CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments

Steve Vinoski
vinoski@iona.com
IONA Technologies, Inc.
60 Aberdeen Ave.
Cambridge, MA USA 02138 [1]

1 Introduction

An important characteristic of large computer networks such as the Internet, the World Wide Web (WWW), and corporate intranets is that they are *heterogeneous*. For example, a corporate intranet might be made up of mainframes, UNIX workstations and servers, PC systems running various flavors of Microsoft Windows, IBM OS/2, or Apple Macintosh, and perhaps even devices such as telephone switches, robotic arms, or manufacturing testbeds. The networks and protocols underlying and connecting these systems might be just as diverse: Ethernet, FDDI, ATM, TCP/IP, Novell Netware, and various remote procedure call (RPC) [1] systems, for example. Fundamentally, the rapidly-increasing extents of these networks are due to the need to share information and resources within and across diverse computing enterprises.

Heterogeneity in such computing systems is the result of several factors:

• **Engineering tradeoffs:** There is rarely only a single acceptable solution to a complex engineering problem. As a result, different people across an enterprise often choose different solutions to similar problems.

• **Cost effectiveness:** Vendors vary in their abilities to provide the "best" systems at the lowest cost. Though there is some amount of "brand name loyalty," many consumers tend to buy the systems that best fulfill their requirements at the most reasonable price, regardless of who makes them.

• **Legacy systems:** Over time, purchasing decisions accumulate, and already-purchased systems may be too critical or too costly to replace. For example, a company that has been successfully running its order fulfillment applications, which are critical to its day-to-day operations, on its mainframe for the last fifteen years is not likely to simply scrap their system and replace it with the latest fad technologies. Alternatively, a company may have spent large sums of money on its current systems, and those systems must be utilized until the investment has paid off.

Ideally, heterogeneity and open systems enable us to use the best combination of hardware and software components for each portion of an enterprise. When the right standards for interoperability and portability between these components are in place, the integration of the components yields a system that is coherent and operational.

Unfortunately, dealing with heterogeneity in distributed computing enterprises is rarely easy. In particular, the development of software applications and components that support and make efficient use of heterogeneous networked systems is very challenging. Many programming interfaces and packages currently exist to help ease the burden of developing software for a single homogeneous platform. However, few help deal with the integration of separately-developed systems in a distributed heterogeneous environment.

In recognition of these problems, the Object Management Group (OMG)[2] was formed in 1989 to develop, adopt, and promote standards for the development and deployment of applications in distributed heterogeneous environments. Since that time, the OMG has grown to become the largest software consortium in the world, with over 700 developers, vendors, and end users on its membership roster. These members contribute technology and ideas in response to *Requests For Proposals* (RFPs) issued by the OMG. Through responses to these RFPs, the OMG adopts specifications based on commercially-available object technology.

This article describes the OMG's *Object Management Architecture* (OMA) [2] and focuses on one of its key components, the *Common Object Request Broker Architecture*

[2]OMG and Object Management are registered trademarks of the Object Management Group. CORBA, OMG Interface Definition Language, CORBAmed, CORBAtel, and CORBAnet are trademarks of the Object Management Group.

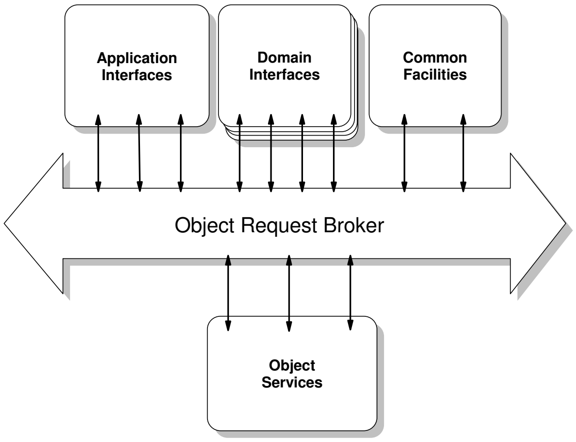

Figure 1: OMA Reference Model Interface Categories

(CORBA) specification [3]. First, a brief high-level overview of the OMA is provided, followed by a detailed outline of CORBA and each of its subcomponents. The summary section lists some of the OMG's current and future plans for further promoting distributed object technology.

2 The Object Management Architecture (OMA)

The OMA is composed of an *Object Model* and a *Reference Model*. The Object Model defines how objects distributed across a heterogeneous environment can be described, while the Reference Model characterizes interactions between those objects. The OMG RFP process is used to adopt technology specifications that fit into the Object Model and the Reference Model and work with the other previously-adopted specifications. Through adherence to the OMA, these specifications allow for the development and deployment of interoperable distributed object systems in heterogeneous environments.

In the OMA Object Model, an object is an encapsulated entity with a distinct immutable identity whose services can be accessed only through well-defined *interfaces*. *Clients* issue requests to objects to perform services on their behalf. The implementation and location of each object are hidden from the requesting client.

Figure 1 shows the components of the OMA Reference Model. The *Object Request Broker* (ORB) component is mainly responsible for facilitating communication between clients and objects. Utilizing the ORB component are four object interface categories:

1. Object Services: These are domain-independent interfaces that are used by many distributed object programs. For example, a service providing for the discovery of other available services is almost always necessary regardless of the application domain. Two examples of Object Services that fulfill this role are:

- The *Naming Service* – which allows clients to find objects based on names;
- The *Trading Service* – which allows clients to find objects based on their properties.

There are also Object Service specifications for lifecycle management, security, transactions, and event notification, as well as many others [4].

2. Common Facilities: Like Object Service interfaces, these interfaces are also horizontally-oriented, but unlike Object Services they are oriented towards end-user applications. An example of such a facility is the *Distributed Document Component Facility* (DDCF) [5], a compound document Common Facility based on OpenDoc.[3] DDCF allows for the presentation and interchange of objects based on a document model, for example, facilitating the linking of a spreadsheet object into a report document.

3. Domain Interfaces: These interfaces fill roles similar to Object Services and Common Facilities but are oriented towards specific application domains. For example, one of the first OMG RFPs issued for Domain Interfaces is for Product Data Management (PDM) Enablers[4] for the manufacturing domain [6]. Other OMG RFPs will soon be or already have been issued in the telecommunications, medical, and financial domains. In Figure 1, multiple boxes are shown for Domain Interfaces to indicate the existence of many separate application domains.

4. Application Interfaces: These are interfaces developed specifically for a given application. Because they are application-specific, and because the OMG does not develop applications (only specifications), these interfaces are not standardized. However, if over time it appears that certain broadly useful services emerge out of a particular application domain, they might become candidates for future OMG standardization.

Figure 2 illustrates the other part of the OMA Reference Model, the concept of *Object Frameworks*. These are domain-specific groups of objects that interact to provide a customizable solution within that application domain. These frameworks are typically oriented towards domains such as telecommunications, medical systems, finance, and manufacturing. In Figure 2, each circle represents a component that uses the ORB to communicate with other components. The interfaces supported by each component are indicated on its outer circle. As the figure shows, some components support application-specific interfaces, as well as domain interfaces, common facilities interfaces, and object services. Other components support only a subset of these interfaces.

[3] OpenDoc is a trademark of Apple Computer, Inc.

[4] "Enabler" is a term derived from Total Quality Management principles. It is simply defined as any entity, such as a computer program or human activity, that provides or supports an abstract business process, *e.g.*, handling engineering change orders.

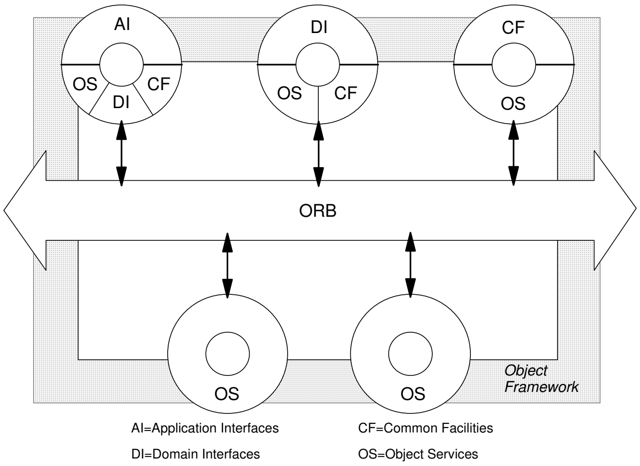

Figure 2: OMA Reference Model Interface Usage

Within an object framework like the one shown in figure 2, each component communicates with others on a *peer-to-peer* basis. That is, each component is both a *client* of other services and a *server* for the services it provides. In CORBA, the terms "client" and "server" are merely roles that are filled on a per-request basis. Very often, a client for one request is the server for another.

Throughout most of its existence, much of the OMG's attention was focused on the ORB component of the OMA. This was necessary because everything else in the OMA depends on the ORB. The rest of this article will focus on the ORB, its components, and how it is used to support distributed object systems. For more information about the upper layers of the OMA, see [7] or visit the OMG home page on the WWW at `http://www.omg.org/`.

3 The Common Object Request Broker Architecture (CORBA)

One of the first specifications to be adopted by the OMG was the CORBA specification. It details the interfaces and characteristics of the ORB component of the OMA. As of this writing, the last major update of the CORBA specification was in mid-1995 when the OMG released CORBA 2.0 [3]. The main features of CORBA 2.0 are:

- ORB Core
- OMG Interface Definition Language (OMG IDL)
- Interface Repository
- Language Mappings
- Stubs and Skeletons
- Dynamic Invocation and Dispatch
- Object Adapters
- Inter-ORB Protocols

Most of these are illustrated in Figure 3, which also shows how the components of CORBA relate to one another. Each component is described in detail below.

3.1 ORB Core

As mentioned above, the ORB delivers requests to objects and returns any responses to the clients making the requests. The object that a client wishes the ORB to direct a request to is called the *target object*. The key feature of the ORB is the transparency of how it facilitates client/object communication. Ordinarily, the ORB hides the following:

• **Object location:** The client does not know where the target object resides. It could reside in a different process on another machine across the network, on the same machine but in a different process, or within the same process.

• **Object implementation:** The client does not know how the target object is implemented, what programming or scripting language(s) it was written in, nor the operating system (if any) and hardware it executes on.

• **Object execution state:** When it makes a request on a target object, the client does not need to know whether that object is currently *activated* (*i.e.*, in an executing process) and ready to accept requests. The ORB transparently starts the object if necessary before delivering the request to it.

• **Object communication mechanisms:** The client does not know what communication mechanisms (*e.g.*, TCP/IP, shared memory, local method call, etc.) the ORB uses to deliver the request to the object and return the response to the client.

These ORB features allow application developers to worry more about their own application domain issues and less about low-level distributed system programming issues.

To make a request, the client specifies the target object by using an *object reference*. When a CORBA object is created an object reference for it is also created. When used by a client, an object reference always refers to the same object for which it was created, for as long as that object still exists. In other words, an object reference only ever refers to one single object. Object references are both immutable and opaque, so a client can't "reach into" the object reference and modify it. Only an ORB knows what's "inside" an object reference. Object references can have standardized formats, such as those for the OMG standard *Internet Inter-ORB Protocol* and *Distributed Computing Environment Common Inter-ORB Protocol* (both of which are described in Section 3.8), or they can have proprietary formats.

Clients can obtain object references in several different ways:

• **Object creation:** A client can create a new object in order to get an object reference. Note that CORBA has no special client operations for object creation – making objects is done by invoking creation requests, which are just ordinary operation invocations, on other objects called *factory objects*.

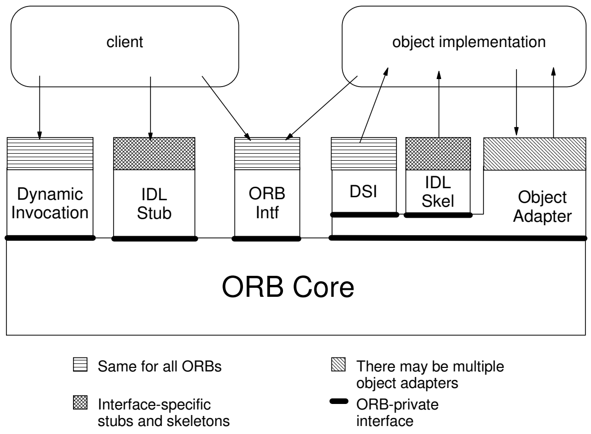

Figure 3: Common Object Request Broker Architecture

A creation request returns an object reference for the newly-created object to the client.

• **Directory service:** A client can invoke a lookup service of some kind in order to obtain object references. Two Object Services mentioned above, the Naming Service and the Trading Service, allow clients to obtain object references by name or by properties of the object, respectively. Unlike factory objects, these services do not create new objects. They store object references and associated information (*e.g.*, names and properties) for existing objects, and supply them upon request.

• **Convert to string and back:** An application can ask the ORB to turn an object reference into a string, and this string can be stored into a file or a database. Later, the string can be retrieved from persistent storage and turned back into an object reference by the ORB. Even after being *stringified* and *de-stringified* in this manner, it can still be used to make requests on the object as long as the object still exists.

Since CORBA has no special object creation operations, object references are always obtained by making requests on other objects. This begs the question of how an application can bootstrap itself and obtain an initial object reference. Not surprisingly, the ORB provides a small, simple "naming service" of its own to provide applications with object references of more general directory services like Naming and Trading. For example, by passing the string "NameService" to the ORB's `resolve_initial_references` operation, an application can obtain an object reference for the Naming Service that is known to its ORB.

The fact that CORBA has no special object creation functions or built-in directory services is indicative of a key theme of CORBA: *Keep the ORB as simple as possible, and push as much functionality as possible to other OMA components such as Object Services and Common Facilities*.[5] The job of the ORB is to simply provide the communication and activation infrastructure for distributed object applications.

3.2 OMG Interface Definition Language (OMG IDL)

Before a client can make requests on an object, it must know the types of operations supported by the object. An object's *interface* specifies the operations and types that the object supports and thus defines the requests that can be made on the object. Interfaces for objects are defined in the *OMG Interface Definition Language* (OMG IDL). Interfaces are similar to classes in C++ and interfaces in Java. An example OMG IDL interface definition is shown below:

```
// OMG IDL
interface Factory {
    Object create();
};
```

This definition specifies an interface named `Factory` that supports one operation, `create`. The `create` operation takes no parameters and returns an object reference of type `Object`. Given an object reference for an object of type `Factory`, a client could invoke it to create a new CORBA object. This interface might be supported by one of the factory objects mentioned above, for example.

An important feature of OMG IDL is its *language independence*. Since OMG IDL is a declarative language, not a programming language, it forces interfaces to be defined separately from object implementations. This allows objects to be constructed using different programming languages and yet still communicate with one another. Language-independent interfaces are important within heterogeneous systems, since not all programming languages are supported or available on all platforms.

OMG IDL provides a set of types that are similar to those found in a number of programming languages. It provides basic types such as `long`, `double`, and `boolean`, constructed types such as `struct` and discriminated `union`, and template types such as `sequence` and `string`. Types are used to specify the parameter types and return types for operations. As seen in the example above, operations are used within `interfaces` to specify the services provided by those objects that support that particular interface type. To define exceptional conditions that may arise during the course of an operation, OMG IDL provides `exception` definitions. Like structs, OMG IDL `exceptions` may have one or more data members of any OMG IDL type. The OMG IDL `module` construct allows for scoping of definition names to prevent name clashes. The OMG IDL type system is described below.

3.2.1 Built-in Types

OMG IDL supports the following *built-in types*:

[5]This theme of "separating components" occurs again later in the discussion of other ORB components such as the Object Adapter.

- `long` (`signed` and `unsigned`) – 32-bit arithmetic types.
- `long long` (`signed` and `unsigned`) – 64-bit arithmetic types.
- `short` (`signed` and `unsigned`) – 16-bit arithmetic types.
- `float`, `double`, and `long double` – IEEE 754-1985 floating point types.
- `char` and `wchar` – character and wide character types
- `boolean` – Boolean type.
- `octet` – 8-bit value.[6]
- `enum` – enumerated type.
- `any` – a tagged type that can hold a value of any OMG IDL type, including built-in types and user-defined types.

The CORBA specification precisely defines the sizes of all the basic types to ensure interoperability across heterogeneous hardware platforms.

3.2.2 Constructed Types

OMG IDL also supports *constructed types*:

- `struct` – data aggregation construct (similar to structs in C/C++).
- discriminated `union` – a type composed of a type discriminator and a value of one of several possible OMG IDL types that are specified in the union definition. OMG IDL `unions` are similar to unions in C/C++, with the addition of the discriminator that keeps track of which alternative is currently valid.

3.2.3 Template Types

In addition, OMG IDL supports *template types* whose exact characteristics are defined at declaration time:

- `string` and `wstring` – string and wide-character string types. Both unbounded `strings`/`wstrings` and bounded `strings`/`wstrings` can be declared. For example, a string with a maximum length of 10 characters requires angle brackets to specify the bound: `string<10>`. An unbounded string, which has no length limit, is simply specified as `string` with no angle brackets or bound numbers.
- `sequence` – a dynamic-length linear container whose maximum length and element type can be specified in angle brackets. For example, `sequence<Factory>` defines an unbounded sequence of `Factory` object references, while `sequence<string,10>` defines a bounded sequence of no more than 10 strings.

[6] An `octet` is guaranteed not to undergo conversions when transmitted over a network by the ORB.

- `fixed` – a fixed-point decimal value with no more than 31 significant digits. For example, `fixed<5,2>` has a precision of 5 digits and a scale of 2, which might be used to represent a monetary value in dollars, up to $999.99, with accuracy to 1 cent.

3.2.4 Object Reference Types

OMG IDL object reference types can simply be declared by naming the desired interface type. For example:

```
// OMG IDL
interface FactoryFinder {

    // define a sequence of Factory
    // object references
    typedef sequence<Factory> FactorySeq;

    FactorySeq find_factories(
                in string interface_name
            );
};
```

This OMG IDL specification defines an interface named `FactoryFinder`[7] that contains the definition of a type named `FactorySeq`. The `FactorySeq` type is defined as an unbounded sequence of `Factory` object references. The `find_factories` operation takes an unbounded `string` type as an input argument and returns an unbounded sequence of `Factory` object references as its result.

3.2.5 Interface Inheritance

An important feature of OMG IDL interfaces is that they can inherit from one or more other interfaces. This makes it possible to reuse existing interfaces when defining new services. For example, given the following OMG IDL specification:

```
// Same as OMG IDL example above
interface Factory {
    Object create();
};

// Forward declaration of Spreadsheet
// interface (full definition not shown)
interface Spreadsheet;

// SpreadsheetFactory derives from Factory
interface SpreadsheetFactory : Factory {
    Spreadsheet create_spreadsheet();
};
```

In this example, the `SpreadsheetFactory` interface inherits from the `Factory` interface, so an object supporting the `SpreadsheetFactory` interface provides two operations:

1. The `create` operation inherited from `Factory`.
2. The `create_spreadsheet` operation defined directly in the `SpreadsheetFactory` interface.

[7] The notion of a "factory finder" comes from the OMG Common Object Services Lifecycle Specification. It is a directory service object for factories that helps applications control the locations at which they create objects. For simplicity, the `FactoryFinder` interface shown here is not the same as the standard interface defined in the Lifecycle Specification.

Interface inheritance is very important in CORBA. It allows the system to be open for extension while keeping it closed for modification, which is called the *Open-Closed Principle* [8, 9]. Since a derived interface inherits all operations defined in all of its base interfaces, objects supporting the derived interface must also support all inherited operations. This allows object references for derived interfaces to be *substituted* anywhere object references for base interfaces are allowed.

For example, a `SpreadsheetFactory` object reference can be used anywhere that a `Factory` object reference is expected because a `SpreadsheetFactory` supports all `Factory` operations. The new capabilities of `SpreadsheetFactory` objects can therefore be added to the system without requiring changes to either existing applications that use the `Factory` interface, or to the `Factory` interface itself.

OMG IDL has one special case of interface inheritance: all interfaces are implicitly derived from the `Object` interface defined in the `CORBA` module. It's as if each interface definition were written as follows:

```
// CORBA::Object is the base interface
// for all interfaces
interface Factory : Object { ... };
```

Since this inheritance from `CORBA::Object`[8] is automatic for every OMG IDL interface, it need not be explicitly declared as shown here.

The OMG IDL type system is sufficient for most distributed applications, yet at the same time it is minimal and will be kept that way. Keeping OMG IDL as simple as possible means that it can be used with many more programming languages (*e.g.*, ranging from COBOL to Java to C++) than it could if it contained types that could not be realized in some popular programming languages. Given the inevitable heterogeneity of distributed object systems, the simplicity of OMG IDL is critical to the success of CORBA as an integration technology.

3.3 Language Mappings

As mentioned above, OMG IDL is just a declarative language, not a full-fledged programming language. As such, it does not provide features like control constructs, nor is it directly used to implement distributed applications. Instead, *language mappings* determine how OMG IDL features are mapped to the facilities of a given programming language.

At the time of this writing, the OMG has standardized language mappings for C, C++, Smalltalk, and Ada 95. Likewise, mappings for the UNIX Bourne shell and for COBOL are nearing completion. A mapping for the Java language is just beginning, but is slated to finish quickly to keep up with the high demand for Java/CORBA integration. Language mappings for other languages such as Perl, Eiffel, and Modula-3 have also been written by various interested parties, but have not been submitted to the OMG for approval.

OMG IDL Type	*C++ Mapping Type*
`long, short`	`long, short`
`float, double`	`float, double`
`enum`	`enum`
`char`	`char`
`boolean`	`bool`
`octet`	`unsigned char`
`any`	`Any` class
`struct`	`struct`
`union`	class
`string`	`char*`
`wstring`	`wchar_t*`
`sequence`	class
`fixed`	`Fixed` template class
object reference	pointer or object
interface	class

Table 1: C++ Mappings for OMG IDL Types

To understand what a language mapping contains, consider the mapping for the C++ language. Not surprisingly, OMG IDL interfaces map to C++ classes, with operations mapping to member functions of those classes. Object references map to objects that support the `operator->` function (*i.e.*, either a normal C++ pointer to an interface class, or an object instance with an overloaded `operator->`). Modules map to C++ `namespaces` (or to nested `classes` for C++ compilers that do not yet support `namespaces`). Mappings for the rest of the OMG IDL types are shown in Table 1.

Another important aspect of an OMG IDL language mapping is how it maps the ORB interface and other *pseudo-objects* that are found in the CORBA specification. Pseudo-objects are ORB interfaces that are not implicitly derived from `CORBA::Object`, such as the ORB itself.[9] In other words, pseudo-objects are not *real* CORBA objects, but specifying such interfaces just like normal object interfaces are specified allows applications to manipulate the ORB much like they manipulate normal objects.

A third important part of any language mapping specification is how CORBA objects are implemented in the language. In object-oriented languages such as Java, Smalltalk, and C++, for example, CORBA objects are implemented as programming language objects. In C, objects are written as abstract data types. For instance, a typical implementation consists of a `struct` that holds the state of the object and a group of C functions (which correspond to the OMG IDL

[8] Within an IDL specification, the keyword `Object` is used to mean `CORBA::Object` – use of the fully-scoped name is not allowed.

[9] For some of the pseudo-objects in the CORBA 2.0 Specification, another differentiating characteristic is that they are defined using non-IDL expressions. Some people consider this a feature, while others consider it a defect in the specification. Because of such disagreements, groups within the OMG are currently working to eliminate pseudo-objects and ensure that all CORBA interfaces are defined in normal OMG IDL.

operations supported by the object) to manipulate that state.

OMG IDL language mappings are where the abstractions and concepts specified in CORBA meet the “real world” of implementation. Thus, their importance for CORBA applications cannot be overstated. A poor or incomplete mapping specification for a given language results in programmers being unable to effectively utilize CORBA technology in that language. Language mapping specifications are therefore always undergoing periodic improvement in order to incorporate evolution of programming languages, as well as to add features that fulfill new requirements discovered by writing new applications.

3.4 Interface Repository

Every CORBA-based application requires access to the OMG IDL type system when it is executing. This is necessary because the application must know the types of values to be passed as request arguments. In addition, the application must know the types of interfaces supported by the objects being used.

Many applications require only *static* knowledge of the OMG IDL type system. Typically, an OMG IDL specification is compiled or translated into code for the application's programming language by following the translation rules for that language, as defined by its language mapping. Then, this generated code is built directly into the application. With this approach, the application's knowledge of the OMG IDL type system is fixed when it is built. If the type system of the rest of the distributed system ever changes in a way that is incompatible with the type system built into the application, the application must be rebuilt. For example, if a client application depends on the `Factory` interface, and the name of the `create` operation in the `Factory` interface is changed to `create_object`, the client application will have to be rebuilt before it can make requests on any `Factory` objects.

There are some applications, however, for which static knowledge of the OMG IDL type system is impractical. For example, consider a *Gateway* that allows applications in a foreign object system (such as Microsoft Component Object Model (COM) applications) to access CORBA objects. Having to recompile and rebuild the Gateway every time someone added a new OMG IDL interface type to the system would result in a very difficult management and maintenance problem. Instead, it would be much better if the Gateway could *dynamically* discover and utilize type information as needed.

The CORBA *Interface Repository* (IR) allows the OMG IDL type system to be accessed and written programmatically at runtime. The IR is itself a CORBA object whose operations can be invoked just like any other CORBA object. Using the IR interface, applications can traverse an entire hierarchy of OMG IDL information. For example, an application can start at the top-level scope of the IR and iterate over all of the `module` definitions defined there. When the desired `module` is found, it can open it and iterate in a similar manner over all the definitions inside it. This hierarchical traversal approach can be used to examine all the information stored within an IR.

Another way to access IR information, perhaps more efficiently, is to obtain an `InterfaceDef` object reference from the `get_interface` operation defined in the `CORBA::Object` interface. Since all interfaces are derived from `CORBA::Object`, every object supports the `get_interface` operation. Thus, an `InterfaceDef` object reference can be obtained for every object without having to know the derived types of interfaces supported by that object.

Since the IR allows applications to programmatically discover type information at runtime, its real utility lies in its support of CORBA dynamic invocation (described in Section 3.6). It can also be used as a source for generating static support code for applications, as described in the next section, since the OMG IDL definitions in the IR are equivalent to those written in an OMG IDL file.

3.5 Stubs and Skeletons

In addition to generating programming language types, OMG IDL language compilers and translators also generate client-side *stubs* and server-side *skeletons*. A stub is a mechanism that effectively creates and issues requests on behalf of a client, while a skeleton is a mechanism that delivers requests to the CORBA object implementation. Since they are translated directly from OMG IDL specifications, stubs and skeletons are normally interface-specific.

Dispatching through stubs and skeletons is often called *static invocation*. OMG IDL stubs and skeletons are built directly into the client application and the object implementation. Therefore, they both have complete *a priori* knowledge of the OMG IDL interfaces of the CORBA objects being invoked.

Language mappings usually map operation invocation to the equivalent of a function call in the programming language. For example, given a `Factory` object reference in C++, the client code to issue a request looks like this:

```
// C++
Factory_var factory_objref;

// Initialize factory_objref using Naming or
// Trading Service (not shown), then issue request
Object_var objref = factory_objref->create();
```

This code makes the invocation of the `create` operation on the target object appear as a regular C++ member function call. However, what this call is really doing is invoking a stub. Because the stub essentially is a stand-in within the local process for the actual (possibly remote) target object, stubs are sometimes called *surrogates* or *proxies*. The stub works directly with the client ORB to *marshal* the request. That is, the stub helps to convert the request from its representation in the programming language to one that is suitable for transmission over the connection to the target object.

Once the request arrives at the target object, the server ORB and the skeleton cooperate to *unmarshal* the request (convert it from its transmissible form to a programming language form) and dispatch it to the object. Once the object completes the request, any response is sent back the way it came: through the skeleton, the server ORB, over the connection, and then back through the client ORB and stub, before finally being returned to the client application. Figure 3 shows the positions of the stub and skeleton in relation to the client application, the ORB, and the object implementation.

This description shows that stubs and skeletons play important roles in connecting the programming language world to the underlying ORB. In this sense they are each a form of the Adapter and Proxy patterns [10]. The stub adapts the function call style of its language mapping to the request invocation mechanism of the ORB. The skeleton adapts the request dispatching mechanism of the ORB to the upcall method form expected by the object implementation.

3.6 Dynamic Invocation and Dispatch

In addition to static invocation via stubs and skeletons, CORBA supports two interfaces for *dynamic invocation*:

- *Dynamic Invocation Interface* (DII) – which supports dynamic client request invocation;
- *Dynamic Skeleton Interface* (DSI) – which provides dynamic dispatch to objects.

The DII and the DSI can be viewed as a *generic stub* and *generic skeleton*, respectively. Each is an interface provided directly by the ORB, and neither is dependent upon the OMG IDL interfaces of the objects being invoked.

3.6.1 Dynamic Invocation Interface

Using the DII, a client application can invoke requests on any object without having compile-time knowledge of the object's interfaces. For example, consider the foreign object Gateway described above. When an invocation is received from the foreign object system, the Gateway must turn that invocation into a request dispatch to the desired CORBA object. Recompiling the Gateway program to include new static stubs every time a new CORBA object is created is impractical. Instead, the Gateway can simply use the DII to invoke requests on any CORBA object. The DII is also useful for interactive programs such as browsers that can obtain the values necessary to supply the arguments for the object's operations from the user.

It is through the `create_request` operation provided by the `CORBA::Object` interface that applications create `Request` pseudo-objects. Since every OMG IDL interface is derived from `CORBA::Object`, every object automatically supports `create_request`. By calling this operation on an object reference for the target object, an application can create a dynamic request for that object. Before the request can be invoked, argument values must be provided for the request by invoking operations directly on the `Request` pseudo-object. The types of the arguments can be determined using the Interface Repository.

Once a `Request` pseudo-object has been created and argument values have been added to it, it can be invoked in one of three ways:

- **Synchronous Invocation** – The client invokes the request, and then blocks waiting for the response. From the client's perspective, this is essentially equivalent in behavior to an RPC. This is the most common invocation mode used for CORBA applications because it is also supported by static stubs.
- **Deferred Synchronous Invocation** – The client invokes the request, continues processing while the request is dispatched, and later collects the response. This is useful if the client has to invoke a number of independent long-running services. Rather than invoking each one serially and blocking for each response, all requests can be issued in parallel, and responses can be collected as they arrive.
- **Oneway Invocation** – The client invokes the request and then continues processing; there is no response. This form is sometimes called "fire and forget" because the only way the client can tell that the request is received is by some other means, *e.g.*, having the object invoke a separate callback request when the first request completes successfully.

Currently, CORBA applications that require the ability to invoke requests using something other than a synchronous or oneway model must use the DII. This is because the deferred synchronous request invocation capability is currently only provided by the DII. However, this restriction will soon be removed. Recently, the OMG issued an RFP for an *Asynchronous Messaging Service* [11] that should result in the adoption of technology for higher-level communications models, such as store-and-forward services for the ORB. This RFP also requests technology for supporting deferred synchronous request invocation via static stubs.

While the DII offers more flexibility than static stubs, users of the DII should also be sure they are aware of its hidden costs [12, 13]. In particular, creating a DII request may cause the ORB to transparently access the IR to obtain information about the types of the arguments and return value. Since the IR is itself a CORBA object, each transparent IR request made by the ORB could in fact be a remote invocation. Thus, the creation and invocation of a single DII request could in fact require several actual remote invocations, making a DII request several times more costly than an equivalent static invocation. Static invocations do not suffer from the overhead of accessing the IR since they rely on type information already compiled into the application.

3.6.2 Dynamic Skeleton Interface

Analogous to the DII is the server-side *Dynamic Skeleton Interface* (DSI). Just as the DII allows clients to invoke requests without having access to static stubs, the DSI allows servers to be written without having skeletons for the objects being invoked compiled statically into the program.

The foreign object Gateway described above is a good example of an application that requires DSI functionality. A bidirectional Gateway must be able to act as both a client and a server – it must translate requests from the foreign object system into requests on CORBA objects, and turn requests from CORBA applications into foreign object invocations. As mentioned above, it can use the DII when it wants to act as a client. To act as a server, however, it needs a server-side equivalent of the DII, allowing it to accept requests without requiring static skeletons for each object's interface type to be compiled into it. Requiring the Gateway to be recompiled each time a new OMG IDL interface was introduced into the CORBA side of the system would not work well in practice.

Unlike most of the other CORBA subcomponents, which were part of the initial CORBA specification, the DSI was only introduced at CORBA 2.0. The main reason for its introduction was to support the implementation of gateways between ORBs utilizing different communications protocols. Even though inter-ORB protocols were also introduced at CORBA 2.0, it was thought by some at the time that gateways would become the method of choice for ORB interoperation. Given that most commercially-available ORB systems already support the standard Internet Inter-ORB Protocol (IIOP) (which is described below in Section 3.8), this prediction does not appear to have come true. Still, the DSI is a useful feature for a certain class of applications, especially for bridges between ORBs and for applications that serve to bridge CORBA systems to non-CORBA services and implementations.

3.7 Object Adapters

The final subcomponent of CORBA, the Object Adapter (OA), serves as the glue between CORBA object implementations and the ORB itself. As described by the Adapter pattern [10], an object adapter is an object that adapts the interface of another object to the interface expected by a caller. In other words, it is an interposed object that uses delegation to allow a caller to invoke requests on an object even though the caller does not know that object's true interface. Figure 4 illustrates the role of an object adapter.

Object adapters represent another aspect of the effort to keep the ORB as simple as possible. Responsibilities of object adapters include:

- *Object registration* – OAs supply operations that allow programming language entities to be registered as implementations for CORBA objects. Details of exactly what is registered and how the registration is accomplished depends on the programming language.
- *Object reference generation* – OAs generate object references for CORBA objects.
- *Server process activation* – If necessary, OAs start up server processes in which objects can be activated.
- *Object activation* – OAs activate objects if they are not already active when requests arrive for them.

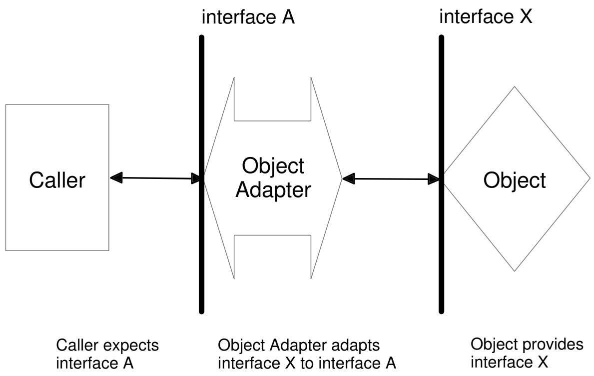

Figure 4: Role of an Object Adapter

- *Request demultiplexing* – OAs must cooperate with the ORB to ensure that requests can be received over multiple connections without blocking indefinitely on any single connection.
- *Object upcalls* – OAs dispatch requests to registered objects.

Without object adapters, the ability of CORBA to support diverse object implementation styles would be severely compromised. The lack of an object adapter would mean that object implementations would connect themselves directly to the ORB to receive requests. Having a standard set of just a few object upcall interfaces would mean that only a few styles of object implementation could ever be supported. Alternatively, standardizing many object upcall interfaces would add unnecessary size and complexity to the ORB itself.

CORBA, therefore, allows for multiple object adapters (as shown in Figure 3). A different object adapter is normally necessary for each different programming language. For example, an object implemented in C would register itself with the object adapter by providing a pointer to a struct holding its state along with a set of function pointers corresponding to the operations defined by its OMG IDL interfaces. Contrast that with a C++ object adapter, which would allow an object implementation to be derived from a standardized object adapter base class that provides the upcall interface. Using the C language object adapter for C++ object implementations or vice-versa would be unnatural to programmers in either language.

Though CORBA states that multiple object adapters are allowed, it currently only provides one: the *Basic Object Adapter* (BOA). When it was first specified, it was hoped that the BOA would suffice for the majority of object implementations, and that other object adapters would only fill niche roles. What the BOA designers failed to realize was that object adapters tend to be very language-specific due to their close proximity to programming language objects. As a result of the goal to make the BOA support multiple languages, the BOA specification had to be made quite vague in certain areas, such as how to register programming language objects as CORBA objects. This in turn has resulted in non-trivial portability problems between BOA implementations because each ORB vendor has filled in the missing pieces with proprietary solutions.

Fortunately, the OMG has recognized this problem and is currently actively working to solve it. It recently issued a *Portability Enhancement RFP* [14] that will result in the adoption of specifications for standard portable object adapters. The OMG should complete its work on the RFP around mid-1997, meaning that portable object adapters should be commercially available by the end of 1997.

3.8 Inter-ORB Protocols

Before CORBA 2.0, one of the biggest complaints about commercial ORB products is that they did not interoperate. Lack of interoperability was caused by the fact that the CORBA specification did not mandate any particular data formats or protocols for ORB communications. The main reason that CORBA did not specify ORB protocols prior to CORBA 2.0 was simply that interoperability was not a focus of the OMG at that time.

CORBA 2.0 introduced a general ORB interoperability architecture that provides for direct ORB-to-ORB interoperability and for bridge-based interoperability. Direct interoperability is possible when two ORBs reside in the same *domain* – in other words, they understand the same object references, the same OMG IDL type system, and perhaps share the same security information. Bridge-based interoperability is necessary when ORBs from separate domains must communicate. The role of the bridge is to map ORB-specific information from one ORB domain to the other.

The general ORB interoperability architecture is based on the *General Inter-ORB Protocol* (GIOP), which specifies transfer syntax and a standard set of message formats for ORB interoperation over any connection-oriented transport. GIOP is designed to be simple and easy to implement while still allowing for reasonable scalability and performance.

The *Internet Inter-ORB Protocol* (IIOP) specifies how GIOP is built over TCP/IP transports. In a way, the relationship between IIOP and GIOP is somewhat like the relationship between an object's OMG IDL interface definition and its implementation. GIOP specifies protocol, just as an OMG IDL interface effectively defines the protocol between an object and its clients. IIOP, on the other hand, determines how GIOP can be implemented using TCP/IP, just as an object implementation determines how an object's interface protocol is realized. For a CORBA 2.0 ORB, support for GIOP and IIOP is mandatory.

The ORB interoperability architecture also provides for other *environment-specific inter-ORB protocols* (ESIOPs). ESIOPs allow ORBs to be built for special situations in which certain distributed computing infrastructures are already in use. The first ESIOP, which utilizes the Distributed Computing Environment (DCE) [15], is called the *DCE Common Inter-ORB Protocol* (DCE-CIOP). It can be used by ORBs in environments where DCE is already installed. This allows the ORB to leverage existing DCE functions, and it allows for easier integration of CORBA and DCE applications. Support for DCE-CIOP or any other ESIOP by a CORBA 2.0 ORB is optional.

In addition to standard interoperability protocols, standard object reference formats are also necessary for ORB interoperability. While object references are opaque to applications, ORBs use the contents of object references to help determine how to direct requests to objects. CORBA specifies a standard object reference format called the *Interoperable Object Reference* (IOR).[10] An IOR stores information needed to locate and communicate with an object over one or more protocols. For example, an IOR containing IIOP information stores hostname and TCP/IP port number information.

Most commercially-available ORB products already support IIOP and IORs and have been tested to ensure interoperability. Interoperability testing is currently done directly between ORB vendors rather than by an independent conformance-testing body. One interesting exception to this rule is an interoperability testbed called *CORBAnet* [16], which was established by the OMG to help facilitate ORB interoperability testing and prove commercial viability. CORBAnet is an interactive meeting room booking system implemented over a number of interoperating commercial ORB products on a variety of hardware platforms. It can be used interactively via a Web browser by accessing `http://corbanet.dstc.edu.au/`.

4 OMG Activities and Future Plans

With over 700 members, the OMG is a very active consortium. Its many task forces and special interest groups cover nearly the entire spectrum of topics related to distributed computing, including real-time computing, Internet, telecommunications, financial systems, medical systems, object analysis and design, electronic commerce, security, database systems, and programming languages. RFPs and technology adoptions in almost all of these areas have either already occured or soon will.

When there were fewer OMG members and CORBA was still under development, most of the OMG's technical activities were focused within its ORB Task Force, which is where the CORBA specification was created. This effectively gave ORB vendors a fair bit of clout when it came to determining the technical direction of the OMG, which tended to keep the technical focus directed at the CORBA component.

In early 1996 the OMG reorganized itself to give users of the CORBA component the power to set their own technical directions. Part of this reorganization involved splitting the OMG Technical Committee into two parts:

[10] Applications use IORs just as they use any other object reference. In OMG IDL terms, there is nothing different about IORs as compared to other object references, *i.e.*, there is no special OMG IDL type for an IOR. Object interfaces are independent of object reference format.

• **Domain Technical Committee (DTC):** Focuses on technologies that are vertically-oriented (*i.e.*, domain-specific). Task Forces chartered under the DTC include Financial, Manufacturing, Medical, Business, and Telecommunications.

• **Platform Technical Committee (PTC):** Focuses on technologies that are horizontally-oriented (*i.e.*, domain-independent). Task Forces chartered under the PTC include ORB/Object Services (ORBOS) and Common Facilities.

This split has resulted in a shift in the OMG focus from the CORBA component to the other higher-level components of the OMA. Such a shift is precisely what should occur as an architecture like the OMA matures. Separating the DTC groups from the domain-independent groups has made it easier for them to issue their own RFPs and adopt suitable domain-specific technology.

To ensure the continued integrity of the OMA even with two technical committees, the OMG also created, as part of the same reorganization, an Architecture Board (AB). The AB, which is composed of ten elected members and a chairperson, has the power to reject RFPs and technologies that do not fit into the OMA. The AB is also charged with finding and defining answers for broad technical issues related to the OMA, such as clarifications of the OMA object model.

Areas that are currently being investigated by OMG task forces include:

• **Medical:** *Master Patient Indexing* – Patient identification can be surprisingly difficult, due to multiple people with the same name, illegal use of identification numbers, etc. At the time of this writing, the *CORBAmed* Medical Task Force was very close to issuing an RFP for technology related to the identification of patients.

• **Telecommunications:** *Isochronous Streams* – Streams for audio and video data have special quality of service requirements due to their isochronous nature. The *CORBAtel* Telecommunications Task Force recently issued an RFP [17] seeking technology for the management and manipulation of isochronous streams.

• **Business:** *Business Objects* – Portions of many business processes are very similar, and thus can be abstracted out into frameworks. The Business Objects Task Force will soon begin evaluating responses to its Business Objects RFP [18], which seeks object frameworks to support business processes.

• **Common Facilities:** *Systems Management Facility* – The OMG has nearly completed the adoption of the X/Open systems management specification [19], which defines a set of extended services for the monitoring and management of distributed systems. These services complement those specified in the existing OMG Common Object Services Lifecycle Specification [4].

• **ORBOS:** *Objects by value* – CORBA currently allows object references to be passed as arguments and return values, but it does not allow objects to be passed by value. This makes the use of encapsulated data types (*e.g.*, linked lists) difficult to use from languages such as C++. The ORBOS Task Force will soon begin evaluating responses to its Objects By Value RFP [20], which will describe technology for passing objects by value between CORBA applications.

Another special area of interest of the ORBOS Task Force is providing specifications that allow for the bidirectional interoperation of Microsoft COM and Distributed COM (DCOM) applications with CORBA applications. A specification for COM/CORBA interoperability has already been approved, while work on DCOM/CORBA interworking has just begun. Contrary to industry rumors, the OMG does not view COM or DCOM as CORBA competitors; rather, it sees them as another set of technologies that can be integrated under the CORBA umbrella.

The end goal of the development of standard OMG specifications is the realization of a true commercial off-the-shelf (COTS) software component marketplace. The OMG will continue working to help create a market in which buying and using software components in distributed heterogeneous environments is a reality. To this end, many OMG member companies have devoted some of their "best and brightest" experts to the OMG to assist with the development of practical, complete, and relevant standards.

The OMG is also working to establish an OMA compliance "branding" program that would prove whether or not an OMA-based product complies properly with the appropriate OMG specifications. Such branding will be necessary in a component marketplace to ensure that OMA-based components interoperate and cooperate correctly.

Of particular importance to the OMG community is a recent press release by Netscape Corporation stating that they will build IIOP and an ORB into future releases of their products, including their Navigator web browser software [21]. They intend to allow remote CORBA objects and services to appear as browser plug-ins, using IIOP to forward requests to them. Because of the popularity of Netscape Navigator, this decision effectively brings CORBA to 40 million desktops around the world. Moreover, it unifies Web technology with distributed object technology, allowing the strengths of each to enhance the other. The deployment of these unified technologies will finally provide the beginnings of a software component marketplace infrastructure.

5 Conclusion

This article has described the Common Object Request Broker Architecture (CORBA) portion of the OMG Object Management Architecture (OMA). CORBA provides a flexible communication and activation substrate for distributed heterogeneous object-oriented computing environments. The strengths of CORBA include:

• **Heterogeneity:** The use of OMG IDL to define object interfaces allows these interfaces to be used from a variety of programming languages and computing platforms. The fact that CORBA systems have already been written in such diverse programming languages as C, C++, Smalltalk, Ada'95, Java, COBOL, Modula-3, Perl, and Python, and successfully deployed across everything from mainframes to test and measurement equipment, is strong evidence that CORBA can be used to implement real-life heterogeneous distributed applications.

• **Object Model:** The Object Model and Reference Model provided by the OMA define the rules for interaction between CORBA objects such that the interactions are independent of underlying network protocols. Unlike typical distributed software systems, which are tied closely to underlying networking protocols and mechanisms, CORBA-based applications are abstracted away from the networking details and thus can be used in a variety of environments.

• **Legacy integration:** Because CORBA does not mandate implementation, a well-designed ORB does not require that components and technologies already in use be abandoned. Instead, the CORBA specification is flexible enough to allow ORBs to incorporate and integrate existing protocols and applications, such as DCE or Microsoft COM, rather than replace them.

• **Object-oriented approach:** CORBA itself and applications built on top of it are best designed using object-oriented (OO) software development principles. For example, the fact that object interfaces must be defined in OMG IDL helps developers think about their applications in terms of interacting reusable components. The management of complexity afforded by OO software development techniques is very important for the practical implementation and deployment of CORBA applications.

Both the Internet and corporate intranets will inevitably remain heterogeneous. Having to deal with the integration of diverse applications, as well as having to manage their associated complexities, are absolute requirements for our ever-expanding networked systems. The on-going work to unify the World Wide Web with CORBA will soon allow the new "universal user interface," the web browser, to cleanly and transparently make use of the varied technologies and legacy systems that exist across today's computing enterprises. With the capabilities and flexibility of CORBA serving to unify the infrastructure, we can focus more on providing solid solutions for higher-level problems and worry less about how to make simple things work in our distributed heterogeneous environments.

Acknowledgements

Thanks to Doug Schmidt for encouraging me to write this article, for his patience while waiting for me to complete it, and for his excellent suggestions on how to improve it. Thanks also to Cindy Buhner, Kevin Currey, Bart Hanlon, Brent Modzelewski, and several anonymous reviewers for their reviews of early drafts of this article.

References

[1] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, pp. 39–59, February 1984.

[2] Object Management Group, *Description of New OMA Reference Model, Draft 1*, OMG Document ab/96-05-02 ed., May 1996.

[3] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[4] Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 ed., Mar. 1995.

[5] Apple Computer, Inc., Component Integration Laboratories, Inc., International Business Machines Corporation, Novell, Incorporated, *Compound Presentation and Compound Interchange Facilities, Part I*, OMG Document 95-12-30 ed., December 1995.

[6] Object Management Group, *Product Data Management Enablers Request For Proposals*, OMG Document mfg/96-08-01 ed., August 1996.

[7] Richard Mark Soley, Ph.D., ed., *Object Management Architecture Guide*. John Wiley & Sons, Inc., Third ed., 1995.

[8] B. Meyer, *Object Oriented Software Construction*. Englewood Cliffs, NJ: Prentice Hall, 1989.

[9] R. C. Martin, "The Open-Closed Principle," *C++ Report*, vol. 8, Jan. 1996.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[11] Object Management Group, *Messaging Service RFP*, OMG Document orbos/96-03-16 ed., March 1996.

[12] S. Vinoski, "Distributed Object Computing with CORBA," *C++ Report*, vol. 5, July/August 1993.

[13] A. Gokhale and D. C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks," in *Proceedings of GLOBECOM '96*, (London, England), pp. 50–56, IEEE, November 1996.

[14] Object Management Group, *ORB Portability Enhancement RFP*, OMG Document 1995/95-06-26 ed., June 1995.

[15] W. Rosenberry, D. Kenney, and G. Fischer, *Understanding DCE*. O'Reilly and Associates, Inc., 1992.

[16] Object Management Group, *OMG Unveils CORBAnet Initiative*, May 13, 1996. Press release. URL: http://www.omg.org/pr96/corbanet.htm.

[17] Object Management Group, *Control and Management of A/V Streams Request For Proposals*, OMG Document telecom/96-08-01 ed., August 1996.

[18] Object Management Group, *Common Business Objects and Business Object Facility RFP*, OMG Document cf/96-01-04 ed., January 1996.

[19] Object Management Group, *Systems Management: Common Management Facilities, Volume 1, Version 2*, OMG Document 1995/95-12-02 through 1995/95-12-06 ed., December 1995.

[20] Object Management Group, *Objects-by-value Request For Proposals*, OMG Document orbos/96-06-14 ed., June 1996.

[21] Netscape Communications Corporation, *New Netscape ONE Platform Brings Distributed Objects To the Internet and Intranets*, July 29, 1996. Press release. URL: http://home.netscape.com/newsref/pr/newsrelease199.html.