this)

A look at the stub declaration (header file)

```
class Echo;
typedef Echo* Echo_ptr;
class Echo {
public:
 typedef Echo_ptr _ptr_type;
 static _ptr_type _duplicate(_ptr_type);
 static _ptr_type _narrow(CORBA::Object_ptr);
 static _ptr_type _nil();
 struct Line { CORBA::Char data[20]; };
 virtual Line  echoString(const Line & mesg) = 0; };
class _objref_Echo: public virtual CORBA::Object,
  public virtual omniObjRef {
public:
  Echo::Line echoString(const Echo::Line& mesg); };
class _impl_Echo: public virtual omniServant {
  virtual Echo::Line
    echoString(const Echo::Line& mesg) = 0; };
```

`_ptr_type` – (a type) points to an `Echo` object
`_duplicate`–make a second reference to the object
`_narrow`–Restrict to being an Echo object
`_nil`–returns an official null pointer (no object)
`is_nil`–official null pointer check
`Echo::Line` exists twice (on both client and server)

A look at the skeleton class (header file)

`_sk_Echo` is the generated skeleton class.

```
class _sk_Echo : public virtual Echo {
public:
 _sk_Echo() {}
 _sk_Echo(const omniORB::objectKey& k);
 virtual ~_sk_Echo();
 Echo_ptr _this() { return Echo::_duplicate(this); }
 void _obj_is_ready(CORBA::BOA_ptr boa)
   { boa->obj_is_ready(this); }
 CORBA::BOA_ptr _boa()
   { return CORBA::BOA::getBOA(); }
 void _dispose() { _boa()->dispose(this); }
 omniORB::objectKey _key();
 virtual Line  echoString(const Line & mesg) = 0;
 ...
};
```

These methods will be available to the object.

The Object Implementation

(you must build this)

```
// File: echo_i.cc
#include "echo.hh"
class Echo_i : public POA_Echo,
  public  PortableServer::RefCountServantBase {
public:
 Echo_i() {}
 virtual ~Echo_i() {}
 struct Line echoString(const struct Line & mesg){
  return mesg;
 };
};
```

Constructor and destructor–default.

The `echoString` is built by you to implement the capability you want to provide.

Note: In building objects, you often use the heap (new)

The Actual Client

(OmniORB style) Do it so all methodology dependent code is in `main`.
We are passed a pointer to the proxy

```
// File: do_echo.cc
#include <iostream>
using namespace std;
int do_Echo(Echo_ptr Echo_Obj ) {
 struct Echo::Line original;
 struct Echo::Line new_version;
 strcpy((char *)original.data, "Hi there" );
 new_version = Echo_Obj->echoString( original );
 cout << new_version.data << endl;
};
```

Server Code

```
#include <iostream> #include "echo.hh"
#include "echoSK.cc" #include "echo_i.cc"
using namespace std;
int main(int argc, char *argv[]) {
  CORBA::ORB_var orb =
    CORBA::ORB_init(argc,argv,"omniORB4");
  CORBA::Object_var obj =
    orb->resolv_initial_references("RootPOA");
  PortableServer::POA_var poa =
    PortableServer::POA::_narrow(obj);
 Echo_i *myobj = new Echo_i();
 PortableServer::ObjectId_var myobjid =
   poa->activate_object(myobj);
 Echo_var myobjRef = myobj->_this();
 CORBA::String_var
   sior(orb->object_to_string(myobjRef));
 cout << (char*)sior << endl;
 myobj->_remove_ref();
 PortableServer::POAManager_var pman =
   poa->the_POAManager();
 pman->activate();
 orb->run();
 orb->destroy();
 return 0;
}
```

Comments on Server

Includes, `hh` is essential, others make for a single compile.

Setup ORB and POA.

Create a `Echo_i`, the real implementation.

Activate the real implementation.

Build a string that represents the server object
(encodes server IP and port numbers)
print string.
(The string makes `do_echo` available to external callers.)

Count down (lower the reference count)

Activate the POA.

Enter "run" mode.

On exit of run mode, free all memory and objects

Client Side

```
#include <iostream>
#include "echo.hh" #include "echoSK.cc"
#include "do_echo.cc"
using namespace std;
int main (int argc, char **argv) {
 CORBA::ORB_ptr orb =
  CORBA::ORB_init(argc,argv,"omniORB4");
 try {
  CORBA::Object_var obj =          //(1)
     orb->string_to_object(argv[1]);
  Echo_ptr Echo_Obj = Echo::_narrow(obj);
  if (CORBA::is_nil(Echo_Obj)) return -1;
  do_echo(Echo_Obj);
 }
 catch(...) {...}
 orb->destroy();
 return 0;
}
```

Usage: `a.out <object reference>`
Type the string representation of the object reference on the command line
1) Use the string representation to set up access to the object.
Call `do_echo` with the object proxy/handle

Vector Example

Regular program (split).

The main class.

```
// vector_services.cc
class Vector_Ops {
 public:
   struct Vector {
     long x,y;
   };
   Vector_Ops() {};
   virtual ~Vector_Ops() {};
   virtual struct Vector Add(
         const struct Vector &a,
         const struct Vector &b) {
     struct Vector Answer;
     Answer.x = a.x + b.x;
     Answer.y = a.y + b.y;
     return Answer;
   };
};
```

The principal function that uses the class.
Note: this must be passed a class instantiation when called.

```
// do_vectors.cc
#include <iostream>
using namespace std;
typedef Vector_Ops* Vector_ptr;
int do_vectors(Vector_ptr Vector_Obj) {
  struct Vector_Ops::Vector p;
  struct Vector_Ops::Vector q;
  struct Vector_Ops::Vector r;
  p.x = 1;
  p.y = 4;
  q.x = 2;
  q.y = 5;
  r = Vector_Obj->Add(p, q);
  cout << "answer (should be (3, 9)) " <<  r.x
  cout << "  " <<  r.y << endl;
  return 0;
}
```

Main program instantiates a vector object and calls the principal function.

```
int main() {
  Vector_ptr Vector_Instance = new Vector_Ops;
  return do_vectors(Vector_Instance);
}
```

The idl file

```
// File: vector_handler.idl
interface Vector_Ops {
   struct Vector {
     long x,y;
   };
   Vector Add( in Vector a, in Vector b);
};
```

The key word in has been added.
Class has been switched to interface.
The structure prefixes have been removed.
The private area has been removed.

The Implementation Class

```
//File: vector_ops_i.cc
#include "vector_handler.hh"
class Vector_Ops_i : public POA_Vector_Ops,
       public PortableServer::RefCountServantBase {
 public:
//   struct Vector {
//     int x,y;
//   };
   Vector_Ops_i() {};
   virtual ~Vector_Ops_i() {};
   virtual struct Vector_Ops::Vector Add(
       const struct Vector_Ops::Vector &a,
       const struct Vector_Ops::Vector &b) {
     struct Vector_Ops::Vector Answer;
     Answer.x = a.x + b.x;
     Answer.y = a.y + b.y;
     return Answer;
   };
};
```

Must inherit from the corba classes.
Change of name to `_i`
Removal of the structure declarations because the idl compiler had placed them in the skeleton class.

Parameter Conformance

New versions of the C++ compiler require exact conformance for parameters
(but not for other things like assignment).

Check the `.hh` file for the exact parameter types you should use.

Many parameter types are "Corbaized" by the idl compiler.

Although they are `typedef`'d to the original types (so assignments still work), your implementation will need to use the forms found in the `.hh` file for parameters.

For example:

an `unsigned long` specified in the `.idl` file gets translated

to a `::CORBA::ULong` in the `.hh` file.
You must change your parameters in the `_i` to match the `.hh` file or you will get compiler errors.

The user interface

```
// do_vectors.cc
#include <iostream>
#include "vector_handler.hh"
using namespace std;
int do_vectors(Vector_Ops_ptr Vector_Obj) {
 struct Vector_Ops::Vector p;
 struct Vector_Ops::Vector q;
 struct Vector_Ops::Vector r;
 p.x = 1;
 p.y = 4;
 q.x = 2;
 q.y = 5;
 r = Vector_Obj->Add(p, q);
 cout << "answer (should be (3, 9)) " << r.x
 cout << "  " << r.y << endl;
 return 0;
}
```

The parameter type has been changed.
The `Vector_Obj` has change type to `..._ptr`
The name of the class must preceed the name of the structure. (The structure is defined within the class.)

Separate Server

```
#include <fstream.h> using namespace std;
#include "vector_handler.hh"
#include "vector_handlerSK.cc"
#include "vector_ops_i.cc"
int main(int argc, char *argv[]) {
 CORBA::ORB_var orb =
  CORBA::ORB_init(argc,argv,"omniORB4");
 CORBA::Object_var obj =
  orb->resolve_initial_references("RootPOA");
 PortableServer::POA_var poa =
  PortableServer::POA::_narrow(obj);
 Vector_Ops_i *myobj = new Vector_Ops_i();
 PortableServer::ObjectId_var myobjid =
  poa->activate_object(myobj);
 obj = myobj->_this();
 CORBA::String_var
  sior(orb->object_to_string(obj));
 ofstream F("mysior.txt");
 F << (char*)sior << endl;
 F.close();
 myobj->_remove_ref();
 PortableServer::POAManager_var pman =
  poa->the_POAManager();
 pman->activate(); orb->run();
 orb->destroy(); return 0;
}
```

Separate Server Comments

Similar to the echo example,
except it drops sior into file.

Entirely boilerplate,

the mentions of `Vector_Ops` occur in exactly the same places as `Echo` in the previous example.

This could be done as text replacement by a reasonable idl compiler.

rpcgen has the advantage here because it does this for the server side main.

Separate Client

The sior string is read in from the file.
The orb initialization is done.
The stringified reference is used to contact the server

```
#include "vector_handler.hh"
#include "vector_handlerSK.cc"
#include "do_vectors.cc"
#include <fstream.h>
using namespace std;
int main(int argc, char *argv[]) {
 // Get the sior string from the file.
 char siorbuf[2048];
 ifstream F("mysior.txt");
 F >> siorbuf ;
 F.close();
 // The object request broker is initialized.
 CORBA::ORB_ptr orb =
   CORBA::ORB_init(argc,argv,"omniORB4");
 CORBA:Object_var obj =
  orb->string_to_object(siorbuf);
```

We do the narrow to a vector object, getting a proxy pointer.
We call the client interface procedure passing the proxy

```
 Vector_Ops_var Vector_Obj =
   Vector_Ops::_narrow(obj);
 if (CORBA::is_nil(Vector_Obj) {
   cerr << "Object request failed\n";
   return -1;
 }
 do_vectors(Vector_Obj);
 orb->destroy(); //cleanup
 return 0;
}
```

Except for the `do_vectors` call this could be generated.
Everything else is “text replacement”