DCOM and CORBA Side by Side, Step by Step, and Layer by Layer

P. Emerald Chung **Yennun Huang** **Shalini Yajnik**
Bell Laboratories, Lucent Technologies
Murray Hill, New Jersey

Deron Liang **Joanne C. Shih** **Chung-Yih Wang**
Institute of Information Science
Academia Sinica
Republic of China, Taiwan

Yi-Min Wang
AT&T Labs, Research
Florham Park, New Jersey

Abstract

DCOM (Distributed Component Object Model) *and* ***CORBA (Common Object Request Broker Architecture)*** *are two popular distributed object models. In this paper, we make architectural comparison of DCOM and CORBA at three different layers: basic programming architecture, remoting architecture, and the wire protocol architecture. A step-by-step description of remote object activation and method invocation is provided to demonstrate the similarities and differences of the two frameworks. A primary goal is for people who are already familiar with one model to quickly understand the basic architecture of the other.*

---

1. Introduction

The explosive growth of the Web, the increasing popularity of PCs and the advances in high-speed network access have brought distributed computing into the main stream. To simplify network programming and to realize component-based software architecture, two distributed object models have emerged as standards, namely, DCOM (Distributed Component Object Model) and CORBA (Common Object Request Broker Architecture).

DCOM is the distributed extension to **COM (Component Object Model)** [COM 95] that builds an object remote procedure call (ORPC) layer on top of DCE RPC [DCE 95] to support remote objects. A *COM server* can create object instances of multiple *object classes*. A COM object can support multiple **interfaces**, each representing a different view or behavior of the object. An interface consists of a set of functionally related methods. A COM client interacts with a COM object by

acquiring a pointer to one of the object's interfaces and invoking methods through that pointer, as if the object resides in the client's address space. COM specifies that any interface must follow a standard memory layout, which is the same as the C++ virtual function table [Rogerson 96]. Since the specification is at the binary level, it allows integration of binary components possibly written in different programming languages such as C++, Java and Visual Basic.

CORBA is a distributed object framework proposed by a consortium of 700+ companies called the Object Management Group (OMG) [CORBA 95]. The core of the CORBA architecture is the **Object Request Broker (ORB)** that acts as the object bus over which objects transparently interact with other objects located locally or remotely [Vinoski 97]. A CORBA object is represented to the outside world by an interface with a set of methods. A particular instance of an object is identified by an object reference. The client of a CORBA object acquires its object reference and uses it as a handle to make method calls, as if the object is located in the client's address space. The ORB is responsible for all the mechanisms required to find the object's implementation, prepare it to receive the request, communicate the request to it, and carry the reply (if any) back to the client. The object implementation interacts with the ORB through either an *Object Adapter (OA)* or through the ORB interface.

The following terminology will be used to refer to the entities in both frameworks.

Interface
A named collection of abstract *operations* (or *methods*) that represent one functionality.
Object class (or class)
A named concrete implementation of one or more interfaces.
Object (or object instance)
An instantiation of some object class.
Object server
A process responsible for creating and hosting object instances.
Client
A process that invokes a method of an object.

Both DCOM and CORBA frameworks provide client-server type of communications. To request a service, a client invokes a method implemented by a remote object, which acts as the server in the client-server model. The service provided by the server is encapsulated as an object and the interface of an object is described in an **Interface Definition Language (IDL)**. The interfaces defined in an IDL file serve as a *contract* between a server and its clients. Clients interact with a server by invoking methods described in the IDL. The actual object implementation is hidden from the client. Some object-oriented programming features are present at the IDL level, such as data encapsulation, polymorphism and single inheritance. CORBA also supports multiple inheritance at the IDL level, but DCOM does not. Instead, the notion of an object having multiple interfaces is used to achieve a similar purpose in DCOM. CORBA IDL can also specify exceptions.

In both DCOM and CORBA, the interactions between a client process and an object server are implemented as object-oriented RPC-style communications [Birrell 84]. Figure 1 shows a typical RPC structure. To invoke a remote function, the client makes a call to the *client stub*. The stub packs the call parameters into a request message, and invokes a wire protocol to ship the message to the server. At the server side, the wire protocol delivers the message to the *server stub*, which then unpacks the request message and calls the actual function on the object. In DCOM, the client stub is referred to as the **proxy** and the server stub is referred to as the **stub**. In contrast, the client stub in CORBA is called the **stub** and the server stub is called the **skeleton**. Sometimes, the term "proxy" is also used to refer to a running instance of the stub in CORBA.

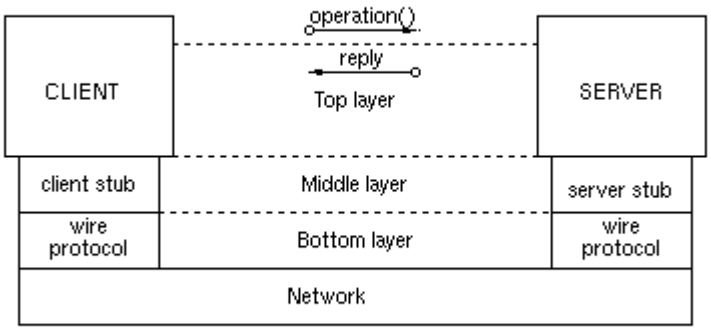

Figure 1: RPC structure

The overall architectures of DCOM and CORBA are illustrated in Figure 2 and Figure 3, respectively. In the following sections, we describe a single example implemented in both DCOM and CORBA, and provide a step-by-step description of object activations and method invocations at the three different layers shown in the figures. The top layer is the **basic programming architecture**, which is visible to the developers of the client and object server programs. The middle layer is the **remoting architecture**, which transparently makes the interface pointers or object references meaningful across different processes. The bottom layer is the **wire protocol architecture**, which further extends the remoting architecture to work across different machines.

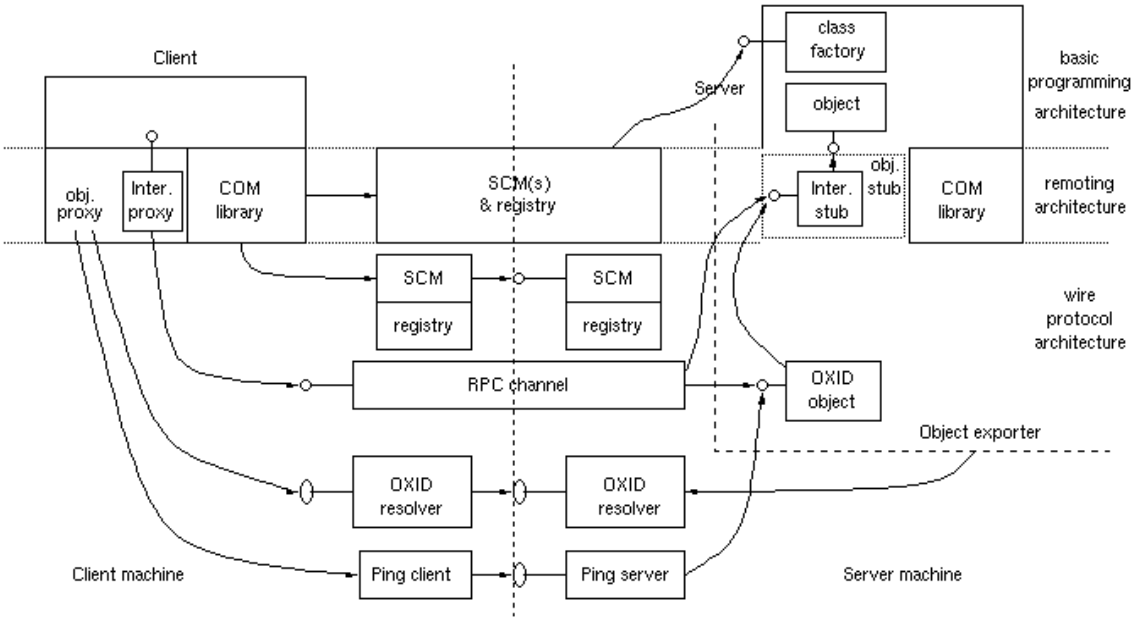

Figure 2: DCOM overall architecture.

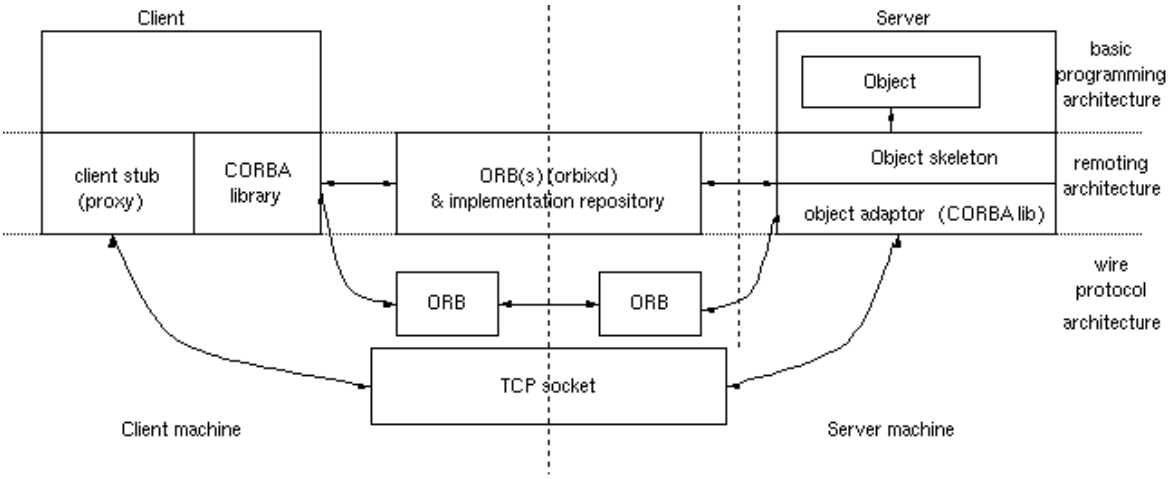

Figure 3: CORBA overall architecture.

Throughout this paper, the description about DCOM is based on the COM specification [COM 95] and the DCOM specification [Brown 96]. The CORBA description is based on the CORBA specification [CORBA 95] whenever possible. For information not specified by CORBA, we use Iona Orbix [Iona 95] implementation to complete the description.

2. Sample Application

We use an example called `Grid` throughout this paper. The `Grid` server object maintains a two-dimensional grid of integers and supports two groups of methods. The first group consists of two methods: `get()` and `set()`, which are invoked to get and set the value at a particular grid point, respectively. The second group has only one method: `reset()`, which sets the value at every grid point to the supplied value. As a simple demonstration, the `Grid` client first invokes the `get()` method to obtain the value at coordinate (0,0), increases the value by one, and then calls `reset()` to set the entire grid to the new value.

We design the DCOM and CORBA implementations in different ways to demonstrate that DCOM supports **objects with multiple interfaces**, while CORBA allows **an interface to inherit from multiple interfaces**. Note that DCOM and CORBA are basically oblivious to the inheritance relationship between the C++ implementation classes.

In CORBA, we define three interfaces: (1) interface `grid1` supports `get()` and `set()`; (2) interface `grid2` supports `reset()`; (3) interface `grid` multiply inherits from `grid1` and `grid2`. In contrast, we define two interfaces in DCOM, `IGrid1` and `IGrid2`, for the two groups of methods. The implementation of the `Grid` object uses multiple inheritance from `IGrid1` and `IGrid2` to implement an object with the two interfaces. Note that we could have merged all three methods into one interface by using interface single inheritance, which then looks very similar to its CORBA counterpart. But DCOM's support for objects with multiple interfaces allows each distinct feature of an object to have a separate interface.

For each implementation, we list the source code from five files. To simplify presentation, only essential code is shown. The first file, shown in Table 1, is the IDL file that defines the interfaces and its methods. The DCOM IDL file also associates multiple interfaces with an object class, as shown in the `coclass` block. Running the IDL file through an IDL compiler in both DCOM and CORBA generates the proxy/stub/skeleton code and the interface header file (`grid.h` or `grid.hh`) that are used by both the server and the client. Note that, in DCOM, each interface is assigned a **globally unique identifier (GUID)** called the **interface ID** (**`IID`**). Similarly, each object class is assigned a unique **class ID** (**`CLSID`**). Also, every COM interface must inherit from the **`IUnknown`** interface that consists of a **`QueryInterface()`** method for navigating between different interfaces of the same object, and two other methods **`AddRef()`** and **`Release()`** for reference counting. Reference counting provides a lifetime control mechanism that allows a COM object to keep track of its clients and can delete itself when it is no longer needed.

The second file shown in Table 2 is the implementation header file that shows how the server implementation class is derived from the interfaces. The DCOM file includes the definition of a class factory `CClassFactory`, which is commonly used but not required. As mentioned previously, the implementation class `CGrid` multiply inherits from the two pure abstract base classes `IGrid1` and `IGrid2` which are

defined in the IDL-generated interface header file `grid.h` (not shown). In the `CGrid` class, `AddRef()` increments the reference count and `Release()` decrements it. When the reference count drops to zero, the server object deletes itself. Again, this is commonly used but not required. Ultimately, it is the server object itself which controls its own life time.

In the CORBA implementation, the IDL compiler generates from the interface definition the interface class `grid` in the header file `grid.hh`(not shown). The application developer writes the implementation class `grid_i`. There are two ways of associating the implementation class with the interface class - the inheritance approach and the delegation approach. In this example, we chose the inheritance approach. In this approach, the IDL compiler in Orbix also generates a class called `gridBOAImpl` that is responsible for instantiating the skeleton class. Class `gridBOAImpl` inherits from the interface class `grid`, which inherits from class `CORBA::Object`. The implementation class `grid_i` inherits from class `gridBOAImpl` to complete the mapping between the interface class and the implementation class. Note that the type `gridBOAImpl` is Orbix specific, since current CORBA do not specify what the skeleton class looks like and what the name of the base class is. This makes the server code not portable to other ORB products. To resolve this issue, *Portable Object Adaptor (POA)* was recently introduced [POA 97]. POA corrects this problem and specifies the name for the base class. In this example, when POA becomes available, the class `grid_i` would inherit from a base class called `POA_grid`. More descriptions of POA are given in Section 4.

The third file shown in Table 3 implements the methods of the server class. The DCOM file also implements some methods of the class factory. The fourth file shown in Table 4 is the main program for the server. The DCOM program creates an event and waits on that event which is signaled when all active server objects are deleted and so the server can exit. The actual client requests are handled concurrently by different threads from a thread pool. (Another DCOM threading model handles requests serially using one thread.)

Similarly, the CORBA server program instantiates an instance of class `grid_i` and then blocks at `impl_is_ready()` to receive the incoming client requests. If the server does not receive any requests during a default timeout period (which can be set by the programmer), it gracefully closes down. The client requests are handled either serially or by different threads, depending on the activation policy used for the object server. The last file shown in Table 5 is the client code. The readers may observe that DCOM client code tends to be longer than CORBA client code due to the additional `IUnknown` method calls. This may not be true for DCOM clients written in Java or Visual Basic, where the virtual machine layer takes care of the `IUnknown` method calls and hides them from the programmers [Chappell 97]. Even in a C++ client, smart interface pointers can be used to hide the reference counting [Rogerson 96].

After compiling and before executing the programs, both DCOM and CORBA require a registration process for the server. In CORBA, the association between the interface name and the path name of the server executable is registered with the

implementation repository. In DCOM, the association between the CLSID and the path name of the server executable is registered with the **registry**. In addition, since a DCOM interface proxy/stub is itself a COM object, its associated server (in the dynamic link library (DLL) form) also needs to be similarly registered.

Due to space limitation, we do not cover *dynamic invocation*, which does not require static type information at compile time. In DCOM, type information for interface methods is stored in a **type library** generated by the IDL compiler and assigned a GUID. It can be used through the `IDispatch` interface to perform dynamic invocation [Rogerson 96]. It can also be used for *type library-driven marshaling* [Grimes 97]: instead of using a separate proxy/stub DLL that contains information specific to an interface, a generic marshaler can perform marshaling by reading type library information. In CORBA, the IDL compiler generates the type information for each method in an interface and stores it in the **Interface Repository** (IR). A client can query the interface repository to get run-time information about a particular interface and then use that information to create and invoke a method on the object dynamically through the **dynamic invocation interface** (DII). Similarly, on the server side, the **dynamic skeleton interface** (DSI) allows a client to invoke an operation on an object that has no compile time knowledge of the type of object it is implementing [CORBA 95].

DCOM IDL

```
// uuid and definition of IGrid1
    [  object,
       uuid(3CFDB283-CCC5-11D0-BA0B-
00A0C90DF8BC),
       helpstring("IGrid1 Interface"),
       pointer_default(unique)
    ]
    interface IGrid1 : IUnknown      {
       import "unknwn.idl";
       HRESULT get([in] SHORT n,
                   [in] SHORT m,
                   [out] LONG *value);
       HRESULT set([in] SHORT n,
                   [in] SHORT m,
                   [in] LONG value);
    };

// uuid and definition of IGrid2
    [     object,
          uuid(3CFDB284-CCC5-11D0-BA0B-
00A0C90DF8BC),
        helpstring("IGrid2 Interface"),
        pointer_default(unique)
    ]
    interface IGrid2 : IUnknown      {
        import "unknwn.idl";
        HRESULT reset([in] LONG value);
    };

// uuid and definition of type library
[   uuid(3CFDB281-CCC5-11D0-BA0B-00A0C90DF8BC),
    version(1.0),
    helpstring("grid 1.0 Type Library)
]
library GRIDLib
{
    importlib("stdole32.tlb");
    // uuid and definition of class
    [     uuid(3CFDB287-CCC5-11D0-BA0B-
00A0C90DF8BC),
        helpstring("Grid Class")
    ]
    // multiple interfaces
    coclass CGrid
    {     [default] interface IGrid1;
                    interface IGrid2;
    };
};
```

CORBA IDL

```
interface grid1
{
        long get(in short n,
                 in short m);
        void set(in short n,
                 in short m,
                 in long value);
};

interface grid2
{
        void reset(in long value);
};

// multiple inheritance of interfaces
interface grid: grid1, grid2
{
};
```

Table 1: The IDL files.

DCOM server class definition (`cgrid.h`)

```
#include "grid.h" // IDL-generated interface header
file

class CClassFactory : public IClassFactory {
 public:
  // IUnknown
  STDMETHODIMP QueryInterface(REFIID riid, void** ppv);
  STDMETHODIMP_(ULONG) AddRef(void) {
    return 1;
  };
  STDMETHODIMP_(ULONG) Release(void) {
    return 1;
  }

  // IClassFactory
  STDMETHODIMP CreateInstance(LPUNKNOWN punkOuter,
REFIID iid, void **ppv);
  STDMETHODIMP LockServer(BOOL fLock) {
    return E_FAIL;
  };
};

class CGrid : public IGrid1, public IGrid2 {
 public:
  // IUnknown
  STDMETHODIMP QueryInterface(REFIID riid, void** ppv);
  STDMETHODIMP_(ULONG) AddRef(void) {
    return InterlockedIncrement(&m_cRef);
  }
  STDMETHODIMP_(ULONG) Release(void) {
    if (InterlockedDecrement(&m_cRef) == 0) {
      delete this; return 0;
    }
    return 1;
  }
  // IGrid1
  STDMETHODIMP get(IN SHORT n, IN SHORT m,
                   OUT LONG *value);
  STDMETHODIMP set(IN SHORT n, IN SHORT m,
                   IN LONG value);
  // IGrid2
  STDMETHODIMP reset(IN LONG value);

  CGrid(SHORT h, SHORT w);
  ~CGrid();
 private:
  LONG  m_cRef, **m_a;
  SHORT m_height, m_width;
};
```

CORBA server class definition (`grid_i.h`)

```
#include "grid.hh" // IDL-generated
interface header file

class grid_i : public gridBOAImpl {
public:

  virtual CORBA::Long get(CORBA::Short n,
CORBA::Short m,
                CORBA::Environment &env);
  virtual void set(CORBA::Short n,
CORBA::Short m,
                CORBA::Long value,
CORBA::Environment &env);

  virtual void reset(CORBA::Long value,

CORBA::Environment &env);
  grid_i(CORBA::Short h, CORBA::Short w);
  virtual ~grid_i();
private:
  CORBA::Long **m_a;
  CORBA::Short m_height, m_width;
};
```

Table 2: The server implementation header files.

DCOM server implementation

```
#include "cgrid.h"

STDMETHODIMP
CClassFactory::QueryInterface(REFIID riid,
void** ppv) {
   if (riid == IID_IClassFactory || riid ==
IID_IUnknown) {
     *ppv = (IClassFactory *) this;
     AddRef(); return S_OK;
   }
   *ppv = NULL;
   return E_NOINTERFACE;
}

STDMETHODIMP
CClassFactory::CreateInstance(LPUNKNOWN p,
REFIID riid, void** ppv) {
   IGrid1* punk = (IGrid1*) new CGrid(100, 100);
   HRESULT hr = punk->QueryInterface(riid, ppv);
   punk->Release();
   return hr;
}

STDMETHODIMP CGrid::QueryInterface(REFIID riid,
void** ppv) {
   if (riid == IID_IUnknown || riid ==
IID_IGrid1)
     *ppv = (IGrid1*) this;
   else if (riid == IID_IGrid2) *ppv = (IGrid2*)
this;
   else { *ppv = NULL; return E_NOINTERFACE; }
   AddRef();
   return S_OK;
}

STDMETHODIMP CGrid::get(IN SHORT n, IN SHORT m,
OUT LONG* value) {
    *value = m_a[n][m];
    return S_OK;
}
STDMETHODIMP CGrid::set(IN SHORT n, IN SHORT m,
IN LONG value) {
    m_a[n][m] = value;
    return S_OK;
}
STDMETHODIMP CGrid::reset(IN LONG value) {
    SHORT n, m;
    for (n=0; n < m_height; n++)
      for (m=0; m < m_width; m++)
        m_a[n][m] = value;
    return S_OK;
}


CGrid::CGrid(SHORT h, SHORT w) {
  m_height = h;
  m_width= w;
  m_a = new LONG*[m_height];
  for (int i=0; i < m_height; i++)
    m_a[i] = new LONG[m_width];
  m_cRef = 1;
}
extern HANDLE hevtDone;

CGrid::~CGrid () {
  for (int i=0; i < m_height; i++)
    delete[] m_a[i];
  delete[] m_a;
  SetEvent(hevtDone);
}
```

CORBA server implementation

```
#include "grid_i.h"

CORBA::Long grid_i::get(CORBA::Short n,
CORBA::Short m, CORBA::Environment &) {
    return m_a[n][m];
}

void grid_i::set(CORBA::Short n, CORBA::Short m,
    CORBA::Long value, CORBA::Environment &) {
    m_a[n][m] = value;
}

void grid_i::reset(CORBA::Long value,
CORBA::Environment &) {
    short n, m;
    for (n = 0; n < m_height; n++)
      for (m = 0; m < m_width; m++)
        m_a[n][m]=value;
    return;
}

grid_i::grid_i(CORBA::Short h, CORBA::Short w) {
    m_height=h;   // set up height
    m_width=w;    // set up width
    m_a = new CORBA::Long* [h];
    for (int i = 0; i < h; i++ )
      m_a[i] = new CORBA::Long[w];
}



grid_i::~grid_i () {
    for (int i = 0; i < m_height; i++)
      delete[] m_a[i];
    delete[] m_a;
}
```

Table 3: The server implementation files.

DCOM server main program

```
HANDLE hevtDone;

void main()
{
    // Event used to signal this main thread
    hevtDone = CreateEvent(NULL, FALSE, FALSE, NULL);
    hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);
    CClassFactory* pcf = new CClassFactory;
    hr = CoRegisterClassObject(CLSID_CGrid, pcf,
          CLSCTX_SERVER,  REGCLS_MULTIPLEUSE ,
&dwRegister);
    // Wait until the event is set by CGrid::~CGrid()
    WaitForSingleObject(hevtDone, INFINITE);
    CloseHandle(hevtDone);
    CoUninitialize();
}
```

CORBA server main program

```
int main()
{
  // create a grid object using the
implementation class grid_i

  grid_i ourGrid(100,100);

  try {
    // tell Orbix that we have completed the
server's initialization:
    CORBA::Orbix.impl_is_ready("grid");
  } catch (...) {
    cout << "Unexpected exception" << endl;
    exit(1);
  }
}
```

Table 4: The server main programs.

DCOM Client code

```
#include "grid.h"

void main(int argc, char**argv)
{
    IGrid1          *pIGrid1;
    IGrid2          *pIGrid2;
    LONG  value;


    CoInitialize(NULL);           // initialize COM
    CoCreateInstance(CLSID_CGrid, NULL, CLSCTX_SERVER,
                     IID_IGrid1, (void**) &pIGrid1);
    pIGrid1->get(0, 0, &value);
    pIGrid1->QueryInterface(IID_IGrid2, (void**)
&pIGrid2);
    pIGrid1->Release();
    pIGrid2->reset(value+1);
    pIGrid2->Release();
    CoUninitialize();
}
```

CORBA Client code

```
#include "grid.hh"

void main (int argc, char **argv)
{
  grid_var gridVar;

  CORBA::Long value;

  // bind to "grid" object; Orbix-
specific
  gridVar = grid::_bind(":grid");

  value = gridVar->get(0, 0);



  gridVar->reset(value+1);
}
```

Table 5: The client main programs.

3. Top Layer: Basic Programming Architecture

At the top layer, we show the programmers' view of DCOM and CORBA. More specifically, we describe how a client requests an object and invokes its methods, and how a server creates an object instance and makes it available to the client. Exactly how the client is connected to the server is totally hidden from the programmers. The client and the server programs interact as if they reside in the same address space on the same machine. The main differences between DCOM and CORBA at this layer include *how a client specifies an interface* and *COM's class factories and the* `IUnknown` *methods*. A step-by-step description is given in Table 6 and illustrated in Figure 4 and Figure 5 for DCOM and CORBA, respectively. **(Numbers in parenthesis are for object activation steps; those in square brackets are for method invocation steps.)**

Although Table 6 gives a common DCOM invocation sequence, we would like to point out two things. First, the use of class factories in COM is optional. A server object can actually call `CoRegisterClassObject()` to register any interface pointer, and clients can invoke another COM API named `CoGetClassObject()` to retrieve that pointer. (A *class object* is a named singleton object that acts as the metaclass for a COM object class.) Second, `CoCreateInstance()` does not necessarily create a fresh instance. Inside `IClassFactory::CreateInstance()`, a server can choose to always return the same interface pointer so that different clients can connect to the same object instance with a particular state. Another way of binding to a named server object instance is to use *monikers* [Box2 97] and/or the *Running Object Table (ROT)* [COM 95].

In CORBA, an object can be activated by invoking any method on an existing object reference. Some vendors provide special method calls, e.g. `_bind()` operation in Orbix, to activate a server object and obtain its object reference. The client may attach to an existing instance instead of a new instance, if there is any existing instance matching the requested type. Note that a client can store an object reference by stringifying it using `object_to_string()` and can later use it again by converting it back by `string_to_object()`.

Another difference to note between DCOM and CORBA at the programming layer is the way they perform exception handling. CORBA provides support for standard C++ exceptions and some CORBA specific exceptions. In addition, user defined exceptions are also allowed and are declared in the IDL. The IDL compiler maps a user defined exception to a C++ class.

In contrast, DCOM requires that all methods return a 32-bit error code called an *HRESULT* (see Table 3) at this layer. At the language/tool level, a set of conventions and system provided services (called the `IErrorInfo` object) allows failure HRESULTs to be converted into exceptions in a way natural to the language. For example, in Microsoft Visual C++ 5.0, client programmers can use standard C++ try/catch blocks to catch errors from COM method invocations; the compiler generates the correct code to map the failure HRESULT into a correct usage of

`IErrorInfo`, effectively translating the failure return code into an exception. Similarly, tools can allow programmers to "throw exceptions" instead of returning failure codes. The DCOM wire protocol includes a mechanism known as *body extensions* [Brown 96] that allow rich exception information (such as a string explaining the error) to be carried.

DCOM	CORBA
Object activation	
1. Client calls COM library's CoCreateInstance() with CLSID_Grid and IID_IGrid1.	1. Client calls client stub's grid::_bind(), which is a static function in the stub.
2. COM infrastructure starts an object server for CLSID_Grid.	2. ORB starts a server that contains an object supporting the interface grid.
3. As shown in the server main program, server creates class factories for all supported CLSIDs, and calls CoRegisterClassObject() to register each factory. Server blocks on waiting for, for example, an event to be set to signal that the server is no longer needed. Incoming client requests will be served by other threads.	3. As shown in the server main program, Server instantiates all supported objects. (In each constructor, calls are made to create and register an object reference.) Server calls CORBA::BOA::impl_is_ready() to tell ORB that it is ready to accept client requests.
4. COM obtains the IclassFactory pointer to the CLSID_Grid factory, and invokes CreateInstance() on it.	
5. In CreateInstance(), server creates an object instance and makes a QueryInterface() call to obtain an interface pointer to the IID_IGrid1 interface.	
6. COM returns the interface pointer as pIGrid1 to the client.	4. ORB returns the object reference for grid as gridVar to the client.

Method invocation	
1. Client calls pIGrid1->get() which eventually invokes CGrid::get() in the server. 2. To obtain a pointer to another interface IID_IGrid2 of the same object instance, client calls pIGrid1->QueryInterface() which invokes Grid::QueryInterface. 3. When finishing using pIGrid1, client calls pIGrid1->Release() (which may not invoke CGrid::Release() [footnote 1]). 4. Client calls pIGrid2->reset() which invokes CGrid::reset. 5. Client calls pIGrid2->Release() which invokes CGrid::Release().	1. Client calls gridVar->get() which eventually invokes grid_i::get() in the server. 2. Client calls gridVar->reset() which invokes grid_i::reset().

Table 6: The top layer description.

Footnote 1: For performance reason, `Release()` calls for individual interfaces may not be actually forwarded to the server side until all interface pointers that a client holds to the same object are all released. This allows caching interface pointers that may be requested again by the client, and allows lower layers to bundle multiple `Release()` calls in a single remote call.

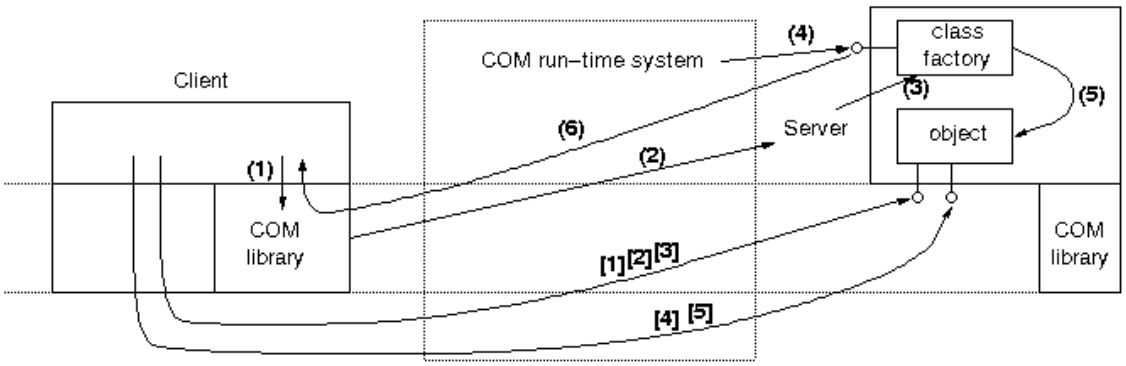

Figure 4: DCOM steps at the top layer.

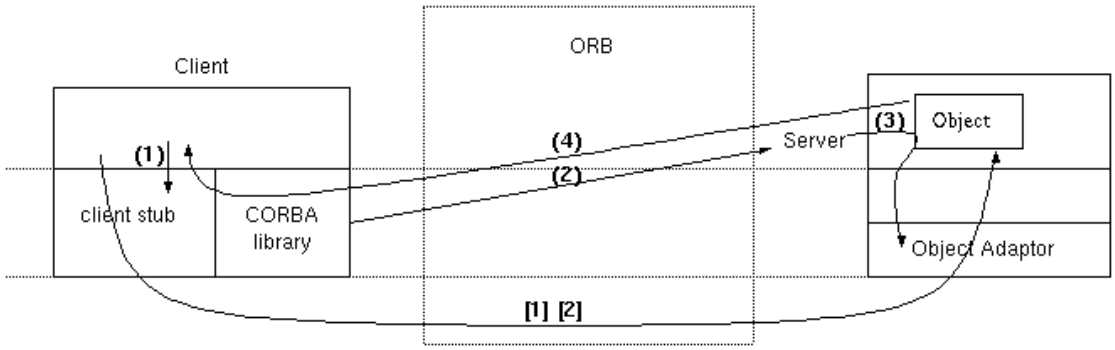

Figure 5: CORBA steps at the top layer.

4. Middle Layer: Remoting Architecture

The middle layer consists of the infrastructure necessary for providing the client and the server with the illusion that they are in the same address space. The description in Table 7 shows how the infrastructure locates and starts the requested server, and the entities involved when a method invocation takes place across different processes. The corresponding illustrations for DCOM and CORBA are shown in Figure 6 and Figure 7, respectively. The main differences between DCOM and CORBA at this layer include *how server objects are registered* and *when proxy/stub/skeleton instances are created*.

To send data across different address spaces requires a process called marshaling and unmarshaling. *Marshaling* packs a method call's parameters (at a client's space) or return values (at a server's space) into a standard format for transmission. *Unmarshaling*, the reverse operation, unpacks the standard format to an appropriate

data presentation in the address space of a receiving process. Note that the marshaling process described in this section is called *standard marshaling* in DCOM terminology. DCOM also provides a *custom marshaling* mechanism to bypass the standard marshaling procedure [Brockschmidt 93] [COM 95] [Box1 97]. By implementing an `IMarshal` interface, a server object declares that it wants to control how and what data are marshaled and unmarshaled, and how the client should communicate with the server. In effect, custom marshaling provides an **extensible architecture** for plugging in application-specific communication infrastructure. It can be useful for client-side data caching, for fault tolerance, etc.

We describe here some of the additional CORBA terms used in Table 7. As stated in the Introduction, the ORB acts as the object bus. The Object Adaptor (OA) sits on top of the ORB, and is responsible for connecting the object implementation to the ORB. Object Adaptors provide services like generation and interpretation of object references, method invocation, object activation and deactivation, mapping object references to implementations. Different object implementation styles have different requirements and need to be supported by different object adapters, e.g. object-oriented database adapter for objects in a database. The Basic Object Adapter (BOA) defines an object adapter which can be used for most conventional object implementations. CORBA specifications do not mandate how the ORB/BOA functionality is to be implemented. Orbix built the ORB/BOA functionality into two libraries and a daemon process (`orbixd`). The daemon is responsible for location and activation of objects. The two libraries, a server-side library and a client-side library, are each linked at compile time with server and client implementations, respectively, to provide the rest of the functionality [Orbix 96].

It is important to note that the recently introduced POA will be a replacement for BOA. The POA specifications provide portability for CORBA server code and also introduce some new features in the Object Adapter. The POA specifications have not yet been incorporated by any ORB vendors into the products. Thus, our descriptions are based on the current products which implements the BOA specifications. However, wherever we discuss BOA specific details we will point out the approach taken by POA in that context.

DCOM	CORBA
Object activation	
1. Upon receiving CoCreateInstance() call, COM library delegates the task to Service Control Manager (SCM). 2. SCM checks if a class factory for CLSID_Grid has been registered; if not, SCM consults the registry to map CLSID_Grid to its server path name, and starts the server.	1. Upon receiving grid::_bind() call, client stub delegates the task to ORB [footnote 2]. 2. ORB consults the Implementation Repository to map grid to its server path name, and activates the server (in Orbix, the orbixd daemon forks the server process).

3. Server registers all supported class factories in a class object table. 4. SCM retrieves from the table the IClassFactory pointer to the CLSID_Grid factory, and invokes CreateInstance() on it.	3. Server instantiates all supported objects, including a grid object of class grid_i. Class grid_i indirectly inherits from CORBA::Object whose constructor calls BOA::create() with a unique reference ID to get back an object reference. It then registers the object reference with ORB by calling obj_is_ready() [Orfali 97].
5. When CreateInstance() returns the IID_IGrid1 pointer, COM (conceptually) creates an object stub for the newly created object instance. 6. The object stub marshals the interface pointer, consults the registry to create an interface stub for IID_IGrid1, and associates it with the server object's actual IID_IGrid1 interface.	4. The constructor for class grid_i also creates an instance of the skeleton class. [footnote 3].
7. When SCM ferries the marshaled pointer back to the client side, COM creates an object proxy for the object instance. 8. The object proxy unmarshals the pointer, consults the registry to create an interface proxy for IID_IGrid1, and associates it with the RPC channel object connected to the stub.	5. When the ORB ferries the object reference back to the client side, it creates an instance of the proxy class and registers it in the proxy object table with its corresponding object reference.
9. COM library returns to the client an IID_IGrid1 pointer to the interface proxy as pIGrid1.	6. Client stub returns to the client an object reference as gridVar.

Method Invocation:	
1. Upon receiving pIGrid1->get() call, interface proxy marshals necessary parameters, and invokes the SendReceive() method on the RPC channel object to send the request.	1. Upon receiving gridVar->get() call, the proxy creates a Request pseudo object, marshals the necessary parameters into it, and calls Request::invoke(), which calls CORBA::Request::send() to put the message in the channel, and waits onCORBA::Request::get_response() for reply.
2. The RPC channel sends the request to the server side, finds the target IID_IGrid1 interface stub, and calls the Invoke() method on it.	2. When the message arrives at the server, the BOA finds the target skeleton, rebuilds the Request object, and forwards it to the skeleton.
3. Interface stub unmarshals the parameters, invokes the method (identified by a method number) on the grid object, marshals the return values, and returns from the Invoke method.	3. The skeleton unmarshals the parameters from the Requestobject, invokes the method (identified by a method name) on the grid object, marshals the return values, and returns from the skeleton method. The ORB builds a reply message and places it in the transmit buffer.
4. When the RPC channel ferries the marshaled return values back to the client side, the interface proxy returns from the SendReceive() call, unmarshals the return values, and returns them to the client to finish the pIGrid1->set() call.	4. When the reply arrives at the client side, CORBA::Request::get_response() call returns after reading the reply message from the receive buffer. The proxy then unmarshals the return values, checks for exceptions, and returns them to the client to finish the gridVar->get() call.
5. Upon receiving pIGrid1->QueryInterface() call, interface proxy delegates the request to the object proxy's IUnknown interface.	
6. The object proxy remotely invokes the actual QueryInterface() call on the grid object through the same process explained above.	

7. Upon returning the new IID_IGrid2 interface pointer, COM creates the interface stub and proxy for it (which share the same object stub and proxy with the IID_IGrid1 interface stub and proxy, respectively). 8. The IID_IGrid1 interface proxy returns to the client an IID_IGrid2 pointer to the new interface proxy. 9. Upon receiving pIGrid1->Release() call, IID_IGrid1 interface proxy delegates the request to the object proxy. 10. Upon receiving pIGrid2->reset() call, IID_IGrid2 interface proxy makes the remote call as usual. 11. Upon receiving pIGrid2->Release() call, IID_IGrid2 interface proxy delegates the request to the object proxy which then makes a remote call to release pIGrid2 (and possibly pIGrid1).	5. Upon receiving gridVar->reset() call, the proxy follows a similar procedure.

Table 7: The middle layer description.

Footnote 2: The stub actually checks its *proxy object table* first to see if it already has an object reference for `grid`. The proxy object table maintains a run-time table of all valid object references on the client side.

Footnote 3: Steps 3 and 4 somewhat correspond to the *implicit activation* policy in POA. POA offers a number of policies related to object activation. Due to lack of space, we will not discuss them in this paper.

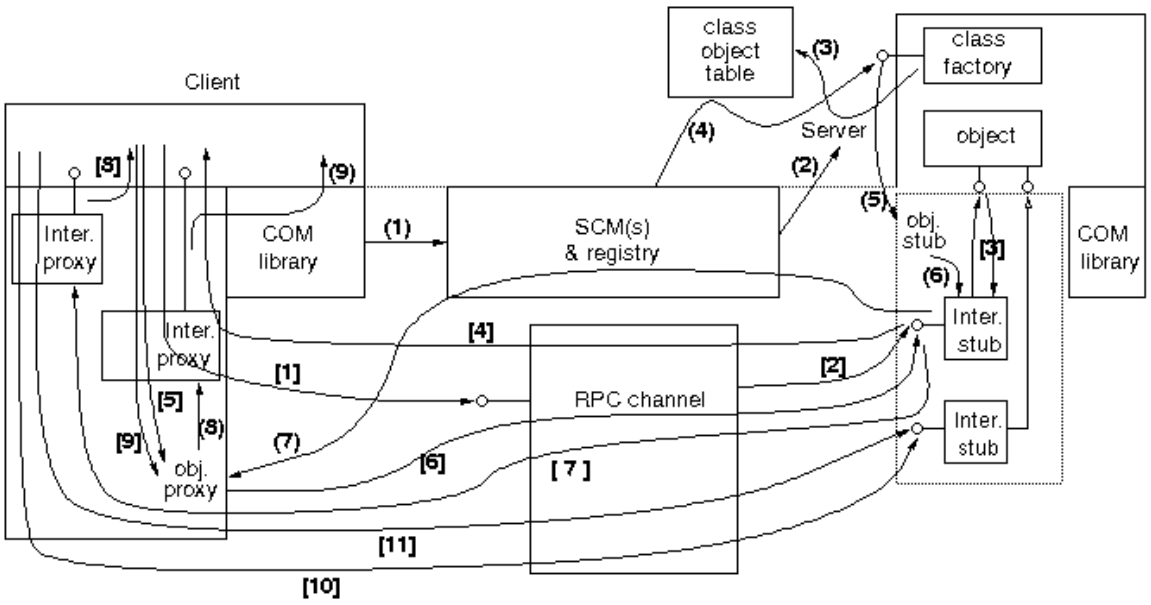

Figure 6 DCOM steps at the middle layer.

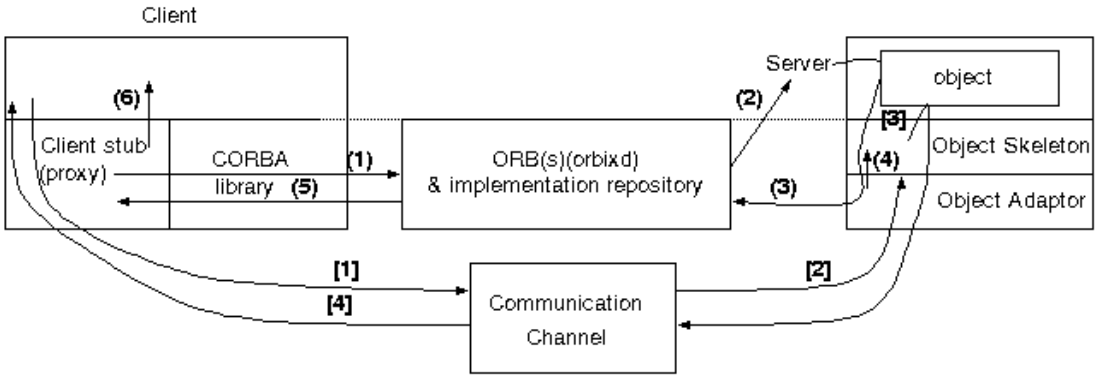

Figure 7 CORBA steps at the middle layer.

5. Bottom Layer: Wire Protocol Architecture

The bottom layer specifies the wire protocol for supporting the client and the server running on different machines. The description in Table 8 shows how objects on a remote machine are created and describes the entities involved when a method invocation is carried out across machines. Figure 8 and Figure 9 illustrate the steps for DCOM and CORBA, respectively. The main difference between DCOM and CORBA at this layer include *how remote interface pointers or object references are represented to convey the server endpoint information to the client*, and *the standard*

format in which the data is marshaled for transmission in a heterogeneous environment. Note that CORBA does not specify a protocol for communication between a client and an object server running on ORBs provided by the same vendor. The protocol for inter-ORB communication between the same vendor ORBs is vendor dependent. However, in order to support the interoperability of different ORB products, a *General Inter-ORB Protocol (GIOP)* is specified. A specific mapping of the GIOP on TCP/IP connections is defined, and known as the *Internet Inter-ORB Protocol (IIOP)*. For CORBA, we include the descriptions for both IIOP and the Orbix implementation.

DCOM wire protocol is mostly based on OSF DCE RPC specification [DCE 95], with a few extensions. That includes remote object reference representation, an `IRemUnknown` interface for optimizing the performance of remote `IUnknown` method calls, and a *pinging protocol* [Brown 96]. Pinging allows a server object to garbage-collect remote object references when a remote client abnormally terminates. When a client obtains an interface pointer to a remote object for the first time, the ping client code (see Figure 2) on the client machine adds the object to a ping set and periodically sends a ping to the server machine to let it know that the client is still alive. Missing a predetermined number of consecutive pings indicates that the client has abnormally terminated and the interface pointers that it holds can be released. To optimize performance, pings are sent on a per-machine basis and in an incremental way. They can also be piggy-backed on normal messages. Whenever necessary, the ping functionality can also be turned off to reduce network traffic.

DCOM	CORBA
Object activation	
1. Upon receiving the delegated CoCreateInstance() request, if the client-side SCM consults local registry and finds out that the grid object should be located on another server machine, it calls a method of the IRemoteActivation RPC interface on the server-side SCM.	1. Upon receiving the delegated grid::_bind() request, client-side ORB consults a locator file to choose a machine that supports grid, and sends a request to the server-side ORB via TCP/IP.
2. When the server is started by the server-side SCM, it is associated with an object exporter and assigned an object exporter identifier (OXID). The mapping from the OXID to the RPCbinding that can be used to reach the server is registered with the server-side OXID resolver.	2. When the server is started by the server-side ORB, a grid object is instantiated by the server, the CORBA::Object constructor is called and BOA::create() is invoked. Inside the BOA::create(), BOA creates a socket endpoint, the grid object is assigned a object ID, unique within the server, an object reference is created, that contains the interface and the implementation names, the reference ID, and the endpoint
3. When the object stub marshals the IID_IGrid1 pointer returned by the CreateInstance(), the pointer is assigned an	

interface pointer identifier (IPID), unique within the server. Also, an object reference (OBJREF) is created to represent the pointer. An OBJREF contains the IPID, OXID, addresses of OXID resolvers (one per protocol), etc.	address. For clients talking the IIOP protocol, the server generates an **Interoperable Object Reference (IOR)** that contains a machine name, a TCP/IP port number, and an object_key. The BOA registers the object reference with the ORB.
4. When the marshaled interface pointer is returned to the client side through the server-side and client-side SCM's, the object proxy extracts the OXID and addresses of OXID resolvers from OBJREF, and calls the IOXIDResolver:ResolveOxid() method of its local OXID resolver.	3. When the object reference is returned to the client side, the proxy extracts the endpoint address and establishes a socket connection to the server.
5. The clients-side OXID resolver checks if it has a cached mapping for the OXID; if not, it invokes the IOXIDResolver:ResolveOxid() method of the server-side OXID resolver which returns the registered RPC binding.	
6. The client-side resolver caches the mapping, and returns the RPC binding to the object proxy. This allows the object proxy to connect itself and the interface proxies that it creates to an RPC channel that is connected to the object exporter.	

Method invocation	
1. Upon receiving pIGrid1->get() call, the interface proxy marshals the parameters in the **Network Data Representation (NDR)** format [DCE 95].	1. Upon receiving gridVar->get() call, the proxy marshals the parameters in the **Common Data Representation (CDR)** format [CORBA 95].
2. The RPC channel sends the request to the target object exporter identified by the OXID-resolved RPC binding.	2. The request is sent to the target server through the established socket connection.
3. The server-side RPC infrastructure finds the target interface stub based on the IPID that is contained in the RPC header.	3. The target skeleton is identified by either the reference ID or object_key.
4. After invoking the actual method on the server object, the interface stub marshals the return values in the NDR format.	4. After invoking the actual method on the server object, the skeleton marshals the return values in the CDR format.
5. Upon receiving the delegated pIGrid1->QueryInterface() call, the object proxy invokes the IRemUnknown::RemQueryInterface method on the OXID object [footnote 4] in the target object exporter. The OXID object then invokes the QueryInterface() method on (possibly multiple) interfaces within the exporter.	
6. Upon receiving the delegated pIGrid2->Release() call, the object proxy invokes the IRemUnknown::RemRelease() method on the OXID object in the target object exporter. The OXID object then invokes the Release() method on (possibly multiple) interfaces within the exporter.	

Table 8: The bottom layer description.

Footnote 4: There is one OXID object per object exporter. Each OXID object supports an `IRemUnknown` interface consisting of three methods: `RemQueryInterface()`, `RemAddRef()`, and `RemRelease()`. These methods allow multiple remote `IUnknown` method calls destined for the same object exporter to be bundled to improve performance. All such calls are first handled by the OXID object, and then forwarded to the target interface. Note that these and other bottom-layer APIs are essentially implementation details. Application programmers will not encounter them.

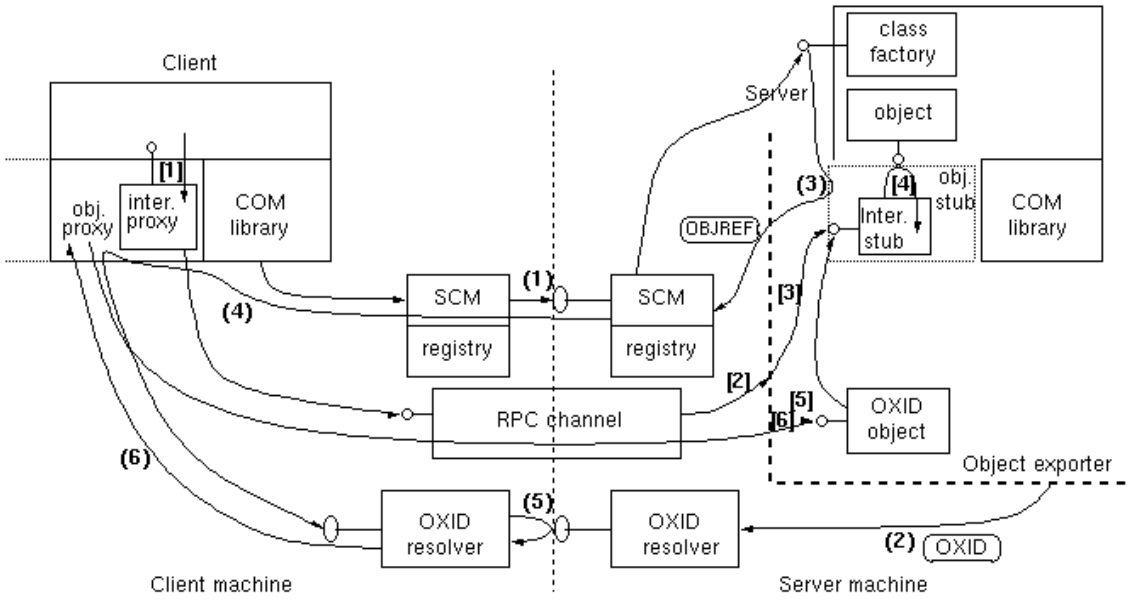

Figure 8: DCOM steps at the bottom layer.

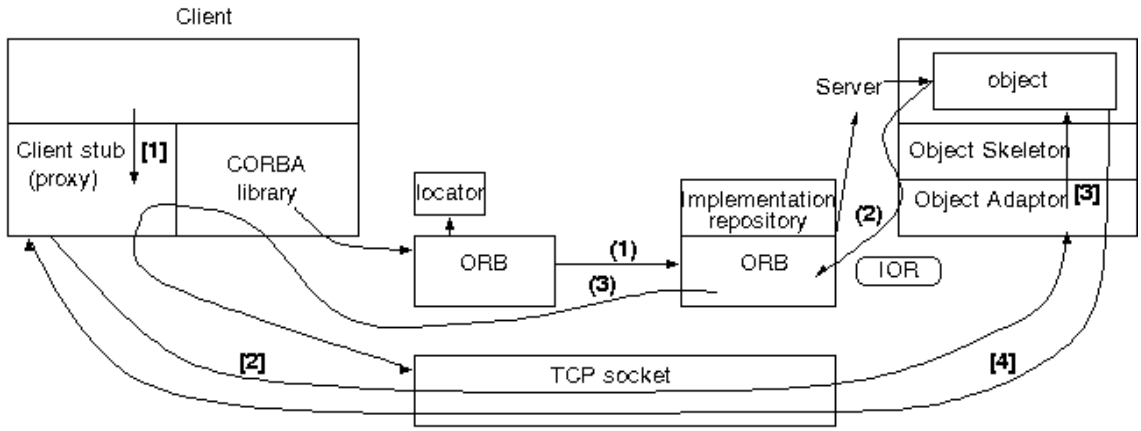

Figure 9: CORBA steps at the bottom layer.

6. Summary

The three-layer step-by-step descriptions have shown that the architectures of DCOM and CORBA/Orbix are basically similar. They both provide the distributed objects infrastructure for transparent activations and accessing of remote objects. Table 9 summarizes the corresponding terms and entities in the two architectures. Note that many of the correspondences are only approximate. Their main differences are also summarized below. First, DCOM supports objects with multiple interfaces and provides a standard `QueryInterface()` method to navigate among the

interfaces. This also introduces the notion of an object proxy/stub dynamically loading multiple interface proxies/stubs in the remoting layer. Such concepts do not exist in CORBA. Second, every CORBA interface inherits from `CORBA::Object`, the constructor of which implicitly performs such common tasks as object registration, object reference generation, skeleton instantiation, etc. In DCOM, such tasks are either explicitly performed by the server programs or handled dynamically by DCOM run-time system. Third, DCOM's wire protocol is strongly tied to RPC, but CORBA's is not. Finally, we would like to point out that DCOM specification contains many details that are considered as implementation issues and not specified by CORBA. As a result, we have used the Orbix implementation in many places in order to complete the side-by-side descriptions.

	DCOM	**CORBA**
Top layer: Basic programming architecture		
Common base class	`IUnknown`	`CORBA::Object`
Object class identifier	`CLSID`	interface name
Interface identifier	`IID`	interface name
Client-side object activation	`CoCreateInstance()`	a method call/`bind()` [footnote 5]
Object handle	interface pointer	object reference
Middle layer: Remoting architecture		
Name to implementation mapping	Registry	Implementation Repository
Type information for methods	Type library	Interface Repository
Locate implementation	SCM	ORB
Activate implementation	SCM	OA
Client-side stub	proxy	stub/proxy
Server-side stub	stub	skeleton
Bottom layer: Wire protocol architecture		
Server endpoint resolver	OXID resolver	ORB
Server endpoint	object exporter	OA
Object reference	OBJREF	IOR (or object reference)
Object reference generation	object exporter	OA
Marshaling data format	NDR	CDR
Interface instance identifier	IPID	object_key

Table 9: Summary of corresponding terms and entities.

Footnote 5: Orbix and most other CORBA vendors provide `bind()` as a way of activating an object and getting its object reference. However, bind() is not specified in the CORBA standard. The standard suggests that a client obtains an object reference from a naming service or a trader service.

Acknowledgment

The authors would like to express thanks to Richard Buskens and Yow-Jian Lin at Bell Labs for their valuable discussions and their assistance in prototyping CORBA applications, and to Chandra Kintala (Bell Labs), Doug Schmidt (Washington University), Charlie Kindel (Microsoft), Nat Brown (Microsoft), Don Box (DevelopMentor), Prem Devanbu (AT&T Labs) for their valuable comments.

References

[Birrell 84]
A. Birrell and B. J. Nelson, Implementing Remote Procedure Calls, *ACM Transactions on Computer Systems*, Vol. 2, No. 1, Feb 1984, pp.39-59.
[Box1 97]
D. Box, Q&A ActiveX/COM, *Microsoft Systems Journal*, March 1997, pp.93-105.
[Box2 97]
D. Box, Q&A ActiveX/COM, *Microsoft Systems Journal*, July 1997, pp.93-108.
[Brockschmidt 93]
K. Brockschmidt, Inside OLE, Redmond, Washington: Microsoft Press, 1993.
[Brown 96]
N. Brown, C. Kindel, Distributed Component Object Model Protocol -- DCOM/1.0, http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt.
[Chappell 96]
D. Chappell, Understanding ActiveX and OLE, Redmond, Washington: Microsoft Press, 1996.
[Chappell 97]
D. Chappell, The Joy of Reference Counting, in *Object Magazine*, pp. 16-17, July 1997.
[COM 95]
The Component Object Model Specification, http://www.microsoft.com/oledev/olecom/title.htm.
[CORBA 95]
The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995, http://www.omg.org/corba/corbiiop.htm.
[DCE 95]
AES/Distributed Computing - Remote Procedure Call, Revision B, Open Software Foundation, http://www.osf.org/mall/dce/free_dce.htm.
[Grimes 97]
R. Grimes, Professional DCOM Programming, Olton, Birmingham, Canada: Wrox Press, 1997.
[Iona 96]
Orbix 2.1 Programming guide and Reference guide, Iona technologies Ltd., http://www.iona.com/.
[Orbix 96]
The Orbix Architecture - IONA Technologies, November 1996. http://www.iona.com/Products/Orbix/Architecture/index.html.

[Orfali 97]
R. Orfali, D. Harkey, J. Edwards, Instant CORBA, Wiley Computer Publishing, John Wiley & Sons, Inc., 1997.
[POA 97]
ORB Portability Joint Submission, Part 1 of 2, orbos/97-04-14, http://www.omg.org/library/schedule/Technology_Adoption.htm .
[Rogerson 96]
D. Rogerson, Inside COM, Redmond, Washington: Microsoft Press, 1996.
[Schmidt 97]
D. Schmidt, S. Vinoski, Object Interconnectins - Object Adapters: Concepts and Terminology (Column 11), to appear in SIGS C++ Report Magazine, October 1997. http://www.cs.wustl.edu/~schmidt/C++-report-col11.ps.gz .
[Vinoski 97]
S. Vinoski, CORBA: Integrating diverse applications within distributed heterogeneous environments, in IEEE Communications, vol. 14, no. 2, Feb. 1997. http://www.iona.com/hyplan/vinoski/ieee.ps.Z.
[Wang1 97]
Y. M. Wang, Introduction to COM/DCOM, http://akpublic.research.att.com/~ymwang/slides/DCOMHTML/ppframe.htm, 1997.
[Wang2 97]
Y. M. Wang, COM/DCOM Resources, http://akpublic.research.att.com/~ymwang/resources/resources.htm, 1997.