2

Distributed Object Oriented Approaches

Chris Horn and Annrai O'Toole
IONA Technologies Ltd.,
8-34 Percy Place
IRL-Dublin 2
email: info@iona.ie   tel:+353.1.6686522

Abstract
This short document gives an overview of two leading object infrastructure technologies, the Microsoft COM/OLE and the OMG's OMA and CORBA. The short paper describes the history, origins and context for the development of both technologies. This is accompanied by a brief technical overview of the major architectural issues deployed in realising each technology.

Keywords
COM, CORBA, OLE, Distributed Objects

1. COM AND OLE

When the Microsoft PowerPoint team, based in Mountain View, Ca. began building their own graphing software a dictum came down from Bill Gates forbidding such an exercise. He ordained that the PowerPoint team should, indeed must, use the Excel graphing software. It was wasteful to reinvent that particular software component. Thus was OLE1.0 conceived. As Philippe Kahn of Borland described it, OLE1.0 enabled the "sharing of real-estate on the desktop". PowerPoint could hand over the rendering of data on a piece of the screen that it owned to a third party. In more sophisticated terms, OLE1.0 enabled the linking and embedding of third party rendering software within a foreign application. It was a large step forward towards software components.

At the same time as OLE1.0 was being developed, Microsoft were also grappling with the problem of how to make Dynamic Link Libraries (DLLs) more usable. In real terms a DLL is nothing more that a list of entry vectors. There are huge problems associated with the versioning of DLLs. If a new DLL were installed on a machine (as part of another application installation for instance) it could easily overwrite existing

DLLs. Then when the old applications went to suck in the DLL they were in for a small surprise. Then entry vectors could well be out of kilter, resulting in that favourite of Windows error messages: GPF!

Work was begun on making OLE1.0 for generic and also trying to find a way to solve the DLL evolution problem. The end result was OLE2.0. OLE2.0 has undergone some serious work since the first release in 1993. We are now at OLE 2.02, and the minor version number does no justice to the rather large changes that have been added to OLE. The OLE2.02 architecture is described in Figure 1 below:

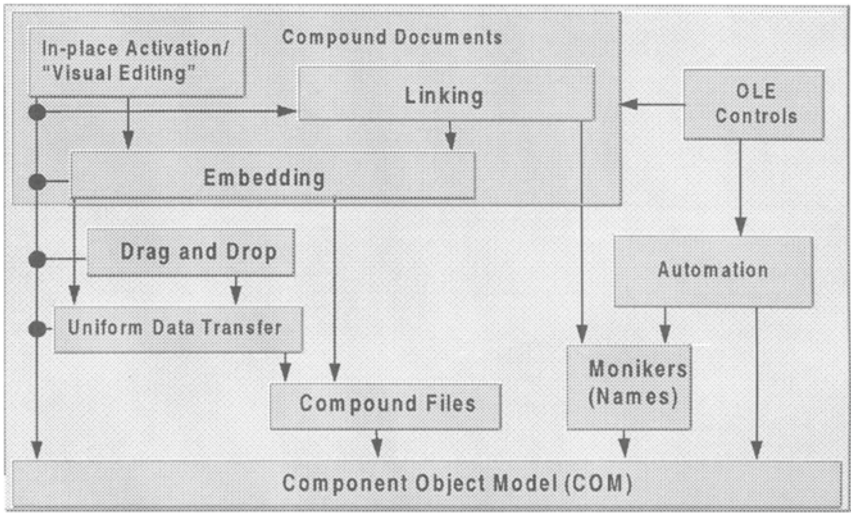

Figure 1: OLE 2.02 Architecture

The cast of characters from the ground up are:

1.1 COM

The Component Object Model is the basis for all COM and OLE objects. All COM objects inherit from the Base Type `IUnknown`. This supports three basic operations: `AddRef`, `Release` and `QueryInterface`. Before discussing these it must be noted that COM is a binary object model, i.e. the interface to a COM object is defined as a list of entry points (remember DLLs!). Thus while inheriting from `IUnknown` the programmer defines the object by listing all the entry points, or `vtable` for this object. The reason for this approach is simple. It enables objects to be written in any language using any compliant compiler (just like DLLs). All the COM expects from this object is that the entry points are clearly and correctly laid out. In turn this approach supports, as the Microsoft marketing organisation like to call it, "shrink wrapped binary objects". The end user is free to buy any software component, such as a graphing tools, from any vendor. It can be delivered in binary form on a floppy and should plug straight into my application. This is essential for desktop software.

Of course Microsoft do not expect developers to go around defining interfaces to their objects in terms of the layout of the entry points in memory: they provide some tools to help in the process. Tools such as the ODL and MIDL compilers (which will be discussed later) are provided by Microsoft as conveniences to developers.

Another fundamental issue about COM is that it frowns on inheritance. The official Microsoft religion disclaims inheritance as an evil blasphemy. Aggregation and delegation are the preferred methods! (It is not entirely clear whether Microsoft really dislike inheritance or whether the issue of things like multiple inheritance would have made the implementation of the binary model more cumbersome than it is worth!)

So, in the absence of inheritance how does client code traverse the hierarchy, given that the only guaranteed thing we know about each object is that it supports `IUnknown`? Easy, we use `QueryInterface`. Essentially we come along to each object and we ask it: Are you one of these? The object is then free to determine whether it actually is (or can perhaps locate) an object of the type the client is looking for. If successful, `QueryInterface` returns the pointer to an implementation which fulfills the requirements desired by the client. This is how COM solves the evolution of DLLs. In an ideal operating system world there would be no DLLs, just COM objects. An application would ask the COM object if it supported a particular interface. If it were a newer version of the object than the application was used to then it would be easy for that COM object to return the application a pointer to a implementation which would conform to the older interface, while also being able to respond positively to applications that were prepared to use the newer interface.

In many cases it may be more useful to think of `QueryInterface` as `QueryImplementation`, because what the client is really asking is: "do you support this implementation?"

At an abstract level, it is important to note that the COM notion of an object is very much a transitory affair. Although each COM object has a unique identifier (a 128 bit Global Unique Identifier - GUID), these are really used to identify and distinguish between different interface types. The process in creating and using a COM object is something like: Ask COM to create a COM object denoted by a GUID that the client supplies; ask that object to load state from a location supplied by the client; perform some operations on that object; and finally ask to object to unload it's state back into the location supplied by the client. There is no equivalence of a single object reference which can uniquely define the combination of an object's interface, code and state.

Once COM is understood is becomes easy to understand the rest of OLE. In simple terms, OLE is merely a collection of predefined and per-implemented COM objects, or components, which provide various levels of service to applications.

1.2 OLE

OLE version 2.0 was the first deployment of a subset of the COM specification that included support for local objects (both in-process and local) and all the infrastructure technologies but did not support remote or networked objects. OLE 2 includes mostly user-interface oriented features based on usability, application integration, and automation of tasks.

1.3 Persistent Storage

This is a set of interfaces and an implementation of those interfaces that create structured storage, otherwise known as a "file system within a file." Information in a file is structured in a hierarchical fashion which enables sharing storage between processes, incremental access to information, transactioning support, and the ability for any code in the system to browse the elements of information in the file. In addition, COM defines standard "persistent storage" interfaces that objects implement to support the ability to save their persistent state to permanent, or persistent, storage devices such that the state of the object can be restored at a later time.

1.4 Monikers

Monikers allow a specific *instantiation* of an object to be given a particular name, so that a client can reconnect to that *exact same object instance with the same state* (not just another object of the same class) at a later time. This also includes the ability to assign a name to some sort of *operation*, such as a query, that could be repeatedly executed using only that name to refer to the operation. This level of indirection allows changes to happen behind the name without requiring any changes to the client that stores that particular name. This technology is centered around a type of object called a *moniker* and COM defines a set of interfaces that moniker objects implement. COM also defines a standard *composite moniker* that is used to create complex names that are built of simpler monikers. Monikers also implement one of the persistent storage interfaces meaning that they know how to save their name or other information to somewhere permanent. Monikers are "intelligent" because they know how to take the name information and somehow relocate the specific object or perform an operation to which that name refers.

1.5 Uniform Data Transfer

A set of interfaces through which data is exchanged between a client and an object and through which a client can ask an object to send notification (call event functions in the client) in case of a data change. The interfaces include support structures used to describe data formats as well as the storage mediums on which the data is exchanged.

The combination of the foundational and the infrastructural COM components reveals a system that describes how to create and communicate with objects, how to store them, how to label to them, and how to exchange data with them. These four aspects of COM form the core of information management. Furthermore, the infrastructure components not only build on the foundation, but monikers and uniform data transfer also build on storage as shown in Figure 2. . The result is a system that is not only very rich, but also deep, which means that work done in an application to implement lower level features is leveraged to build higher level features.

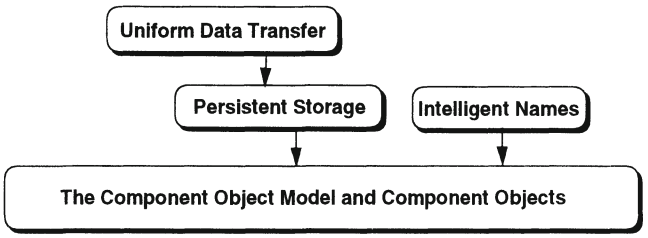

Figure 2: COM is built in progressively higher level technologies that depend upon lower level technologies.

1.6 Drag and Drop

The ability to exchange data by picking up a selection with the mouse and visibly dropping it onto another window.

1.7 Compound Documents

The ability to embed or link information in a central document encourages a more document-centric user interface. This also includes In-Place Activation (also called "Visual Editing") as a user interface improvement for embedding where the end user can work on information from different applications in the context of the compound document, without having to switch to other windows.

Microsoft in cooperation with other vendors is continuing to enhance OLE with new interfaces to extend compound documents and to define architectures for creating components such as OLE Controls, OLE DB, OLE for Design & Modeling, OLE for Healthcare, and in the future more system-level OLE architectures that build not only on the COM infrastructure but also on the rest of OLE as well. Again, the key is leveraged work: by implementing lower level features in an application you create a strong base of reusable code for higher level features.

1.8 Automation

The ability to create "programmable" applications that can be driven externally from a script running in another application to automate common end user tasks. Automation enables cross-application macro programming.

Automation was built by the Visual Basic team at Microsoft and it used to enable VB script to other applications. i.e. through Automation a VB program can launch and control an application such as Excel.

Automation is somewhat similar to the dynamic invocation interface in CORBA. A scripting language cannot be pre-compiled with the stubs needed to access a object. Through Automation the scripting language can discover the information about an object interface at runtime. This is achieved through the use of TypeLibraries. Each Automation server must provide a description of its interface in a TypeLibrary (TypeLib). The scripting tool can then read the TypeLib information and ensure that the client is only trying to perform valid operations on the Automation server. For this reason, Automation is described as "late binding".

Microsoft provide a tool to help in writing Automation Servers. Using a language called the "Object Description Language" (ODL), a developer can describe the interface to an Automation Server. This ODL is then used to automatically generate the TypeLib information needed so that a client can script to that Automation Server.

1.9 OLE Controls

The final piece of the OLE architecture is OLE Controls. This architecture feature circles the OLE square. Before OLE Controls there were two distinct pieces to OLE: Compound Documents and Automation. A programmer could write a GUI software component using the Compound Document features or they could write a non-visible, "programmatic" Automation server using Automation. Now with OLE Controls they can write GUI Automation Servers. OLE Controls allow the programmer to associate logic functions with entities that can be displayed on the screen and embedded in Containers.

At time of writing, the main target Container for OLE Controls is VisaulBASIC 4.0. VB3.0 allowed the programmer to extend the VB environment through the use of VisualBASIC Extensions (VBXs). in VB4, these have been replaced by OLE Controls or OCXs.

2. OMG

The Object Management Group was established in 1989 with the explicit aim of building a consensus based approach to the problems of application integration. From an OMG perspective, the largest problems needing to be tackled in the software industry at that point were those relating to application integration. There existed no standards based solution for application integration. The OMG took a longterm view on what application integration actually meant. For them it was more than the sharing of real estate on the desktop, but rather it must encompass networking, programming languages, heterogeneous platforms and differing implementation choices. In other words, true application integration should enable an application to use and share another application component regardless of the language in which that component is written, the type of operating system it is running on or indeed its location in the network regardless of the networking protocol employed.

2.1 OMA

With this ambitious goal in mind the OMG set about creating an architecture to meet these criteria. It was an architecture to be based on the concepts extolled by the object oriented paradigm. An object approach was adopted, not because object are "good" but rather that object technology seemed to offer the best technical solution to the problems of application integration.

The resulting architecture created by the OMG is called the "Object Management Architecture" (OMA), outlined below in Figure 3.

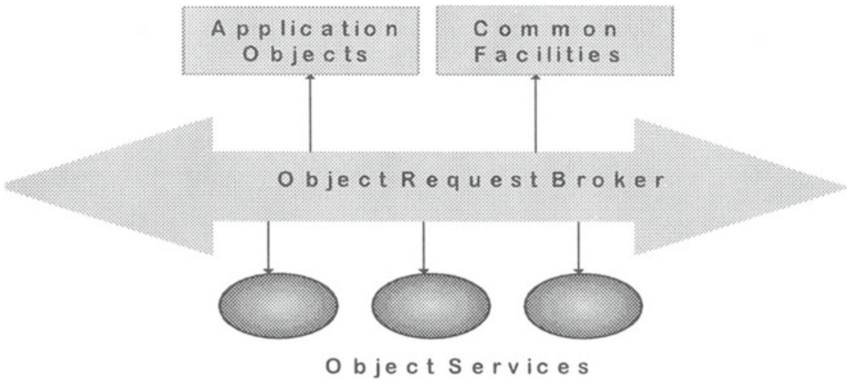

Figure 3: The Object Management Architecture

2.2 CORBA

At the heart of the OMA is a central software bus, or Object Request Broker. The aim of this component is to regularise the communication between the various connected "object" or software components. The concrete instantiation of this entity is defined in the "Common Object Request Broker: Architecture and Specification" document published by the OMG. It is more normally referred to as CORBA.

The central component of CORBA is the Interface Definition Language (OMG IDL). OMG IDL is to object systems, what DDL is to databases. Programmers use OMG IDL to describe the interface to their objects. OMG IDL is the fundamental basis for the definition of the contract exposed by the object to the rest of the world.

A sample OMG IDL description might look like:

```
interface Person {
        attribute long Age;
        attribute float Height;

        oneway void Marry(in Person Spouse);
        boolean getJob(out float Salary);
};
```

Programmers who wish to develop CORBA objects must first begin by designing the OMG IDL for the objects they wish to create. Having completed the OMG IDL specification an implementor is then free to implement that language in any programming language (there are currently OMG mappings for C, C++, Ada and Smalltalk).

The actual CORBA architecture is depicted below in Figure 4:

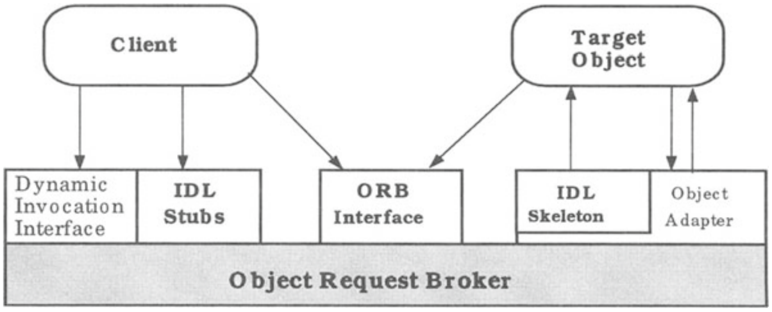

Figure 4: CORBA Architecture

When a client wishes to avail of the service provided by an OMG IDL object it can do so in two ways: it can use a static invocation or dynamic invocation mechanism. The static approach assumes that the client has pre-compiled knowledge of the OMG IDL service it is about to use. (In concrete terms this means that the client application has been built with a generated stub or proxy for that remote object). Using the dynamic approach, the client must make a runtime discovery of the OMG IDL interface of the object. To do this it contacts the Interface Repository which is used to store runtime information about every OMG IDL interface. Having obtained at runtime the details about the interface supported by the target object, the client can use the DII to build a Request and sent that Request to the target object.

In CORBA, objects are created in one location and remain at that location for a given lifetime. The entities which are passed over the network are "object references". An object reference is a unique identifier used to locate and describe a given instance of a given object type.

2.3 Object Services

The CORBA specification defines a basic software bus. In addition, the OMA provides for a set of Object Services. Since 1991, the OMG have been populating the Object Service space with a range of specifications. The full set of Object Services are as follows (services marked in bold have had their formally specifications adopted by the OMG):

- **Naming**
  A directory service which enables object references to be named by high level "human readable" names. The service is organised a collection of contexts (directories). These can be related in a hierarchical or federated manner. A universal "root" is not assumed.

- **Persistence**
  Enables objects to exist beyond the lifetime of their creator.

- **Life Cycle**
  Provides a simple service for creating, destroying, copying and moving objects. Is in large part a style guide on what sort of operations a "well behaved" object interface should provide.

- Properties

- **Concurrency**
  Provides interfaces to acquire and release locks that let multiple clients coordinate their access to shared resources.

- Collections

- Security

- Trader
  Provides a "matchmaking" service for objects. Enables one object to establish a link to another object based on a set of arbitrary properties, i.e. the printer object which "is closest to me" or the "hotel object which has free rooms", etc.

- **Externalisation**
  Enables an object to stream in and stream out its internal state.

- **Events**
  Provides a framework whereby objects can exchange events among themselves. Enables the creation of Event channels to which events can be "pushed" or "pulled".

- **Transactions**
  The Transaction service enables programmers to invoke objects within the context of a transaction which provides the standard ACID properties. Two usage scenarios are envisaged within the Transaction service. A transaction context can be passed implicitly: When an operation on an object is invoked within a transaction, the OTS will ensure that that object is correctly involved in the transaction without any programmer intervention. Alternatively, with explicit transaction support, the programmer can choose to explicitly control which objects are involved in the transaction.

- **Query**
  Enables an object to select an object based on its attributed: e.g.. all Shape objects whose colour attribute is red.

- **Relationships**
  Provides the basic service which lets objects to be "related" to one another in a fully dynamic way: e.g. a bank is related to accounts because it maintains them for customers.

- Time

- Change Management

- **License**
  Enables an object to license itself to control authorised and paid up usage.

All Object Services (or CORBAServices) are specified in IDL. In addition, the specification provide a number of conformance points which ensure interoperability between different implementations of those services.

2.4 Common Facilities

The Common Facilities, or CORBAFacilities provide a layer of application specific services. The first round of CORBAFacilities being adopted by the OMG are those for Compound Documents. The two main technologies under consideration are OpenDoc (an alternative to OLE) and Fresco (a C++ class library for structured graphics).

2.5 Interoperability

With the CORBA 2.0 specification comes a description of a mandatory protocol which all ORBs must support in order to be a CORBA2.0 compliant implementation. The basis of this protocol is as follows:

- The **General Inter-ORB Protocol** (GIOP) specifies a set of message formats and common data representation for communications between ORBs. The GIOP is specifically designed for a CORBA-to-CORBA communication. It is based on the principle of KISS (Keep it Simple, Scaleable).

- The **Internet Inter-ORB Protocol** (IIOP) is a concrete instantiation of the GIOP over a TCP/IP communication protocol. This is the mandatory protocol which all CORBA2 implementations must support.

- The **Environment Specific Inter-ORB Protocols** (ESIOPs) are intended for alternative implementation of the GIOP over different networking protocols. The first ESIOP to be adopted is support for GIOP over the DCE RPC. Other ESIOPs, such as support for IPX/SPX and NetBIOS are envisaged.

The IIOP brings binary interoperability to CORBA. Through the combination of IDL and the IIOP, CORBA offers a comprehensive solution to interoperability and "shrink wrapped" network objects.

3. Conclusions

This short paper has given an overview of both COM and CORBA technologies. It is clear that while both started out with the familiar theme of application integration the two have taken different tracks in achieving that goal. COM (and OLE) with a document centric approach has developed leading desktop technology and is now trying to extend that base into the network. OMG with the OMA and CORBA have adopted a very network centric approach and are now extending towards the desktop. It is clear the two will meet in the middle.

From a technical perspective this paper has focused on highlighting the difference in the object models used by each technology. COM adopts a binary object model whereas CORBA has focused on a language based approach. Each have their merits and domain of applicability. COM and OLE are likely to remain the desktop standard whereas CORBA is likely to win the hearts and minds of the world networks.