http://scottge.wordpress.com

The Component Object Model Specification

Version 0.9
October 24, 1995

This document contains the specification to the Component Object Model (COM), an architecture and supporting infrastructure for building, using, and evolving component software in a robust manner. This specification contains the standard APIs supported by the COM Library, the standard suites of interfaces supported or used by software written in a COM environment, along with the network protocols used by COM in support of distributed computing. This specification is still in draft form, and thus subject to change.

Note: This document is an early release of the final specification. It is meant to specify and accompany software that is still in development. Some of the information in this documentation may be inaccurate or may not be an accurate representation of the functionality of the final specification or software. Microsoft assumes no responsibility for any damages that might occur either directly or indirectly from these inaccuracies. Microsoft may have trademarks, copyrights, patents or pending patent applications, or other intellectual property rights covering subject matter in this document. The furnishing of this document does not give you a license to these trademarks, copyrights, patents, or other intellectual property rights.

The Component Object Model Specification
Draft Version 0.9, October 24, 1995
Microsoft Corporation and Digital Equipment Corporation

Table of Contents

How to Read This Document

This specification is written to help a variety of readers understand the design and implementation of the Component Object Model (referred to herein simply as COM) as much as they would like. The presentation of COM gradually progresses from high-level overviews to COM benefits and eventually into the underlying mechanisms and programming interfaces to COM. This section is intended to help the reader determine what parts of this document to read.

This specification is divided into four parts, each of which contains one or more chapters. Part I is an overview and introduction. Chapter 1, the only chapter in Part I, explains at a high level the motivations of COM and the problems it addresses. It describes what COM is and its features, and describes the major benefits and advantages of COM. All readers should be interested in this chapter.

Part II contains the programming interface to COM, the suite of interfaces and APIs by which Component Object Model software is implemented and used. Chapters 2 through 8 are in Part II.

Chapter 2 goes into more detail about COM features and mechanisms without getting into the details of function call specifications and code. The chapter is intended for technical readers who want to know more than simply what COM is and what problems it solves, and therefore delves deeper into how applications use COM and the benefits of such use.

Chapters 3-6 contain programming-level information for readers who are interested in actually making use of COM in an application. These chapters explain the fundamentals of objects in COM and the creation of object clients as well as object servers. Chapter 3 details the basic object structures and mechanisms and provides the functional specifications of the core of COM. Chapter 4 covers the COM programming interfaces that all applications making use of COM must follow. Chapter 5 then deals specifically with COM clients; Chapter 6 specifically with COM servers.

Chapter 7 contains more detailed information about how COM clients and servers communicate with objects. This information is generally needed only by sophisticated programmers. Nevertheless, programmers may find this chapter enlightening and can gain a clear understanding of all the underlying mechanisms that make COM truly powerful.

Chapter 8 contains information on how communications between COM clients and severs can be made secure.

Part III (Chapters 9-12) provides the functional specifications for the extended features of COM, including storage, naming, and exchange of data. These added features are built on top of the core COM functionality described in the previous chapters.

Part IV specifies standards relating to tools used to assist the authorship of COM software. It includes Chapter 13, which specifies the COM extensions to the standard Interface Definition Language (IDL) of the Open Software Foundation (OSF) Distributed Computing Environment (DCE). This will be of interest primarily to tools vendors who support tools that work with this language. Chapter 14 covers Type Libraries which are the binary equivalent to IDL.

Finally, Part V specifies information needed by programmers who will be implementing COM on other platforms—that is, the programmer who will be implementing COM on a systems level rather than an application level. Within Part V, Chapter 15 specifies the protocol used by COM when performing distributed computing between machines over a network. This chapter heavily references the OSF DCE RPC specification, noted in the Bibliography as [CAE RPC].

This page left intentionally blank.

Part I: Component Object Model Introduction

Part I is an overview and introduction to the Component Object Model. The only chapter in Part I (Chapter 1), explains at a high level the motivations of COM and the problems it addresses. It describes what COM is and its features, and describes the major benefits and advantages of COM.

This page intentionally left blank.

1. Introduction

1 Challenges Facing The Software Industry

Constant innovation in computing hardware and software have brought a multitude of powerful and sophisticated applications to users' desktops and across their networks. Yet with such sophistication have come commensurate problems for application developers, software vendors, and users:

- Today's applications are large and complex—they are time-consuming to develop, difficult and costly to maintain, and risky to extend with additional functionality.
- Applications are monolithic—they come prepackaged with a wide range of features but most features cannot be removed, upgraded independently, or replaced with alternatives.
- Applications are not easily integrated—data and functionality of one application are not readily available to other applications, even if the applications are written in the same programming language and running on the same machine.
- Operating systems have a related set of problems. They are not sufficiently modular, and it is difficult to override, upgrade, or replace OS-provided services in a clean and flexible fashion.
- Programming models are inconsistent for no good reason. Even when applications have a facility for cooperating, their services are provided to other applications in a different fashion from the services provided by the operating system or the network. Moreover, programming models vary widely depending on whether the service is coming from a provider in the same address space as the client program (via dynamic linking), from a separate process on the same machine, from the operating system, or from a provider running on a separate machine (or set of cooperating machines) across the network.

In addition, a result of the trends of hardware down-sizing and increasing software complexity is the need for a new style of distributed, client/server, modular and "componentized" computing. This style calls for:

- A generic set of facilities for finding and using service providers (whether provided by the operating system or by applications, or a combination of both), for negotiating capabilities with service providers, and for extending and evolving service providers in a fashion that does not inadvertently break the consumers of earlier versions of those services.
- Use of object-oriented concepts in system and application service architectures to better match the new generation of object-oriented development tools, to manage increasing software complexity through increased modularity, to re-use existing solutions, and to facilitate new designs of more self-sufficient software components.
- Client/server computing to take advantage of, and communicate between, increasingly powerful desktop devices, network servers, and legacy systems.
- Distributed computing to provide a single system image to users and applications and to permit use of services in a networked environment regardless of location, machine architecture, or implementation environment.

As an illustration of the issues at hand, consider the problem of creating a system service API (Application Programming Interface) that works with multiple providers of some service in a "polymorphic" fashion. That is, a client of the service can transparently use any particular provider of the service without any special knowledge of which specific provider —or implementation —is in use. In traditional systems, there is a central piece of code—conceptually, the service manager is a sort of "object manager," although traditional systems usually involve function-call programming models with system-provided handles used as the means for "object" selection—that every application calls to access meta-operations such as selecting an object and connecting to it. But once applications have used those "object manager" operations and are connected to a service provider, the "object manager" only gets in the way and forces unnecessary overhead upon all applications as shown in Figure 1-1.

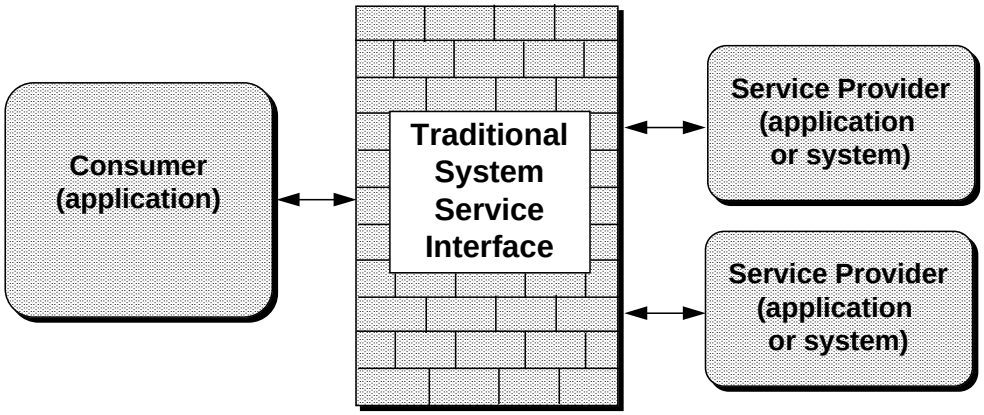


Figure 1-1: Traditional system service APIs require all applications to communicate through a central manager with corresponding overhead.

In addition to the overhead of the system-provided layer, another significant problem with traditional service models is that it is impossible for the provider to express new, enhanced, or unique capabilities to potential consumers in a standard fashion. A well-designed traditional service architecture may provide the notion of different levels of service. (Microsoft's Open Database Connectivity (ODBC) API is an example of such an API.) Applications can count on the minimum level of service, and can determine at run-time if the provider supports higher levels of service in certain pre-defined quanta, but the providers are restricted to providing the levels of services defined at the outset by the API; they cannot readily provide a new capability and then evangelize consumers to access it cheaply and in a fashion that fits within the standard model. To take the ODBC example, the vendor of a database provider intent on doing more than the current ODBC standard permits must convince Microsoft to revise the ODBC standard in a way that exposes that vendor's extra capabilities. Thus, traditional service architectures cannot be readily extended or supplemented in a decentralized fashion.

Traditional service architectures also tend to be limited in their ability to robustly evolve as services are revised and versioned. The problem with versioning is one of representing capabilities (what a piece of code can do) and identity (what a piece of code *is*) in an interrelated, fuzzy way. A later version of some piece of code, such as "Code version 2" indicates that it is *like* "Code version 1" but different in some way. The problem with traditional versioning in this manner is that it's difficult for code to indicate *exactly how* it differs from a previous version and worse yet, for clients of that code to react appropriately to new versions—or to not react at all if they expect only the previous version. The versioning problem can be reasonably managed in a traditional system when (i) there is only a single provider of a certain kind of service, (ii) the version number of the service is checked by the consumer when it binds to the service, (iii) the service is extended only in an upward-compatible manner—*i.e.*, features can only be added and never removed (a significant restriction as software evolves over a long period of time)—so that a version N provider will work with consumers of versions 1 through N-1 as well, and (iv) references to a running instance of the service are not freely passed around by consumers to other consumers, all of which may expect or require different versions. But these kind of restrictions are obviously unacceptable in a multi-vendor, distributed, modular system with polymorphic service providers.

These problems of service management, extensibility, and versioning have fed the problems stated earlier. Application complexity continues to increase as it becomes more and more difficult to extend functionality. Monolithic applications are popular because it is safer and easier to collect all interdependent services and the code that uses those services into one package. Interoperability between applications suffers accordingly, where monolithic applications are loathe to allow independent agents to access their functionality and thus build a dependence upon a certain behavior of the application. Because end users demand interoperability, however, application are compelled to attempt interoperability, but this leads directly back to the problem of application complexity, completing a circle of problems that limit the progress of software development.

2 The Solution: Component Software

Object-oriented programming has long been advanced as a solution to the problems at hand. However, while object-oriented programming is powerful, it has yet to reach its full potential because no standard framework exists through which software objects created by different vendors can interact with one another within the same address space, much less across address spaces, and across network and machine architecture boundaries. The major result of the object-oriented programming revolution has been the production of "islands of objects" that can't talk to one another across the sea of application boundaries in a meaningful way.

The solution is a system in which application developers create reusable *software components*. A component is a reusable piece of software in binary form that can be plugged into other components from other vendors with relatively little effort. For example, a component might be a spelling checker sold by one vendor that can be plugged into several different word processing applications from multiple vendors. It might be a math engine optimized for computing fractals. Or it might be a specialized transaction monitor that can control the interaction of a number of other components (including service providers beyond traditional database servers). Software components must adhere to a binary external standard, but their internal implementation is completely unconstrained. They can be built using procedural languages as well as object-oriented languages and frameworks, although the latter provide many advantages in the component software world.

Software component objects are much like integrated circuit (IC) components, and component software is the integrated circuit of tomorrow. The software industry today is very much where the hardware industry was 20 years ago. At that time, vendors learned how to shrink transistors and put them into a package so that no one ever had to figure out how to build a particular discrete function—an NAND gate for example—ever again. Such functions were made into an integrated circuit, a neat package that designers could conveniently buy and design around. As the hardware functions got more complex, the ICs were integrated to make a board of chips to provide more complex functionality and increased capability. As integrated circuits got smaller yet provided more functionality, boards of chips became just bigger chips. So hardware technology now uses chips to build even bigger chips.

The software industry is at a point now where software developers have been busy building the software equivalent of discrete transistors—software routines—for a long time.

The Component Object Model enables software suppliers to package their functions into reusable software components in a fashion similar to the integrated circuit. What COM and its objects do is bring software into the world where an application developer no longer has to write a sorting algorithm, for example. A sorting algorithm can be packaged as a binary object and shipped into a marketplace of component objects. The developer who need a sorting algorithm just uses any sorting object of the required type without worrying about how the sort is implemented. The developer of the sorting object can avoid the hassles and intellectual property concerns of source-code licensing, and devote total energy to providing the best possible binary version of the sorting algorithm. Moreover, the developer can take advantage of COM's ability to provide easy extensibility and innovation beyond standard services as well as robust support for versioning of components, so that a new component works perfectly with software clients expecting to use a previous version.

As with hardware developers and the integrated circuit, applications developers now do not have to worry about *how* to build that function; they can simply purchase that function. The situation is much the same as when you buy an integrated circuit today: You don't buy the sources to the IC and rebuild the IC yourself. COM allows you to simply buy the software component, just as you would buy an integrated circuit. The component is compatible with anything you "plug" it into.

By enabling the development of component software, COM provides a much more productive way to design, build, sell, use, and reuse software. Component software has significant implications for software vendors, users, and corporations:

- **Application developers** are enabled to build and distribute applications more easily than ever before. Component objects provide both scalability from single processes to enterprise networks and modularity for code reuse. In addition, developers can attain higher productivity because they can learn one object system for many platforms.

- **Vendors** are provided with a single model for interacting with other applications and the distributed computing environment. While component software can readily be added to existing applications without fundamental rewriting, it also provides the opportunity to modularize applications and to incrementally replace system capabilities where appropriate. The advent of component software will help create more diverse market segments and niches for small, medium, and large vendors.
- **End-users** will see a much greater range of software choices, coupled with better productivity. Users will have access to hundreds of objects across client and server platforms—objects that were previously developed by independent software vendors (ISVs) and corporations. In addition, as users see the possibilities of component software, demand is likely to increase for specialized components they can purchase at a local software retail outlet and plug into applications.
- **Corporations** benefit from lower costs for corporate computing, helping IS departments work more efficiently, and enabling corporate computer users to be more productive. IS developers will spend less time developing general purpose software components and more time developing "glue" components to create business-specific solutions. Existing applications do not need to be rewritten to take advantage of a component architecture. Instead, corporate developers can create object-based "wrappers" that encapsulate the legacy application and make its operations and data available as an object to other software components in the network.

3 The Component Software Solution: OLE's COM

The Component Object Model provides a means to address problems of application complexity and evolution of functionality over time. It is a widely available, powerful mechanism for customers to adopt and adapt to a new style multi-vendor distributed computing, while minimizing new software investment.. COM is an open standard, fully and completely publicly documented from the lowest levels of its protocols to the highest. As a robust, efficient and workable component architecture it has been proven in the marketplace as the foundation of diverse and several application areas including compound documents, programming widgets, 3D engineering graphics, stock market data transfer, high performance transaction processing, and so on.

The Component Object Model is an object-based programming model designed to promote software interoperability; that is, to allow two or more applications or "components" to easily cooperate with one another, even if they were written by different vendors at different times, in different programming languages, or if they are running on different machines running different operating systems. To support its interoperability features, COM defines and implements mechanisms that allow applications to connect to each other as *software objects.* A software object is a collection of related function (or intelligence) and the function's (or intelligence's) associated state.

In other words, COM, like a traditional system service API, provides the operations through which a client of some service can connect to multiple providers of that service in a polymorphic fashion. But once a connection is established, *COM drops out of the picture.* COM serves to connect a client and an object, but once that connection is established, the client and object communicate directly without having to suffer overhead of being forced through a central piece of API code as illustrated in Figure 1-2.

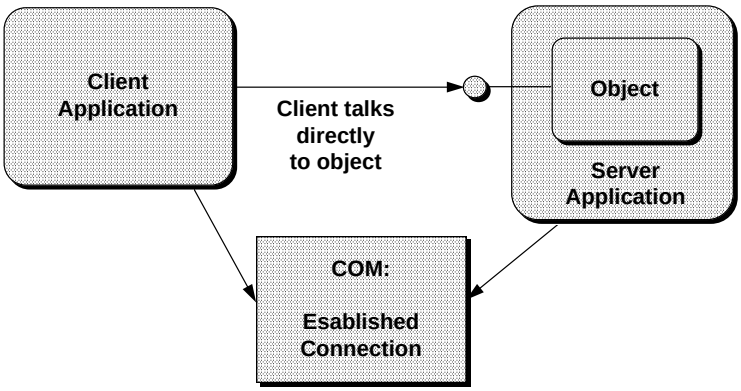


Figure 1-2: Once COM connects client and object, the client and object communicate directly without added overhead.

COM is not a prescribed way to structure an application; rather, it is a set of technologies for building robust groups of services in both systems and applications such that the services and the clients of those ser-

vices can evolve over time. In this way, COM is a technology that makes the programming, use, and uncoordinated/independent evolution of binary objects *possible*. COM is not a technology designed primarily for making programming necessarily *easy*; indeed, some of the difficult requirements that COM accepts and meets necessarily involve some degree of complexity.[1] However, COM provides a ready base for extensions oriented towards increased ease-of-use, as well as a great basis for powerful, easy development environments, language-specific improvements to provide better language integration, and pre-packaged functionality within the context of application frameworks.

This is a fundamental strength of COM over other proposed object models: COM solves the "deployment problem," the versioning/evolution problem where it is necessary that the functionality of objects can incrementally evolve or change without the need to simultaneously and in lockstep evolve or change all existing the clients of the object. Objects/services can easily continue to support the interfaces through which they communicated with older clients as well as provide new and better interfaces through which they communicate with newer clients.

To solve the versioning problems as well providing connection services without undue overhead, the Component Object Model builds a foundation that:

- Enables the creation and use of reusable components by making them "component objects."
- Defines a binary standard for interoperability.
- Is a true system object model.
- Provides distributed capabilities.

The following sections describe each of these points in more detail.

.1 Reusable Component Objects

Object-oriented programming allows programmers to build flexible and powerful software objects that can easily be reused by other programmers. Why is this? What is it about objects that are so flexible and powerful?

The definition of an object is a piece of software that contains the functions that represent what the object can do (its intelligence) and associated state information for those functions (data). An object is, in other words, some data structure and some functions to manipulate that structure.

An important principle of object-oriented programming is *encapsulation*, where the exact implementation of those functions and the exact format and layout of the data is only of concern to the object itself. This information is hidden from the clients of an object. Those clients are interested only in an object's behavior and not the object's internals. For instance, consider an object that represents a stack: a user of the stack cares only that the object supports "push" and "pop" operations, not whether the stack is implemented with an array or a linked list. Put another way, a client of an object is interested only in the "contract"—the promised behavior—that the object supports, not the implementation it uses to fulfill that contract.

COM goes as far as to formalize the notion of a contract between object and client. Such a contract is the basis for interoperability, and for interoperability to work on a large scale requires a strong standard.

.2 Binary and Wire-Level Standards for Interoperability

The Component Object Model defines a completely standardized mechanism for creating objects and for clients and objects to communicate. Unlike traditional object-oriented programming environments, these mechanisms are independent of the applications that use object services and of the programming languages used to create the objects. The mechanisms also support object invocations across the network. COM therefore defines a *binary interoperability standard* rather than a language-based interoperability standard on any given operating system and hardware platform. In the domain of network computing, COM defines a standard architecture-independent wire format and protocol for interaction between objects on heterogeneous platforms.

[1] "Easy" is a relative term: without COM, some sorts of programming are simply not *possible* and thus the term "easy" is utterly empty.

.1 Why Is Providing a Binary and Network Standard Important?

By providing a binary and network standard, COM enables interoperability among applications that different programmers from different companies write. For example, a word processor application from one vendor can connect to a spreadsheet object from another vendor and import cell data from that spreadsheet into a table in the document. The spreadsheet object in turn may have a "hot" link to data provided by a data object residing on a mainframe. As long as the objects support a predefined standard interface for data exchange, the word processor, spreadsheet, and mainframe database don't have to know anything about each other's implementation. The word processor need only know how to connect to the spreadsheet; the spreadsheet need only know how to expose its services to anyone who wishes to connect. The same goes for the network contract between the spreadsheet and the mainframe database. All that either side of a connection needs to know are the standard mechanisms of the Component Object Model.

Without a binary and network standard for communication and a standard set of communication interfaces, programmers face the daunting task of writing a large number of procedures, each of which is specialized for communicating with a different type of object or client, or perhaps recompiling their code depending on the other components or network services with which they need to interact. With a binary and network standard, objects and their clients need no special code and no recompilation for interoperability. But these standards must be efficient for use in both a single address space and a distributed environment; if the mechanism used for object interaction is not extremely efficient, especially in the case of local (same machine) servers and components within a single address space, mass-market software developers pressured by size and performance requirements simply will not use it.

Finally, object communication must be programming language-independent since programmers cannot and should not be forced to use a particular language to interact with the system and other applications. An illustrative problem is that every C++ vendor says, "We've got class libraries and you can use our class libraries." But the interfaces published for that one vendor's C++ object usually differs from the interfaces publishes for another vendor's C++ object. To allow application developers to use the objects' capabilities, each vendor has to ship the source code for the class library for the objects so that application developers can rebuild that code for the vendor's compiler they're using. By providing a binary standard to which objects conform, vendors do not have to send source code to provide compatibility, nor to users have to restrict the language they use to get access to the objects' capabilities. COM objects are compatible by nature.

.2 COM's Standards Enable Object Interoperability

With COM, applications interact with each other and with the system through collections of function calls—also known as methods or member functions or requests—called *interfaces*. An "interface" in the COM sense[2] is a strongly typed *contract* between software components to provide a relatively small but useful set of semantically related operations. An interface is an articulation of an expected behavior and expected responsibilities, and the semantic relation of interfaces gives programmers and designers a concrete entity to use when referring to the contract. Although not a strict requirement of the model, interfaces should be factored in such fashion that they can be re-used in a variety of contexts. For example, a simple interface for generically reading and writing streams of data can be re-used by many different types of objects and clients.

The use of such interfaces in COM provides four major benefits:

1. **The ability for functionality in applications (clients or servers of objects) to evolve over time:** This is accomplished through a request called *QueryInterface* that all COM objects support (or else they are not COM objects). *QueryInterface* allows an object to make more interfaces (that is, new groups of functions) available to new clients while at the same time retaining complete binary compatibility with existing client code. In other words, revising an object by adding new, even unrelated functionality will not require any recompilation on the part of any existing clients. Because COM allows objects to have multiple interfaces, an object can express any number of "versions" simultaneously, each of which may be in simultaneous use by clients of different vintage.

[2] The term "interface" is used in a very similar sense in the Component Object Request Broker Architecture (CORBA) design of the Object Management Group. In both cases the idea of an "interface" is a signature of functions and, implicitly, capabilities, entirely abstracted from the implementation. The major difference between COM and CORBA at this high level is that CORBA objects have one and only one interface while COM objects can have many interfaces simultaneously. DCE RPC (from OSF) uses the term "interface" in a similar manner.

And when its clients pass around a reference to the "object," an occurrence that in principle cannot be known and therefore "guarded against" by the object, they actually pass a reference to a particular *interface* on the object, thus extending the chain of backward compatibility. The use of immutable interfaces and multiple interfaces per object solves the problem of versioning.

2. **Very fast and simple object interaction for same-process objects:** Once a client establishes a connection to an object, calls to that object's services (interface functions) are simply indirect functions calls through two memory pointers. As a result, the performance overhead of interacting with an in-process COM object (an object that is in the same address space) as the calling code is negligible—only a handful of processor instructions slower than a standard direct function call and no slower than a compile-time bound C++ single-inheritance object invocation.[3]
3. **"Location transparency":** The binary standard allows COM to intercept a interface call to an object and make instead a *remote procedure call* (RPC) to the "real" instance of the object that is running in another process or on another machine. A key point is that the caller makes this call exactly as it would for an object in the same process. Its binary and network standards enables COM to perform inter-process and cross-network function calls transparently. While there is, of course, a great deal more overhead in making a remote procedure call, no special code is necessary in the client to differentiate an in-process object from out-of-process objects. All objects are available to clients in a uniform, transparent fashion.[4]

This is all well and good. But in the real world, it is sometimes necessary for performance reasons that special considerations be taken into account when designing systems for network operation that need not be considered when only local operation is used. What is needed is not pure local / remote transparency, but "local / remote transparency, unless you need to care." COM provides this capability. An object implementor can if he wishes support *custom marshaling* which allows his objects to take special action when they are used from across the network, different action if he would like than is used in the local case. The key point is that this is done completely transparently to the client. Taken as a whole, this architecture allows one to design client / object interfaces at their natural and easy semantic level without regard to network performance issues, then at a later address network performance issues without disrupting the established design.

4. **Programming language independence:** Because COM is a binary standard, objects can be implemented in a number of different programming languages and used from clients that are written using completely different programming languages. Any programming language that can create structures of pointers and explicitly or implicitly call functions through pointers—languages such as C, C++, Pascal, Ada, Smalltalk, and even BASIC programming environments—can create and use COM objects immediately. Other languages can easily be enhanced to support this requirement.

In sum, only with a binary standard can an object model provide the type of structure necessary for full interoperability, evolution, and re-use between any application or component supplied by any vendor on a single machine architecture. Only with an architecture-independent network wire protocol standard can an object model provide full interoperability, evolution, and re-use between any application or component supplied by any vendor in a network of heterogeneous computers. With its binary and networking standards, COM opens the doors for a revolution in software innovation without a revolution in networking, hardware, or programming and programming tools.

---

[3] Indeed, in principle the intrinsic method dispatch overhead of COM is in fact *less* than the intrinsic overhead of C++ multiple inheritance method invocations. In a multiple inheritance situation, C++ must on every method invocation adjust the this pointer to be as appropriate for the actual method which is to be executed. In an COM object which supports multiple interfaces, which is directly analogous to the multiple inheritance situation, one must of course also do a similar sort of adjustment, and this is done in the QueryInterface method. However, when using a given interface on the object, one can invoke QueryInterface once and use the returned pointer many times. Thus, the cost of the QueryInterface operation can be amortized over all the subsequent usage, resulting in less overall dispatch overhead. Be aware, however, that this distinction is completely academic. In almost all real word situations, both dispatch mechanisms provide more than adequate performance.

[4] There can be subtle differences in the flow-of-control between calling in-process and out-of-process objects. In particular, an out-of-process object call may result in a call-back prior to the completion of the original call. COM provides standard mechanisms to deal with call-backs and reentrancy; even on single-threaded operating systems. Without such standards, true interoperability between out-of-process objects (of which cross-network objects is just a typical case) is impossible.

.3 A True System Object Model

To be a true system model, an object architecture must allow a distributed, evolving system to support millions of objects without risk of erroneous connections of objects and other problems related to strong typing or definition. COM is such an architecture. In addition to being an object-based service architecture, COM is a true system object model because it:

- Uses "globally unique identifiers" to identify object classes and the interfaces those objects may support.
- Provides methods for code reusability without the problems of traditional language-style implementation inheritance.
- Has a single programming model for in-process, cross-process, and cross-network interaction of software components.
- Encapsulates the life-cycle of objects via reference counting.
- Provides a flexible foundation for security at the object level.

The following sections elaborate on each of these aspects of COM.

.1 Globally Unique Identifiers

Distributed object systems have potentially millions of interfaces and software components that need to be uniquely identified. Any system that uses human-readable names for finding and binding to modules, objects, classes, or requests is at risk because the probability of a collision between human-readable names is nearly 100% in a complex system. The result of name-based identification will inevitably be the accidental connection of two or more software components that were not designed to interact with each other, and a resulting error or crash—even though the components and system had no bugs and worked as designed.

By contrast, COM uses globally unique identifiers (GUIDs)—128-bit integers that are virtually guaranteed to be unique in the world across space and time—to identify every interface and every object class and type.[5] These globally unique identifiers are the same as UUIDs (Universally Unique IDs) as defined by DCE. Human-readable names are assigned only for convenience and are locally scoped. This helps insure that COM components do not accidentally connect to an object or via an interface or method, even in networks with millions of objects.[6]

.2 Code Reusability and Implementation Inheritance

Implementation inheritance—the ability of one component to "subclass" or "inherit" some of its functionality from another component while "over-riding" other functions—is a very useful technology for building applications. But more and more experts are concluding that it creates serious problems in a loosely coupled, decentralized, evolving object system. The problem is technically known as the lack of type-safety in the specialization interface and is well-documented in the research literature.[7]

The general problem with traditional implementation inheritance is that the "contract" or interface between objects in an implementation hierarchy is not clearly defined; indeed, it is implicit and ambiguous. When the parent or child component changes its implementation, the behavior of related components may become undefined. This tight coupling of implementations is not a problem when the implementation hierarchy is under the control of a defined group of programmers who can, if necessary, make updates to all components simultaneously. But it is precisely this ability to control and change a set of related components simultaneously that differentiates an application, even a complex application, from a true distributed object

---

[5] Although "class" and "type" can often be used interchangeably, in COM a "type" is the total signature of an object, which is the union of the interfaces that the object supports. "Class" is a particular implementation of a type, and can include certain unique implementation-specific attributes such as product name, icon, etc. For example, the "chart" type (identified by a GUID by whomever first defines that particular combination of interfaces) might be supported by Lotus 1-2-3 for Windows and Microsoft Excel for the Macintosh, each of which are separate classes. Normally, types are polymorphic; any consumer of the services provided by interfaces making up the type can use any class that implements the type.

[6] As an illustration of how unique GUIDs are consider that one could generate 10 million GUIDs a second until the year 5770 AD and each one would be unique.

[7] See, for example, Richard Helm (Senior Researcher, IBM Thomas J. Watson Research Center), *Ensuring Semantic Integrity of Reusable Objects (Panel)*, OOPSLA '92 Conference Proceedings, p.300; John Lamping (Xerox PARC), *Typing the Specialization Interface*, OOPSLA '93 Conference Proceedings, p.201.

system. So while traditional implementation inheritance can be a very good thing for building applications and components, it is inappropriate in a system object model.

Today, COM provides two mechanisms for code reuse called *containment/delegation* and *aggregation*. In the first and more common mechanism, one object (the "outer" object) simply becomes the client of another, internally using the second object (the "inner" object) as a provider of services that the outer object finds useful in its own implementation. For example, the outer object may implement only stub functions that merely pass through calls to the inner object, only transforming object reference parameters from the inner object to itself in order to maintain full encapsulation. This is really no different than an application calling functions in an operating system to achieve the same ends—other objects simply extend the functionality of the system. Viewed externally, clients of the outer object only ever see the outer object—the inner "contained" object is completely hidden—encapsulated—from view. And since the outer object is itself a client of the inner object, it always uses that inner object through a clearly defined contracts: the inner object's interfaces. By implementing those interfaces, the inner object signs the contract promising that it will not change its behavior unexpectedly.

With *aggregation,* the second and more rare reuse mechanism, COM objects take advantage of the fact that they can support multiple interfaces. An aggregated object is essentially a composite object in which the outer object exposes an interface from the inner object directly to clients as if it were part of the outer object. Again, clients of the outer object are impervious to this fact, but internally, the outer object need not implement the exposed interface at all. The outer object has determined that the implementation of the inner object's interface is exactly what it wants to provide itself, and can reuse that implementation accordingly. But the outer object is still a client of the inner object and there is still a clear contract between the inner object and any client. Aggregation is really nothing more than a special case of containment/delegation to prevent the outer object from having to implement an interface that does nothing more than delegate every function to the same interface in the inner object. Aggregation is really a performance convenience more than the primary method of reuse in COM.

Both these reuse mechanisms allow objects to exploit existing implementation while avoiding the problems of traditional implementation inheritance. However, they lack a powerful, if dangerous, capability of traditional implementation inheritance: the ability of a child object to "hook" calls that a parent object might make on itself and override entirely or supplement partially the parent's behavior. This feature of implementation inheritance is definitely useful, but it is also the key area where imprecision of interface and implicit coupling of *implementation* (as opposed to interface) creeps in to traditional implementation inheritance mechanisms. A future challenge for COM is to define a set of conventions that components can use to provide this "hooking" feature of implementation inheritance while maintaining the strictness of contract between objects and the full encapsulation required by a true system object model, even those in "parent/child" relationships.[8]

.3 Single Programming Model

A problem related to implementation inheritance is the issue of a single programming model for in-process objects and out-of-process/cross-network objects. In the former case, class library technology (or application frameworks) permits only the use of features or objects that are in a single address. Such technology is far from permitting use of code outside the process space let alone code running on another machine altogether. In other words, a programmer can't subclass a remote object to reuse its implementation. Similarly, features like public data items in classes that can be freely manipulated by other objects within a single address space don't work across process or network boundaries. In contrast, COM has a single interface-based binding model and has been carefully designed to minimize differences between the in-process and out-of-process programming model. Any client can work with any object anywhere else on the machine or network, and because the object reusability mechanisms of containment and aggregation maintain

[8] Readers interested in this issue should examine the "connectable object" architecture described in Chapter 11. Connectable objects enable an event model that provides a standard, powerful convention for a COM object to signal to any interested client that is about to do something, that is doing something, and that it is finished doing something. The model also allows clients to cancel the event outright or to cancel it in favor of an "overriding" event supplied by the client. This event model coupled with a few additional conventions could provide COM with all the traditional features of implementation inheritance and more without the traditional risks. For an interesting discussion of the problems of traditional implementation inheritance as well as a description of how an inheritance system might be provide robust type-safety, see Hauck, *Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance*, OOPSLA '93 Conference Proceedings, p.231.

a client/server relationship between objects, reusability is also possible across process and network boundaries.

.4 Life-cycle Encapsulation

In traditional object systems, the life-cycle of objects—the issues surrounding the creation and deletion of objects—is handled implicitly by the language (or the language runtime) or explicitly by application programmers. In other words, an object-based application, there is always someone (a programmer or team of programmers) or something (for example, the startup and shutdown code of a language runtime) that has complete knowledge when objects must be created and when they should be deleted.

But in an evolving, decentralized *system* made up of objects, it is no longer true that someone or something always "knows" how to deal with object life-cycle. Object creation is still relatively easy; assuming the client has the right security privileges, an object is created whenever a client requests that it be created. But object deletion is another matter entirely. How is it possible to "know" *a priori* when an object is no longer needed and should be deleted? Even when the original client is done with the object, it can't simply shut the object down since it is likely to have passed a reference to the object to some other client in the system, and how can it know if/when that client is done with the object?—or if that second client has passed a reference to a third client of the object, and so on.

At first, it may seem that there are other ways of dealing with this problem. In the case of cross-process and cross-network object usage, it might be possible to rely on the underlying communication channel to inform the system when all *connections* to an object have disappeared. The object can then be safely deleted. There are two drawbacks to this approach, however, one of which is fatal. The first and less significant drawback is that it simply pushes the problem out to the next level of software. The object system will need to rely on a connection-oriented communications model that is capable of tracking object connections and taking action when they disappear. That might, however, be an acceptable trade-off.

But the second drawback is flatly unacceptable: this approach requires a major difference between the cross-process/cross-network programming model, where the communication system can provide the hook necessary for life-cycle management, and the single-process programming model where objects are directly connected together without any intervening communications channel. In the latter case, object life-cycle issues must be handled in some other fashion. This lack of location transparency would mean a difference in the programming model for single-process and cross-process objects. It would also force clients to make a once-for-all compile-time decision about whether objects were going to run in-process or out-of-process instead of allowing that decision to be made by *users* of the binary component on a flexible, ad hoc basis. Finally, it would eliminate the powerful possibility of composite objects or aggregates made up of both in-process and out-of-process objects.

Could the issue simply be ignored? In other words, could we simply ignore garbage collection (deletion of unused objects) and allow the operating system to clean up unneeded resources when the process was eventually torn down? That non-"solution" might be tempting in a system with just a few objects, or in a system (like a laptop computer) that comes up and down frequently. It is totally unacceptable, however, in the case of an environment where a single process might be made up of potentially thousands of objects or in a large server machine that must never stop. In either case, lack of life-cycle management is essentially an embrace of an inherently unstable system due to memory leaks from objects that never die.

There is only one solution to this set of problems, the solution embraced by COM: clients must tell an object when they are using it and when they are done, and objects must delete themselves when they are no longer needed. This approach, based on reference counting by all objects, is summarized by the phrase "life-cycle encapsulation" since objects are truly encapsulated and self-reliant if and only if they are responsible, with the appropriate help of their clients acting singly and not collectively, for deleting themselves.

Reference counting is admittedly complex for the new COM programmer; arguably, it is the most difficult aspect of the COM programming model to understand and to get right when building complex peer-to-peer COM applications. When viewed in light of the non-alternatives, however, its inevitability for a true system object model with full location transparency is apparent. Moreover, reference counting is precisely the kind of mechanical programming task that can be automated to a large degree or even entirely by well-designed programming tools and application frameworks. Tools and frameworks focused on building COM components exist today and will proliferate increasingly over the next few years. Moreover, the COM model itself

may evolve to provide support for optionally delegating life-cycle management to the system. Perhaps most importantly, reference counting in particular and native COM programming in general involves the kind of mind-shift for programmers—as in GUI event-driven programming just a few short years ago—that seems difficult at first, but becomes increasingly easy, then second-nature, then almost trivial as experience grows.

.5 Security

For a distributed object system to be useful in the real world it must provide a means for secure access to objects and the data they encapsulate. The issues surrounding system object models are complex for corporate customers and ISVs making planning decisions in this area, but COM meets the challenges, and is a solid foundation for an enterprise-wide computing environment.

COM provides security along several crucial dimensions. First, COM uses standard operating system permissions to determine whether a client (running in a particular user's security context) has the right to start the code associated with a particular class of object. Second, with respect to persistent objects (class code along with data stored in a persistent store such as file system or database), COM uses operating system or application permissions to determine if a particular client can load the object at all, and if so whether they have read-only or read-write access, etc. Finally, because its security architecture is based the design of the DCE RPC security architecture, an industry-standard communications mechanism that includes fully authenticated sessions, COM provides cross-process and cross-network object servers with standard security information about the client or clients that are using it so that a server can use security in more sophisticated fashion than that of simple OS permissions on code execution and read/write access to persistent data.

.4 Distributed Capabilities

COM supports *distributed objects*; that is, it allows application developers to split a single application into a number of different component objects, each of which can run on a different computer. Since COM provides network transparency, these applications do not appear to be located on different machines. The entire network appears to be one large computer with enormous processing power and capacity.

Many single-process object models and programming languages exist today and a few distributed object systems are available. However, none provides an identical, transparent programming model for small, in-process objects, medium out-of-process objects on the same machine, and potentially huge objects running on another machine on the network. The Component Object Model provides just such a transparent model, where a client uses an object in the same process in precisely the same manner as it would use one on a machine thousands of miles away. COM explicitly bars certain kinds of "features"—such as direct access to object data, properties, or variables—that might be convenient in the case of in-process objects but would make it impossible for an out-of-process object to provide the same set of services. This is called *location transparency*.

4 Objects and Interfaces

What is an object? An object is an instantiation of some *class*. At a generic level, a "class" is the definition of a set of related data and capabilities grouped together for some distinguishable common purpose. The purpose is generally to provide some service to "things" outside the object, namely clients that want to make use of those services.

A object that conforms to COM is a special manifestation of this definition of object. A COM object appears in memory much like a C++ object. Unlike C++ objects, however, a client never has direct access to the COM object in its entirety. Instead, clients always access the object through clearly defined contracts: the interfaces that the object supports, *and only those interfaces*.

What exactly is an interface? As mentioned earlier, an interface is a strongly-typed group of semantically-related functions, also called "interface member functions." The name of an interface is always prefixed with an "I" by convention, as in IUnknown. (The real identity of an interface is given by its GUID; names are a programming convenience, and the COM system itself uses the GUIDs exclusively when operating on interfaces.) In addition, while the interface has a specific name (or type) and names of member functions, it defines only how one would use that interface and what behavior is expected from an object through that

interface. Interfaces do not define any implementation. For example, a hypothetical interface called IStack that had member functions of Push and Pop would only define the parameters and return types for those functions and what they are expected to do from a client perspective; the object is free to implement the interface as it sees fit, using an array, linked list, or whatever other programming methods it desires.

When an object "implements an interface" that object implements each member function of the interface and provides pointers to those functions to COM. COM then makes those functions available to any client who asks. This terminology is used in this document to refer to the object as the important element in the discussion. An equivalent term is an "interface on an object" which means the object implements the interface but the main subject of discussion is the interface instead of the object.

.1 Attributes of Interfaces

Given that an interface is a contractual way for an object to expose its services, there are four very important points to understand:

An interface is not a class: An interface is not a class in the normal definition of "class." A class can be instantiated to form an object. An interface cannot be instantiated by itself because it carries no implementation. An object must implement that interface and that object must be instantiated for there to be an interface. Furthermore, different object classes may implement an interface differently yet be used interchangeably in binary form, so long as the behavior conforms to the interface definition (such as two objects that implement IStack where one uses an array and the other a linked list).

An interface is not an object: An interface is just a related group of functions and is the binary standard through which clients and objects communicate. The object can be implemented in any language with any internal state representation, so long as it can provide pointers to interface member functions.

Interfaces are strongly typed: Every interface has its own interface identifier (a GUID) thereby eliminating any chance of collision that would occur with human-readable names. Programmers must consciously assign an identifier to each interface and must consciously support that interface and/or the interfaces defined by others: confusion and conflict among interfaces cannot happen by accident, leading to much improved robustness.

Interfaces are immutable: Interfaces are never versioned, thus avoiding versioning problems. A new version of an interface, created by adding or removing functions or changing semantics, is an entirely new interface and is assigned a new unique identifier. Therefore a new interface does not conflict with an old interface even if all that changed is the semantics. Objects can, of course, support multiple interfaces simultaneous; and they can have a single internal implementation of the common capabilities exposed through two or more similar interfaces, such as "versions" (progressive revisions) of an interface. This approach of immutable interfaces and multiple interfaces per object avoids versioning problems.

Two additional points help to further reinforce the second point about the relationship of an object and its interfaces:

Clients only interact with pointers to interfaces: When a client has access to an object, it has nothing more than a pointer through which it can access the functions in the interface, called simply an *interface pointer*. The pointer is opaque, meaning that it hides all aspects of internal implementation. You cannot see any details about the object such as its state information, as opposed to C++ *object pointers* through which a client may directly access the object's data. In COM, the client can only call functions of the interface to which it has a pointer. But instead of being a restriction, this is what allows COM to provide the efficient binary standard that enables location transparency.

Objects can implement multiple interfaces: A object class can—and typically does—implement more than one interface. That is, the class has more than one set of services to provide from each object. For example, a class might support the ability to exchange data with clients as well as the ability to save its persistent state information (the data it would need to reload to return to its current state) into a file at the client's request. Each of these abilities is expressed through a different interface, so the object must implement two interfaces.

Note that just because a class supports one interface, there is no general requirement that it supports any other. Interfaces are meant to be small contracts that are independent of one another. There are no contractual units smaller than interfaces; if you write a class that implements an interface, your class must imple-

ment all the functions defined by that interface (the implementation doesn't always have to *do* anything). Also note that an object may be attempting to conform to a higher specification than COM, such as a compound document standard like Microsoft's OLE Documents architecture. In such cases, the objects in question must implement specific groups of interfaces to conform to the OLE Documents specification for compound documents. It is then true that all compound document objects will always implement the same basic set of interfaces, but those interfaces themselves do not depend on the presence of the others. It is instead the clients of those objects that depend on the presence of all the interfaces.

The encapsulation of functionality into objects accessed through interfaces makes COM an open, extensible system. It is open in the sense that anyone can provide an implementation of a defined interface and anyone can develop an application that uses such interfaces, such as a compound document application. It is extensible in the sense that new or extended interfaces can be defined without changing existing applications and those applications that understand the new interfaces can exploit them while continuing to interoperate with older applications through the old interfaces.

.2 Object Pictures

It is convenient to adopt a standard pictorial representation for objects and their interfaces. The adopted convention is to draw each interface on an object as a "plug-in jack." These interfaces are generally drawn out the left or right side of a box representing the object as a whole as illustrated in Figure 1-3. If desired, the names of the interfaces are positioned next to the interface jack itself.

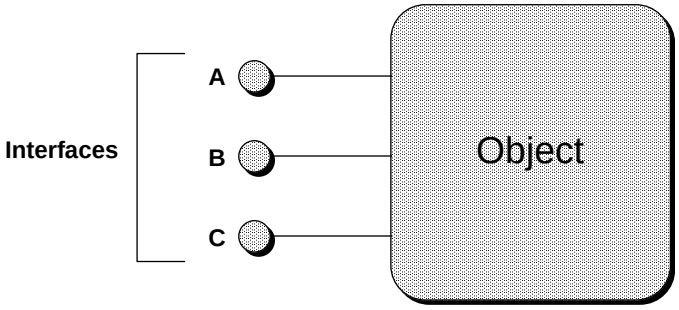


Figure 1-3: A typical picture of an object that supports three interfaces A, B, and C.

The side from which interfaces extend is usually determined by the position of a client in the same picture, if applicable. If there is no client in the picture then the convention is for interfaces to extend to the left as done in Figure 1-3. With a client in the picture, the interfaces extend towards the client, and the client is understood to have a pointer to one or more of the interfaces on that object as illustrated in Figure 1-4.

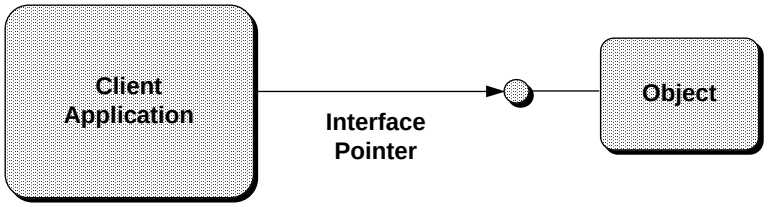


Figure 1-4: Interfaces extend towards the clients connected to them.

In some circumstances a client may itself implement a small object to provide another object with functions to call on various events or to expose services itself. In such cases the client is also an object implementor and the object is also a client. Illustrations for such are similar to that in Figure 1-5.

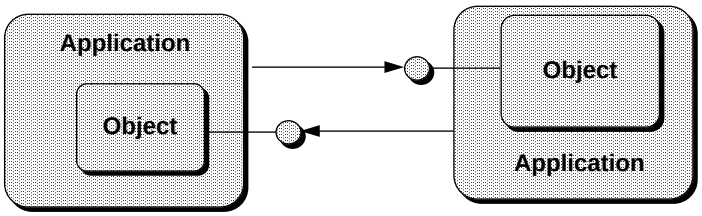


Figure 1-5: Two applications may connect to each other's objects, in which case they extend their interfaces towards each other.

Some objects may be acting as an intermediate between other clients in which case it is reasonable to draw the object with interfaces out both sides with clients on both sides. This is, however, a less frequent case than illustrating an objects connected to one client.

There is one interface that demands a little special attention: IUnknown. This is the base interface of all other interfaces in COM that all objects must support. Usually by implementing any interface at all an object also implements a set of IUnknown functions that are contained within that implemented interface. In some cases, however, an object will implement IUnknown by itself, in which case that interface is extended from the top of the object as shown in Figure 1-6.

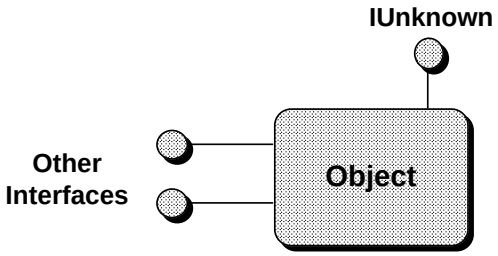


Figure 1-6: The IUnknown interface extends from the top of objects by convention.

In order to use an interface on a object, a client needs to know what it would want to do with that interface—that's what makes it a client of an interface rather than just a client of the object. In the "plug-in jack" concept, a client has to have the right kind of plug to fit into the interface jack in order to do anything with the object through the interface. This is like having a stereo system which has a number of different jacks for inputs and outputs, such as a ¼" stereo jack for headphones, a coax input for an external CD player, and standard RCA connectors for speaker output. Only headphones, CD players, and speakers that have the matching plugs are able to plug into the stereo object and make use of its services. Objects and interfaces in COM work the same way.

.3 *Objects with Multiple Interfaces and* QueryInterface

In COM, an object can support multiple interfaces, that is, provide pointers to more than one grouping of functions. Multiple interfaces is a fundamental innovation of COM as the ability for such avoids versioning problems (interfaces are immutable as described earlier) and any strong association between an interface and an object class. Multiple interfaces is a great improvement over systems in which each object only has one massive interface, and that interface is a collection of everything the object does. Therefore the identity of the object is strongly tied to the exact interface, which introduces the versioning problems once again. Multiple interfaces is the cleanest way around the issue altogether.

The existence of multiple interfaces does, however, bring up a very important question. When a client initially gains access to an object, by whatever means, that client is given *one and only one* interface pointer in return. How, then, does a client access the other interfaces on that same object?

The answer is a member function called QueryInterface that is present in all COM interfaces and can be called on any interface polymorphically. QueryInterface is the basis for a process called *interface negotiation* whereby the client asks the object what services it is capable of providing. The question is asked by calling QueryInterface and passing to that function the unique identifier of the interface representing the services of interest.

Here's how it works: when a client initially gains access to an object, that client will receive at minimum an IUnknown interface pointer (the most fundamental interface) through which it can only control the lifetime of the object—tell the object when it is done using the object—and invoke QueryInterface. The client is programmed to ask each object it manages to perform some operations, but the IUnknown interface has no functions for those operations. Instead, those operations are expressed through other interfaces. The client is thus programmed to negotiate with objects for those interfaces. Specifically, the client will ask each object—by calling QueryInterface—for an interface through which the client may invoke the desired operations.

Now since the object implements QueryInterface, it has the ability to accept or reject the request. If the object accepts the client's request, QueryInterface returns a new pointer to the requested interface to the client. Through that interface pointer the client thus has access to the functions in that interface. If, on the other hand, the object rejects the client's request, QueryInterface returns a null pointer—an error—and the client has no pointer through which to call the desired functions. An illustration of both success and error cases is shown in Figure 1-7 where the client initially has a pointer to interface A and asks for interfaces B and C. While the object supports interface B, it does not support interface C.

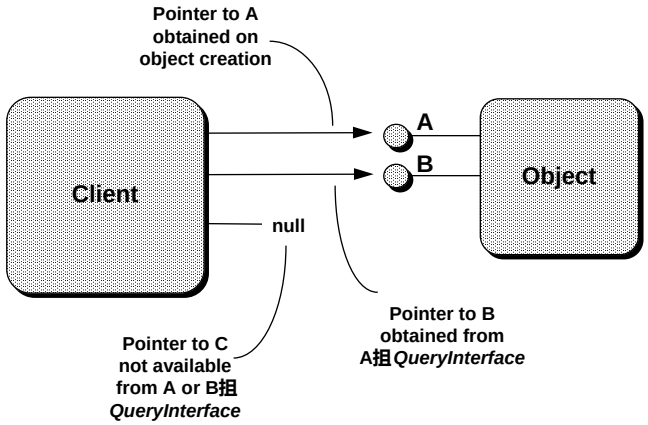


Figure 1-7: Interface negotiation means that a client must ask an object for an interface pointer that is the only way a client can invoke functions of that interface.

A key point is that when an object rejects a call to QueryInterface, it is impossible for the client to ask the object to perform the operations expressed through the requested interface. A client *must* have an interface pointer to invoke functions in that interface, period. If the object refuses to provide one, a client must be prepared to do without, simply failing whatever it had intended to do with that object. Had the object supported that interface, the client might have done something useful with it. Compare this with other object-oriented systems where you cannot know whether or not a function will work until you call that function, and even then, handling of failure is uncertain. QueryInterface provides a reliable and consistent way to know before attempting to call a function.

.1 Robustly Evolving Functionality Over Time

Recall that an important feature of COM is the ability for functionality to evolve over time. This is not just important for COM, but important for all applications. QueryInterface is the cornerstone of that feature as it allows a client to ask an object "do you support functionality X?" It allows the client to implement code that will use this functionality *if and only if* an object supports it. In this manner, the client easily maintains compatibility with objects written before and after the "X" functionality was available, and does so in a robust manner. An old object can reliably answer the question "do you support X" with a "no" whereas a new object can reliably answer "yes." Because the question is asked by calling QueryInterface and therefore on a contract-by-contract basis instead of an individual function-by-function basis, COM is very efficient in this operation.

To illustrate the QueryInterface cornerstone, imagine a client that wishes to display the contents of a number of text files, and it knows that for each file format (ASCII, RTF, Unicode, etc.) there is some object class associated with that format. Besides a basic interface like IUnknown, which we'll call interface A, there are two others that the client wishes to use to achieve its ends: interface B allows a client to tell an object to

load some information from a file (or to save it), and interface C allows a client to request a graphical rendering of whatever data the object loaded from a file and maintains internally.

With these interfaces, the client is then programmed to process each file as follows:

1. Find the object class associated with a the file format.
2. Instantiate an object of that class obtaining a pointer to a basic interface A in return.
3. Check if the object supports loading data from a file by calling interface A's QueryInterface function requesting a pointer to interface B. If successful, ask the object to load the file through interface B.
4. Check if the object supports graphical rendering of its data by calling interface A or B's QueryInterface function (doesn't matter which interface, because queries are uniform on the object) requesting a pointer to interface C. If successful, ask the object for a graphic of the file contents that the client then displays on the screen.

If an object class exists for every file format in the client's file list, and all those objects implement interfaces A, B, and C, then the client will be able to display all the contents of all the files. But in an imperfect world, let's say that the object class for the ASCII text formats does not support interface C, that is, the object can load data from a file and save it to another file if necessary, but can't supply graphics. When the client code, written as described above, encounters this object, the QueryInterface for interface C fails, and the client cannot display the file contents. Oh well...

Now the programmers of the object class for ASCII realizes that they are losing market share because they don't support graphics, and so they update the object class such that it now supports interface C. This new object is installed on the machine alone with the client application, but nothing else changes in the entire system. The client code remains exactly the same. What now happens the next time someone runs the client?

The answer is that the client *immediately begins to use interface C on the updated object*. Where before the object failed QueryInterface when asked for interface C, it now succeeds. Because it succeeds, the client can now display the contents of the file that it previously could not.

Here is the raw power of QueryInterface: a client can be written to take advantage of as much functionality as it would *ideally* like to use on every object it manages. When the client encounters an object that lacks the ideal support, the client can use as much functionality as is available on that given object. When the object it later updated to support new interfaces, the same exact code in the client, without any recompilation, redeployment, or changes whatsoever, automatically begins to take advantage of those additional interfaces. This is true component software. This is true evolution of components independently of one another and retaining full compatibility.

Note that this process also works in the other direction. Imagine that since the client application above was shipped, all the objects for rendering text into graphics were each upgraded to support a new interface D through which a client might ask the object to spell-check the text. Each object is upgraded independently of the client, but since the client never queries for interface D, the objects all continue to work perfectly with just interfaces B and C. In this case the objects support more functionality than the client, but still retain full compatibility requiring absolutely no changes to the client. The client, at a later date, might then implement code to use interface D as well as code for yet a newer interface E (that supports, say, language translation). That client begins to immediately use interface D in all existing objects that support it, without requiring any changes to those objects whatsoever.

This process continues, back and forth, ad infinitum, and applies not only to new interfaces with new functionality but also to improvements of existing interfaces. Improved interface are, for all practical purposes, a brand-new interface because any change to any interface requires a new interface identifier. A new identifier isolates an improved interface from its predecessor as much as it isolates unrelated interfaces from each other. There is no concept of "version" because the interfaces are totally different in identity.

So up to this point there has been this problem of versioning, presented at the beginning of this chapter, that made independent evolution of clients and objects practically impossible. But now, for all time, QueryInterface solves that problem and removes the barriers to rapid software innovation without the growing pains.

5 Clients, Servers, and Object Implementors

The interaction between objects and the users of those objects in COM is based on a client/server model. This chapter has already been using the term 'client' to refer to some piece of code that is using the services of an object. Because an object supplies services, the implementor of that object is usually called the "server," the one who serves those capabilities. A client/server architecture in any computing environment leads to greater robustness: if a server process crashes or is otherwise disconnected from a client, the client can handle that problem gracefully and even restart the server if necessary. As robustness is a primary goal in COM, then a client/server model naturally fits.

However, there is more to COM than just clients and servers. There are also *object implementors*, or some program structure that implements an object of some kind with one or more interfaces on that object. Sometimes a client wishes to provide a mechanism for an object to call back to the client when specific events occur. In such cases, COM specifies that the client itself implements an object and hands that object's first interface pointer to the other object outside the client. In that sense, both sides are clients, both sides are servers in some way. Since this can lead to confusion, the term "server" is applied in a much more specific fashion leading to the following definitions that apply in all of COM:

Object A unit of functionality that implements one or more interfaces to expose that functionality. For convenience, the word is used both to refer to an object class as well as an individual instantiation of a class. Note that an object class does not need a class identifier in the COM sense such that other applications can instantiate objects of that class—the class used to implement the object internally has no bearing on the externally visible COM class identifier.

Object Implementor Any piece of code, such as an application, that has implemented an object with any interfaces for any reason. The object is simply a means to expose functions outside the particular application such that outside agents can call those functions. Use of "object" by itself implies an object found in some "object implementor" unless stated otherwise.

Client There are two definitions of this word. The general definition is any piece of code that is using the services of some object, wherever that object might be implemented. A client of this sort is also called an "object user." The second definition is the active agent (an application) that drives the flow of operation between itself an other objects and uses specific COM "implementation locator" services to instantiate or create objects through servers of various object classes.

Server A piece of code that structures an object class in a specific fashion and assigns that class a COM class identifier. This enables a client to pass the class identifier to COM and ask for an object of that class. COM is able to load and run the server code, ask the sever to create an object of the class, and connect that new object to the client. A server is specifically the necessary structure around an object that serves the object to the rest of the system and associates the class identifier: a server is not the object itself. The word "server" is used in discussions to emphasize the serving agent more than the object. The phrase "server object" is used specifically to identify an object that is implemented in a server when the context is appropriate.

Putting all of these pieces together, imagine a client application that initially uses COM services to create an object of a particular class. COM will run the server associated with that class and have it create an object, returning an interface pointer to the client. With that interface pointer the client can query for any other interface on the object. If a client wants to be notified of events that happen in the object in the server, such as a data change, the client itself will implement an "event sink" object and pass the interface pointer to that sink to the server's object through an interface function call. The server holds onto that interface pointer and thus itself becomes a client of the sink object. When the server object detects an appropriate event, it calls the sink object's interface function for that even. The overall configuration created in this scenario is much like that shown earlier in Figure 1-5. There are two primary modules of code (the original client and the server) who both implement objects and who both act in some aspects as clients to establish the configuration.

When both sides in a configuration implement objects then the definition of "client" is usually the second one meaning the active agent who drives the flow of operation between all objects, even when there is more than one piece of code that is acting like a client of the first definition. This specification endeavors to provide enough context to make it clear what code is responsible for what services and operations.

.1 Server Flavors: In-Process and Out-Of-Process

As defined in the last section, a "server" in general is some piece of code that structures some object in such a way that COM "implementor locator" services can run that code and have it create objects. The section below entitled "The COM Library" expands on the specific responsibilities of COM in this sense.

Any specific server can be implemented in one of a number of flavors depending on the structure of the code module and its relationship to the client process that will be using it. A server is either "in-process" which means it's code executes in the same process space as the client, or "out-of-process" which means it runs in another process on the same machine or in another process on a remote machine. These three types of servers are called "in-process," "local," and "remote" as defined below:

In-Process Server A server that can be loaded into the client's process space and serves "in-process objects." Under Microsoft Windows and Microsoft Windows NT, these are implemented as "dynamic link libraries" or DLLs. This specification uses **DLL** as a generic term to describe any piece of code that can be loaded in this fashion which will, of course, differ between operating systems.

Local Server A server that runs in a separate process on the same machine as the client and serves "local objects." This type of server is another complete application of its own thus defining the separate **process**. This specification uses the terms "**EXE**" or "**executable**" to describe an application that runs in its own process as opposed to a DLL which must be loaded into an existing process.

Remote Server A server that runs on a separate machine and therefore always runs in another process as well to serve "**remote** objects." Remote servers may be implemented in either DLLs or EXEs; if a remote server is implemented in a DLL, a surrogate process will be created for it on the remote machine.

Note that the same words "in-process," "local," and "remote" are used in this specification as a qualifier for the word "object" where emphasis is on the object more than the server.

Object implementors choose the type of server based on the requirements of implementation and deployment. COM is designed to handle all situations from those that require the deployment of many small, lightweight in-process objects (like controls, but conceivably even smaller) up to those that require deployment of a huge central corporate database server. Furthermore, COM does so in a transparent fashion, with what is called *location transparency*, the topic of the next section.

.2 Location Transparency

COM is designed to allow clients to *transparently* communicate with objects regardless of where those objects are running, be it the same process, the same machine, or a different machine. What this means is that there is a *single programming model* for all types of objects for not only clients of those objects but also for the servers of those objects.

From a client's point of view, all objects are access through interface pointers. A pointer must be in-process, and in fact, any call to an interface function always reaches *some* piece of in-process code first. If the object is in-process, the call reaches it directly, with no intervening system-infrastructure code. If the object is out-of-process, then the call first reaches what is called a "proxy" object provided by COM itself which generates the appropriate remote procedure call to the other process or the other machine.

From a server's point of view, all calls to an object's interface functions are made through a pointer to that interface. Again, a pointer only has context in a single process, and so the caller must always be some piece of in-process code. If the object is in-process, the caller is the client itself. Otherwise, the caller is a "stub" object provided by COM that picks up the remote procedure call from the "proxy" in the client process and turns it into an interface call to the server object.

As far as both clients and servers know, they always communicate directly with some other in-process code as illustrated in Figure 1-8.

The bottom line is that *dealing with in-process or remote objects is transparent and identical to dealing with in-process objects*. This location transparency has a number of key benefits:

- **A common solution to problems that are independent of the distance between client and server**: For example, connection, function invocation, interface negotiation, feature evolution, and so forth.
- **Programmers leverage their learning**: New services are simply exposed through new interfaces, and once programmers learn how to deal with interfaces, they already know how to deal with new services that will be created in the future. This is a great improvement over environments where each service is exposed in a completely different fashion.
- **Systems implementation is centralized**: The implementors of COM can focus on making the central process of providing this transparency as efficient and powerful as possible such that every piece of code that uses COM benefits immensely.
- **Interface designers focus on design**: In designing a suite of interfaces, the designers can spend their time in the essence of the design—the contracts between the parties—without having to think about the underlying communication mechanisms for any interoperability scenario. COM provides those mechanisms for free and transparently.

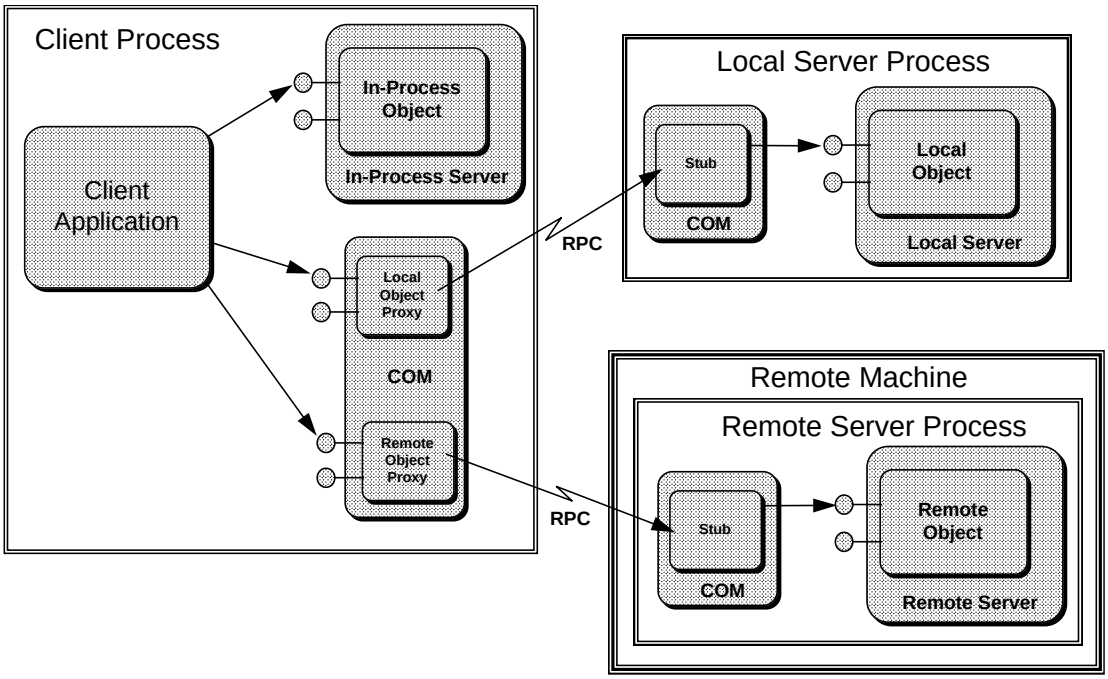


Figure 1-8: Clients always call in-process code; objects are always called by in-process code. COM provides the underlying transparent RPC.

The clear separation of interface from implementation provided by location transparency for some situations gets in the way when performance is of critical concern. When designing an interface while focusing on making it natural and functional from the client's point of view, one is sometimes lead to design decisions that are in tension with allowing for efficient implementation of that interface across a network. What is needed is not pure location transparency, but "location transparency, unless you need to care." COM provides this capability. An object implementor can if he wishes support *custom marshaling* which allows his objects to take special action when they are used from across the network, different action if he would like than is used in the local case. The key point is that this is done completely transparently to the client. Taken as a whole, this architecture allows one to design client / object interfaces at their natural and easy semantic level without regard to network performance issues, then at a later address network performance issues without disrupting the established design.[9]

[9] Not only are there situations where there is a need for designs optimized for cross network efficiency, but there are also cases where in-process efficiency is more important. Just as COM provides mechanisms whereby the remote case can be optimized (custom marshaling) it also allows for the design of interfaces that are optimized for the in-process case.

Also note again that COM is not a specification for how applications are structured: it is a specification for how applications interoperate. For this reason, COM is not concerned with the internal structure of an application—that is the job of programming languages and development environments. Conversely, programming environments have no set standards for working with objects outside of the immediate application. C++, for example, works extremely well to work with objects inside an application, but has no support for working with objects outside the application. Generally all other programming languages are the same in this regard. Therefore COM, through language-independent interfaces, picks up where programming languages leave off to provide the network-wide interoperability.

6 The COM Library

It should be clear by this time that COM itself involves some systems-level code, that is, some implementation of its own. However, at the core the Component Object Model by itself is a specification (hence "Model") for how objects and their clients interact through the binary standard of interfaces. As a specification it defines a number of other standards for interoperability:

- The fundamental process of interface negotiation through QueryInterface.
- A *reference counting* mechanism through objects (and their resources) are managed even when connected to multiple clients.
- Rules for memory allocation and responsibility for those allocations when exchanged between independently developed components.
- Consistent and rich error reporting facilities.

In addition to being a specification, COM is also an implementation contained what is called the "COM Library." The implementation is provided through a library (such as a DLL on Microsoft Windows) that includes:

- A small number of fundamental API functions that facilitate the creation of COM applications, both clients and servers. For clients, COM supplies basic object creation functions; for servers the facilities to expose their objects.
- Implementation locator services through which COM determines from a class identifier which server implements that class and where that server is located. This includes support for a level of indirection, usually a system registry, between the identity of an object class and the packaging of the implementation such that clients are independent of the packaging which can change in the future.
- Transparent remote procedure calls when an object is running in a local or remote server, as illustrated in Figure 1-8 in the previous section.
- A standard mechanism to allow an application to control how memory is allocated within its process.

In general, only one vendor needs to, or should, implement a COM Library for any particular operating system. For example, Microsoft has implemented COM on Microsoft Windows 3.1, Microsoft Windows 95, Microsoft Windows NT, and the Apple Macintosh. Part V of this document specifies in detail the internals of the COM Library for those vendors who wish to implement the COM Library on a platform for which it does not already have support.

7 COM as a Foundation

The binary standard of interfaces is the key to COM's extensible architecture, providing the foundation upon which is built the rest of COM and other systems such as OLE.

.1 COM Infrastructure

COM provides more than just the fundamental object creation and management facilities: it also builds an infrastructure of three other core operating system components.

Persistent Storage: 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 ad-

dition, 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.

Persistent, Intelligent Names (Monikers): The ability to give a specific *instantiation* of an object a particular name that would allow a client to 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.[10]

Uniform Data Transfer: Standard 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 standards include powerful structures used to describe data formats as well as the storage mediums on which the data is exchanged.

The combination of the foundation and the infrastructure 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 1-9. 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.

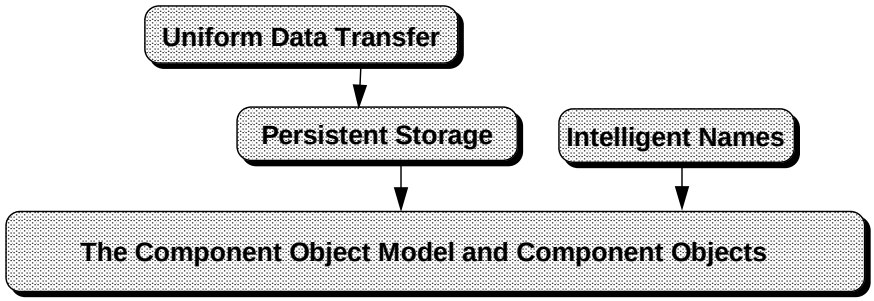


Figure 1-9: COM is built in progressively higher level technologies that depend upon lower level technologies.

.2 OLE

Microsoft's OLE technology is really a collection of additional higher-level technologies that build upon COM and its infrastructure. OLE version 2.0 was the first deployment of a subset of this COM specification that included support for in-process and local objects and all the infrastructure technologies but did not support remote objects. OLE 2 includes mostly user-interface oriented features based on usability, application integration, and automation of tasks. All of these features are implemented by means of specific interfaces on different objects and defined sequences of operation in both clients and servers and their relationships and dependencies on the lower level infrastructure of COM is shown in Figure 1-10.

[10] Monikers are COM's way of providing support for what other object systems (e.g. CORBA) call persistent interfaces.

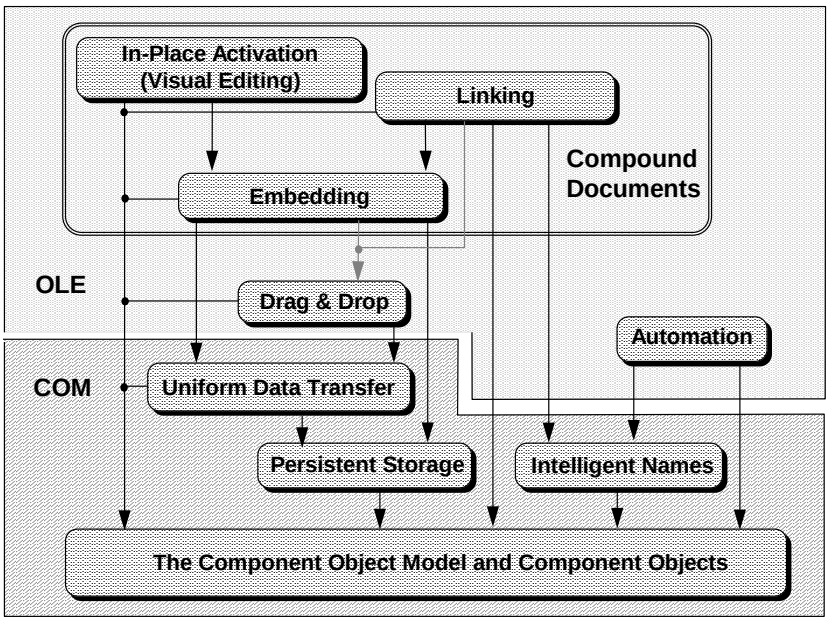


Figure 1-10: OLE builds its features on COM.

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

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.

Compound Documents: The ability to embed or link information in a central document encouraging a more document-centric user interface. Also includes In-Place Activation (also called "Visual Editing") as a user interface improvement to embedding where the end user can works 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.

This page left intentionally blank.

Part II: Component Object Model Programming Interface

Part II contains the programming interface to COM, the suite of interfaces and APIs by which Component Object Model software is implemented and used.

This page intentionally left blank.

2. Component Object Model Technical Overview

Chapter 1 introduced some important challenges and problems in computing today and the Component Object Model as a solution to these problems. Chapter 1 introduced interfaces, mentioned the base interface called IUnknown, and described how interfaces are generally used to communicate between an object and a client of that object, and explained the role that COM has in that communication to provide location transparency.

Yet there are plenty of topics that have not been covered in much technical detail, specifically, how certain mechanisms work, some of the interfaces involved, and how some of these interfaces are used on a high level. This chapter will describe COM in a more technical light but not going as far as describing individual interface functions or COM Library API functions. Instead, this chapter will refer to later chapters in the COM Specification that cover various topics in complete detail including the specifications for functions and interfaces themselves.

This chapter is generally organized in the same order as Chapter 1 and covers the following topics which are then treated in complete detail in the indicated chapters:

- Objects and Interfaces: A comparison of interfaces to C++ classes, the IUnknown interface (including the QueryInterface function and reference counting), the structure of an instantiated interface and the benefits of that structure, and how clients of objects deal with interfaces. Chapter 3 covers the underlying interfaces and API functions themselves.
- COM Applications: The responsibilities of all applications making use of COM which includes rules for memory management. How applications meet these responsibilities is covered in Chapter 4.
- COM Clients and Servers: The roles and responsibilities of each specific type of application, the use of class identifiers, and the COM Library's role in providing communication. Chapter 5 and 6 treat COM Clients and Servers separately. How COM achieves location transparency is described in Chapter 7.
- Reusability: A discussion about why implementation inheritance is not used in COM and what mechanisms are instead available. How an object server is written to handle the COM mechanisms is a topic of Chapter 6.
- Connectable Objects: A brief overview of the connection point interfaces and semantics. The actual functional specification of connectable objects is in Chapter 9.
- Persistent Storage: A detailed look at what persistent storage is, what benefits it holds for applications including incremental access and transactioning support, leaving the APIs and interface specifications to Chapter 10.
- Persistent, Intelligent Names: Why it is important to assign names to individual object instantiations (as opposed to a class identifier for an object class) and the mechanisms for such naming including moniker objects. The interfaces a moniker implements as well as other support functions are described in Chapter 11.
- Uniform Data Transfer: The separation of transfer protocols from data exchange, improvements to data format descriptions, the expansion of available exchange mediums (over global memory), and data change notification mechanisms. New data structures and interfaces specified to support data transfer is given in Chapter 12.

1 Objects and Interfaces

Chapter 1 described that interfaces are—strongly typed semantic contracts between client and object—and that an object in COM is any structure that exposes its functionality through the interface mechanism. In addition, Chapter 1 noted how interfaces follow a binary standard and how such a standard enables clients and objects to interoperate regardless of the programming languages used to implement them. While the *type* of an interface is by colloquial convention referred to with a name starting with an "I" (for interface), this name is only of significance in source-level programming tools. Each interface itself—the immutable contract, that is—as a functional group is referred to at runtime with a globally-unique interface identifier, an "IID" that allows a client to ask an object if it supports the semantics of the interface without unnecessary overhead and without versioning problems. Clients ask questions using a QueryInterface function that all objects support through the base interface, IUnknown.

Furthermore, clients always deal with objects through interface pointers and never directly access the object itself. Therefore an interface is not an object, and an object can, in fact, have more than one interface if it has more than one group of functionality it supports.

Let's now turn to how interfaces manifest themselves and how they work.

.1 Interfaces and C++ Classes

As just reiterated, an interface is not an object, nor is it an object class. Given an interface definition by itself, that is, the type definition for an interface name that begins with "I," you cannot create an object of that type. This is one reason why the prefix "I" is used instead of the common C++ convention of using a "C" to prefix an object class, such as CMyClass. While you can instantiate an object of a C++ class, you cannot instantiate an object of an interface type.

In C++ applications, interfaces are, in fact, defined as *abstract base classes*. That is, the interface is a C++ class that contains nothing but pure virtual member functions. This means that the interface carries no implementation and only prescribes the function signatures for some other class to implement—C++ compilers will generate compile-time errors for code that attempts to instantiate an abstract base class. C++ applications implement COM objects by inheriting these function signatures from one or more interfaces, overriding each interface function, and providing an implementation of each function. This is how a C++ COM application "implements interfaces" on an object.

Implementing objects and interfaces in other languages is similar in nature, depending on the language. In C, for example, an interface is a structure containing a pointer to a table of function pointers, one for each method in the interface. It is very straightforward to use or to implement a COM object in C, or indeed in any programming language which supports the notion of function pointers. No special tools or language enhancements are required (though of course such things may be desirable).

The abstract-base class comparison exposes an attribute of the "contract" concept of interfaces: if you want to implement any single function in an interface, you must provide some implementation for *every* function in that interface. The implementation might be nothing more than a single return statement when the object has nothing to do in that interface function. In most cases there is some meaningful implementation in each function, but the number of lines of code varies greatly (one line to hundreds, potentially).

A particular object will provide implementations for the functions in every interface that it supports. Objects which have the same set of interfaces and the same implementations for each are often said (loosely) to be instances of the same class because they generally implement those interfaces in a certain way. However, all access to the instances of the class by clients will only be through interfaces; clients know nothing about an object other than it supports certain interfaces. As a result, classes play a much less significant role in COM than they do in other object oriented systems.

COM uses the word "interface" in a sense different from that typically used in object-oriented programming using C++. In the C++ context, "interface" describes *all* the functions that a class supports and that clients of an object can call to interact with it. A COM interface refers to a pre-defined group of related functions that a COM class implements, but does not necessarily represent *all* the functions that the class supports. This separation of an object's functionality into groups is what enables COM and COM applications to avoid the problems inherent with versioning traditional all-inclusive interfaces.

.2 Interfaces and Inheritance

COM separates class hierarchy (or indeed any other *implementation* technology) from interface hierarchy and both of those from any implementation hierarchy. Therefore, interface inheritance is only applied to reuse the definition of the contract associated with the base interface. There is no selective inheritance in COM: if one interface inherits from another, it includes all the functions that the other interface defines, for the same reason than an object must implement all interface functions it inherits.

Inheritance is used sparingly in the COM interfaces. Most of the pre-defined interfaces inherit directly from IUnknown (to receive the fundamental functions like QueryInterface), rather than inheriting from another interface to add more functionality. Because COM interfaces are inherited from IUnknown, they tend to be small and distinct from one another. This keeps functionality in separate groups that can be independently updated from the other interfaces, and can be recombined with other interfaces in semantically useful ways.

In addition, interfaces only use single inheritance, never multiple inheritance, to obtain functions from a base interface. Providing otherwise would significantly complicate the interface method call sequence, which is just an indirect function call, and, further, the utility of multiple inheritance is subsumed within the capabilities provided by QueryInterface.

.3 Interface Definitions: IDL

When a designer creates an interface, that designer usually defines it using an Interface Description Language (IDL). From this definition an IDL compiler can generate header files for programming languages such that applications can use that interface, create proxy and stub objects to provide for remote procedure calls, and output necessary to enable RPC calls across a network.

IDL is simply a tool (one of possibly many) for the convenience of the interface designer and is not central to COM's interoperability. It really just saves the designer from manually creating many header files for each programming environment and from creating proxy and stub objects by hand, which would not likely be a fun task.

Chapter 13 describes the Microsoft Interface Description Language in detail. In addition, Chapter 14 covers Type Libraries which are the machine readable form of IDL, used by tools and other components at runtime.

.4 Basic Operations: The IUnknown *Interface*

All objects in COM, through any interface, allow clients access to two basic operations:

- Navigating between multiple interfaces on an object through the QueryInterface function.
- Controlling the object's lifetime through a reference counting mechanism handled with functions called AddRef and Release.

Both of these operations as well as the three functions (and only these three) make up the IUnknown interface from which all other interfaces inherit. That is, all interfaces are polymorphic with IUnknown so they all contain QueryInterface, AddRef, and Release functions.

.1 Navigating Multiple Interfaces: the *QueryInterface* Function

As described in Chapter 1, QueryInterface is the mechanism by which a client, having obtained one interface pointer on a particular object, can request additional pointers to *other* interfaces on that same object. An input parameter to QueryInterface is the interface identifier (IID) of the interface being requested. If the object supports this interface, it returns that interface on itself through an accompanying output parameter typed as a generic void; if not, the object returns an error.

In effect, what QueryInterface accomplishes is a switch between contracts on the object. A given interface embodies the interaction that a certain contract requires. Interfaces are groups of functions because contracts in practice invariably require more than one supporting function. QueryInterface separates the request "Do you support a given contract?" from the high-performance use of that contract once negotiations have been successful. Thus, the (minimal) cost of the contract negotiation is amortized over the subsequent use of the contract.

Conversely, QueryInterface provides a robust and reliable way for a component to indicate that in fact does *not* support a given contract. That is, if using QueryInterface one asks an "old" object whether it supports a "new" interface (one, say, that was invented after the old object has been shipped), then the old object will reliably and robustly answer "no;" the technology which supports this is the algorithm by which IIDs are allocated. While this may seem like a small point, it is excruciatingly important to the overall architecture of the system, and this capability to robustly inquire of old things about new functionality is, surprisingly, a feature not present in most other object architectures.

The strengths and benefits of the QueryInterface mechanism need not be reiterated here further, but there is one pressing issue: how does a client obtain its first interface pointer to an object? That question is of central interest to COM applications but has no one answer. There are, in fact, four methods through which a client obtains its first interface pointer to a given object:

- Call a COM Library API function that creates an object of a pre-determined type—that is, the function will only return a pointer to one specific interface for a specific object class.
- Call a COM Library API function that can create an object based on a class identifier and that returns any type interface pointer requested.
- Call a member function of some interface that creates another object (or connects to an existing one) and returns an interface pointer on that separate object.[11]
- Implement an object with an interface through which other objects pass their interface pointer to the client directly. This is the case where the client is an object implementor and passes a pointer to its object to another object to establish a bi-directional connection.

.2 Reference Counting: Controlling Object Life-cycle

Just like an application must free memory it allocated once that memory is no longer in use, a client of an object is responsible for freeing the object when that object is no longer needed. In an object-oriented system the client can only do this by giving the object an instruction to free itself.

However, the difficulty lies in having the object know when it is safe to free itself. COM objects, which are dynamically allocated, must allow the client to decide when the object is no longer in use, especially for local or remote objects that may be in use by multiple clients at the same time—the object must wait until *all* clients are finished with it before freeing itself.

COM specifies a *reference counting* mechanism to provide this control. Each object maintains a 32-bit reference count that tracks how many clients are connected to it, that is, how many pointers exist to any of its interfaces in any client. The use of a 32-bit counter (more than four billions clients) means that there's virtually no chance of overloading the count.

The two IUnknown functions of AddRef and Release that all objects must implement control the count: AddRef increments the count and Release decrements it. When the reference count is decremented to zero, Release is allowed to free the object because no one else is using it anywhere. Most objects have only one implementation of these functions (along with QueryInterface) that are shared between all interfaces, though this is just a common implementation approach. Architecturally, from a client's perspective, reference counting is strictly and clearly a per-interface notion.

Whenever a client calls a function that returns a new interface pointer to it, such as QueryInterface, the function being called is responsible for incrementing the reference count through the returned pointer. For example, when a client first creates an object it receives back an interface pointer to an object that, from the client's point of view, has a reference count of one. If the client calls QueryInterface once for another interface pointer, the reference count is two. The client must then call Release through *both* pointers (in any order) to decrement the reference count to zero before the object as a whole can free itself.

In general, every copy of any pointer to any interface requires a reference count on it. Chapter 3, however, identifies some important optimizations that can be made to eliminate extra unnecessary overhead with reference counting and identifies the specific cases in which calling AddRef is absolutely necessary.

.5 How an Interface Works

An instantiation of an interface implementation (because the defined interfaces themselves cannot be instantiated without implementation) is simply pointer to an array of pointers to functions. Any code that has access to that array—a pointer through which it can access the array—can call the functions in that interface. In reality, a pointer to an interface is actually a pointer to a pointer to the table of function pointers. This is an inconvenient way to speak about interfaces, so the term "interface pointer" is used instead to refer to this multiple indirection. Conceptually, then, an interface pointer can be viewed simply as a pointer to a function table in which you can call those functions by dereferencing them by means of the interface pointer as shown in Figure 2-1.

[11] Connecting to objects through an "intelligent/persistent name" (moniker) falls into this category.

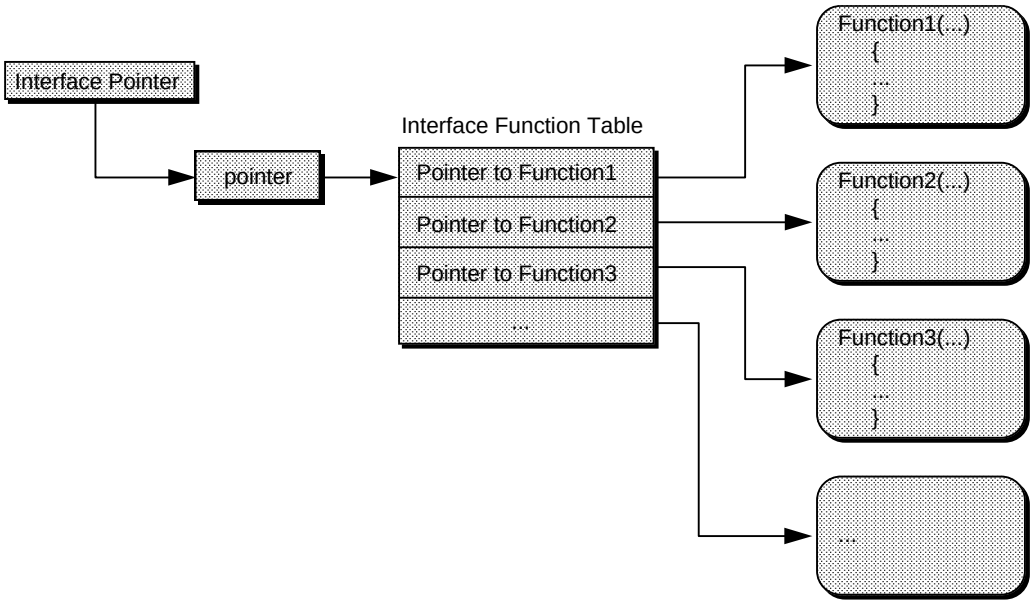
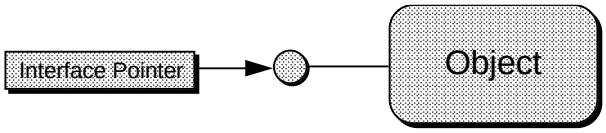


Figure 2-1: An interface pointer is a pointer to a pointer to an array of pointers to the functions in the interface.

Since these function tables are inconvenient to draw they are represented with the "plug-in jack" or "bubbles and push-pins" diagram first shown in Chapter 1 to mean exactly the same thing:



Objects with multiple interfaces are merely capable of providing more than one function table. Function tables can be created manually in a C application or almost automatically with C++ (and other object oriented languages that support COM). Chapter 3 describes exactly how this is accomplished along with how the implementation of the interface functions know exactly which object is being used at any given time.

With appropriate compiler support (which is inherent in C and C++), a client can call an interface function through the name of the function and not its position in the array. The names of functions and the fact that an interface is a type allows the compiler to check the types of parameters and return values of each interface function call. In contrast, such type-checking is not available even in C or C++ if a client used a position-based calling scheme.

.6 Interfaces Enable Interoperability

COM is designed around the use of interfaces because interfaces enable interoperability. There are three properties of interfaces that provide this: polymorphism, encapsulation, and transparent remoting.

.1 Polymorphism

Polymorphism means the ability to assume many forms, and in object-oriented programming it describes the ability to have a single statement invoke different functions at different times. All COM interfaces are polymorphic; when you call a function using an interface pointer, you don't specify which implementation is invoked. A call to pInterface->SomeFunction can cause different code to run depending on what kind of object is the implementor of the interface pointed by pInterface—while the semantics of the function are always the same, the implementation details can vary.

Because the interface standard is a binary standard, clients that know how to use a given interface can interact with any object that supports that interface *no matter how the object implements that contract*. This allows interoperability as you can write an application that can cooperate with other applications without you knowing who or what they are beforehand.

.2 Encapsulation

Other advantages of COM arise from its enforcement of encapsulation. If you have implemented an interface, you can change or update the implementation without affecting any of the clients of your class. Similarly, you are immune to changes that others make in their implementations of their interfaces; if they improve their implementation, you can benefit from it without recompiling your code.

This separation of contract and implementation can also allow you to take advantage of the different implementations underlying an interface, even though the interface remains the same. Different implementations of the same interface are interchangeable, so you can choose from multiple implementations depending on the situation.

Interfaces provides extensibility; a class can support new functionality by implementing additional interfaces without interfering with any of its existing clients. Code using an object's ISomeInterface is unaffected if the class is revised to in addition support IAnotherInterface.

.3 Transparent Remoting

COM interfaces allow one application to interact with others anywhere on the network just as if they were on the same machine. This expands the range of an object's interoperability: your application can use any object that supports a given contract, no matter how the object implements that contract, and no matter what machine the object resides on.

Before COM, class code such as C++ class libraries ran in same process, either linked into the executable or as a dynamic-link library. Now class code can run in a separate process, on the same machine or on a different machine, and your application can use it with no special code. COM can intercept calls to interfaces through the function table and generate remote procedure calls instead.

2 COM Application Responsibilities

Each process that uses COM in any way—client, server, object implementor—is responsible for three things:

1. Verify that the COM Library is a compatible version with the COM function CoBuildVersion.
2. Initialize the COM Library before using any other functions in it by calling the COM function CoInitialize.
3. Un-initialize the COM Library when it is no longer in use by calling the COM function CoUninitialize.

While these responsibilities and functions are covered in detail in Chapter 4, note first that most COM Library functions, primarily those that deal with the COM foundation, are prefixed with "Co" to identify their origin. The COM Library may implement other functions to support persistent storage, naming, and data transfer without the "Co" prefix.

.1 Memory Management Rules

In COM there are many interface member functions and APIs which are called by code written by one programming organization and implemented by code written by another. Many of the parameters and return values of these functions are of types that can be passed around by value; however, sometimes there arises the need to pass data structures for which this is not the case, and for which it is therefore necessary that the caller and the callee agree as to the allocation and de-allocation policy. This could in theory be decided and documented on an individual function by function basis, but it is much more reasonable to adopt a universal convention for dealing with these parameters. Also, having a clear convention is important technically in order that the COM remote procedure call implementation can correctly manage memory.

Memory management of pointers to interfaces is always provided by member functions in the interface in question. For all the COM interfaces these are the AddRef and Release functions found in the IUnknown interface, from which again all other COM interfaces derive (as described earlier in this chapter). This section relates only to non-by-value parameters which are *not* pointers to interfaces but are instead more mundane things like strings, pointers to structures, etc.

The COM Library provides an implementation of a memory allocator (see CoGetMalloc and CoTaskMemAlloc). Whenever ownership of an allocated chunk of memory is passed through a COM interface or between a client and the COM library, this allocator must be used to allocate the memory.[12]

Each parameter to and the return value of a function can be classified into one of three groups: an **in** parameter, an **out** parameter (which includes return values), or an **in-out** parameter. In each class of parameter, the responsibility for allocating and freeing non-by-value parameters is the following:

in parameter	Allocated and freed by the caller.
out parameter	Allocated by the callee; freed by the caller.
in-out parameter	Initially allocated by the caller, then freed and re-allocated by the callee if necessary. As with out parameters, the caller is responsible for freeing the final returned value.

In the latter two cases there is one piece of code that allocates the memory and a different piece of code that frees it. In order for this to be successful, the two pieces of code must of course have knowledge of which memory allocator is being used. Again, it is often the case that the two pieces of code are written by independent development organizations. To make this work, we require that the COM allocator be used.

Further, the treatment of out and in-out parameters in failure conditions needs special attention. If a function returns a status code which is a failure code, then in general the caller has no way to clean up the *out* or *in-out* parameters. This leads to a few additional rules:

out parameter	In error returns, out parameters must be *always* reliably set to a value which will be cleaned up without any action on the caller's part. Further, it is the case that all out pointer parameters (usually passed in a pointer-to-pointer parameter, but which can also be passed as a member of a caller-allocate callee-fill structure) *must* explicitly be set to NULL. The most straightforward way to ensure this is (in part) to set these values to NULL on function entry.[13] (On success returns, the semantics of the function of course determine the legal return values.)
in-out parameter	In error returns, all in-out parameters must either be left alone by the callee (and thus remaining at the value to which it was initialized by the caller; if the caller didn't initialize it, then it's an out parameter, not an in-out parameter) or be explicitly set as in the out parameter error return case.

The specific COM APIs and interfaces that apply to memory management are discussed further below.

Remember that these memory management conventions for COM applications apply only across public interfaces and APIs—there is no requirement at all that memory allocation strictly internal to a COM application need be done using these mechanisms.

3 The COM Client/Server Model

Chapter 1 mentioned how COM supports a model of client/server interaction between a user of an object's services, the client, and the implementor of that object and its services, the server. To be more precise, the client is *any* piece of code (not necessarily an application) that somehow obtains a pointer through which it can access the services of an object and then invokes those services when necessary. The server is some piece of code that implements the object and structures in such a way that the COM Library can match that implementation to a class identifier, or CLSID. The involvement of a class identifier is what differentiates a server from a more general object implementor.

The COM Library uses the CLSID to provide "implementation locator" services to clients. A client need only tell COM the CLSID it wants and the type of server—in-process, local, or remote—that it allows

[12] Any internally-used memory in COM and in-process objects can use any allocation scheme desired, but the COM memory allocator is a handy, efficient, and thread-safe allocator.

[13] This rule is stronger than it might seem to need to be in order to promote more robust application interoperability.

COM to load or launch. COM, in turn, locates the implementation of that class and establishes a connection between it and the client. This relationship between client, COM, and server is illustrated in Figure 2-2 on the next page.

Chapter 1 also introduced the idea of Location transparency, where clients and servers never need to know how far apart they actually are, that is, whether they are in the same process, different processes, or different machines.

This section now takes a closer look at the mechanisms in COM that make this transparency work as well as the responsibilities of client and server applications.

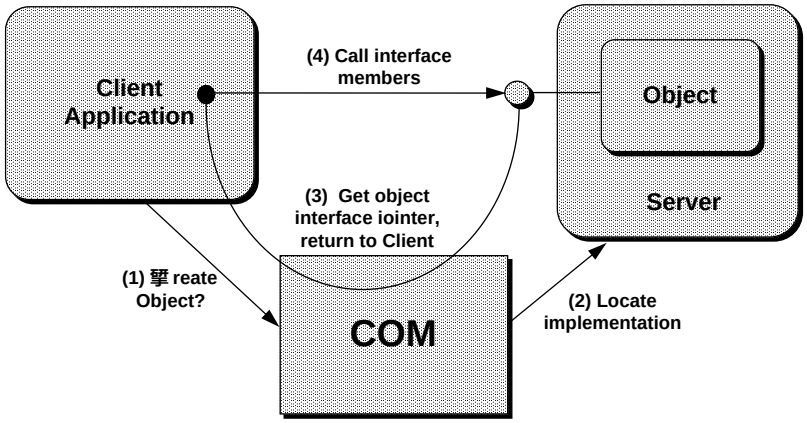


Figure 2-2: Clients locate and access objects through implementation locator services in COM. COM then connects the client to the object in a server. Compare this with Figure 1-2 in Chapter 1.

.1 COM Objects and Class Identifiers

A COM class is a particular implementation of certain interfaces; the implementation consists of machine code that is executed whenever you interact with an instance of the COM class. COM is designed to allow a class to be used by different applications, including applications written without knowledge of that particular class's existence. Therefore class code exists either in a dynamic linked library (DLL) or in another application (EXE). COM specifies a mechanism by which the class code can be used by many different applications.

A COM object is an object that is identified by a unique 128-bit CLSID that associates an object class with a particular DLL or EXE in the file system. A CLSID is a GUID itself (like an interface identifier), so no other class, no matter what vendor writes it, has a duplicate CLSID. Servers implementors generally obtain CLSIDs through the CoCreateGUID function in COM, or through a COM-enabled tool that internally calls this function.

The use of unique CLSIDs avoids the possibility of name collisions among classes because CLSIDs are in no way connected to the names used in the underlying implementation. So, for example, two different vendors can write classes which they call "StackClass," but each will have a unique CLSID and therefore avoid any possibility of a collision.

Further, no central authoritative and bureaucratic body is needed to allocate or assign CLSIDs. Thus, server implementors across the world can independently develop and deploy their software without fear of accidental collision with software written by others.

On its host system, COM maintains a registration database (or "registry") of all the CLSIDs for the servers installed on the system, that is, a mapping between each CLSID and the location of the DLL or EXE that houses the server for that CLSID. COM consults this database whenever a client wants to create an instance of a COM class and use its services. That client, however, only needs to know the CLSID which keeps it independent of the specific location of the DLL or EXE on the particular machine.

If a requested CLSID is not found in the local registration database, various other administratively-controlled algorithms are available by which the implementation is attempted to be located on the network to which the local machine may be attached; these are explained in more detail below.

Given a CLSID, COM invokes a part of itself called the Service Control Manager (SCM[14]) which is the system element that locates the code for that CLSID. The code may exist as a DLL or EXE on the same machine or on another machine: the SCM isolates most of COM, as well as all applications, from the specific actions necessary to locate code. We'll return a discussion of the SCM in a moment after examining the roles of the client and server applications.

.2 COM Clients

Whatever application passes a CLSID to COM and asks for an instantiated object in return is a COM Client. Of course, since this client uses COM, it is also a COM application that must perform the required steps described above and in subsequent chapters.

Regardless of the type of server in use (in-process, local, or remote), a COM Client always asks COM to instantiate objects in exactly the same manner. The simplest method for creating one object is to call the COM function CoCreateInstance. This creates one object of the given CLSID and returns an interface pointer of whatever type the client requests. Alternately, the client can obtain an interface pointer to what is called the "class factory" object for a CLSID by calling CoGetClassObject. This class factory supports an interface called IClassFactory through which the client asks that factory to manufacture an object of its class. At that point the client has interface pointers for *two separate objects*, the class factory and an object of that class, that each have their own reference counts. It's an important distinction that is illustrated in Figure 2-3 and clarified further in Chapter 5.

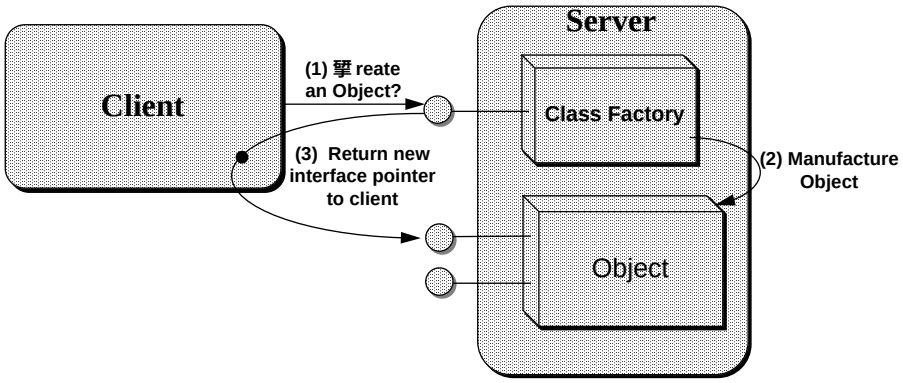


Figure 2-3: A COM Client creates objects through a class factory.

The CoCreateInstance function internally calls CoGetClassObject itself. It's just a more convenient function for clients that want to create one object.

The bottom line is that a COM Client, in addition to its responsibilities as a COM application, is responsible to use COM to obtain a class factory, ask that factory to create an object, initialize the object, and to call that object's (and the class factory's) Release function when the client is finished with it. These steps are the bulk of Chapter 5 which also explains some features of COM that allow clients to manage when servers are loaded and unloaded to optimize performance.

.3 COM Servers

There are two basic kinds of object servers:

- **Dynamic Link Library (DLL) Based:** The server is implemented in a module that can be loaded into, and will execute within, a client's address space. (The term DLL is used in this specification to describe any shared library mechanism that is present on a given COM platform.)
- **EXE Based:** The server is implemented as a stand-alone executable module.

Since COM allows for distributed objects, it also allows for the two basic kinds of servers to implemented on a remote machine. To allow client applications to activate remote objects, COM defines the Service Control Manager (SCM) whose role is described below under "The COM Library."

[14] Colloquially, of course, pronounced "scum."

As a client is responsible for using a class factory and for server management, a server is responsible for implementing the class factory, implementing the class of objects that the factory manufactures, exposing the class factory to COM, and providing for unloading the server under the right conditions. A diagram illustrating what exists inside a server module (EXE or DLL) is shown in Figure 2-4.

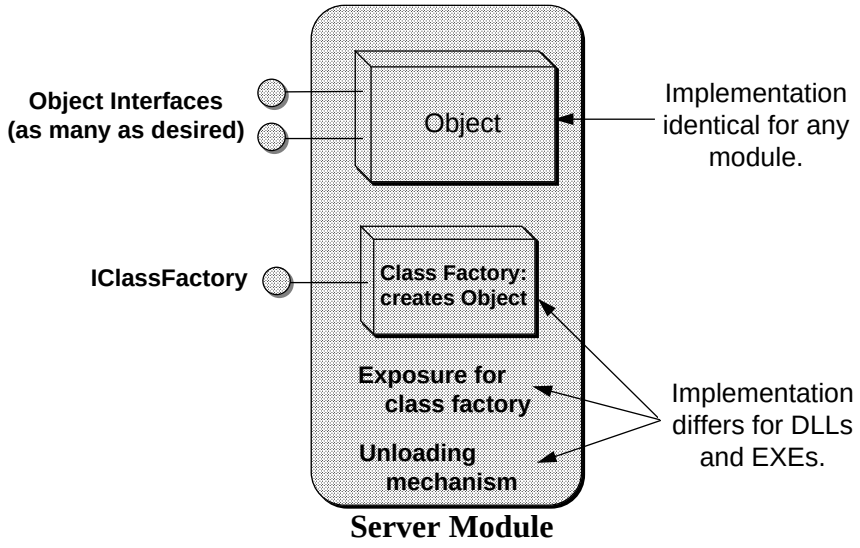


Figure 2-4: The general structure of a COM server.

How a server accomplishes these requirements depends on whether the server is implemented as a DLL or EXE, but is independent of whether the server is on the same machine as the client or on a remote machine. That is, remote servers are the same as local servers but have been registered to be visible to remote clients. Chapter 6 goes into all the necessary details about these implementations as well as how the server publishes its existence to COM in the registration database.

A special kind of server is called an "custom object handler" that works in conjunction with a local server to provide a partial in-process implementation of an object class.[15] Since in-process code is normally much faster to load, in-process calls are extremely fast, and certain resources can be shared only within a single process space, handlers can help improve performance of general object operations as well as the quality of operations such as printing. An object handler is architecturally similar to an in-process server but with more specialized semantics for its use. While the client can control the loading of handlers, it doesn't have to do any special work whatsoever to work with them. The existence of a handler changes nothing for clients.

.4 The COM Library and Service Control Manager

As described in Chapter 1, the COM Library itself is the implementation of the standard API functions defined in COM along with support for communicating between objects and clients. The COM Library is then the underlying "plumbing" that makes everything work transparently through RPC as shown in Figure 2-5 (this the same figure as Figure 1-8 in Chapter 1, repeated here for convenience). Whenever COM determines that it has to establish communication between a client and a local or remote server, it creates "proxy" objects that act as in-process objects to the client. These proxies then talk to "stub" objects that are in the same process as the server and can call the server directly. The stubs pick up RPC calls from the proxies, turn them into function calls to the real object, then pass the return values back to the proxy via RPC which in turn returns them to the client.[16] The underlying remote procedure call mechanism is based on the standard DCE remote procedure call mechanism.

---

[15] Strictly speaking, the "handler" is simply the representative of a remote object that resides in the client's process and which internally contains the remote connection. There is thus *always* a handler present when remoting is being done, though very often the handler is a trivial one which merely forwards all calls. In that sense, "handler" is synonymous with the terms "proxy object" or "object proxy." In practice the term "handler" tends to be used more when there is in fact a non-trivial handler, with "proxy" usually used when the handler is in fact trivial.

[16] Readers more familiar with RPC than with COM will recognize "client stub" and "server stub" rather than "proxy" and "stub" but the phrases are analogous.

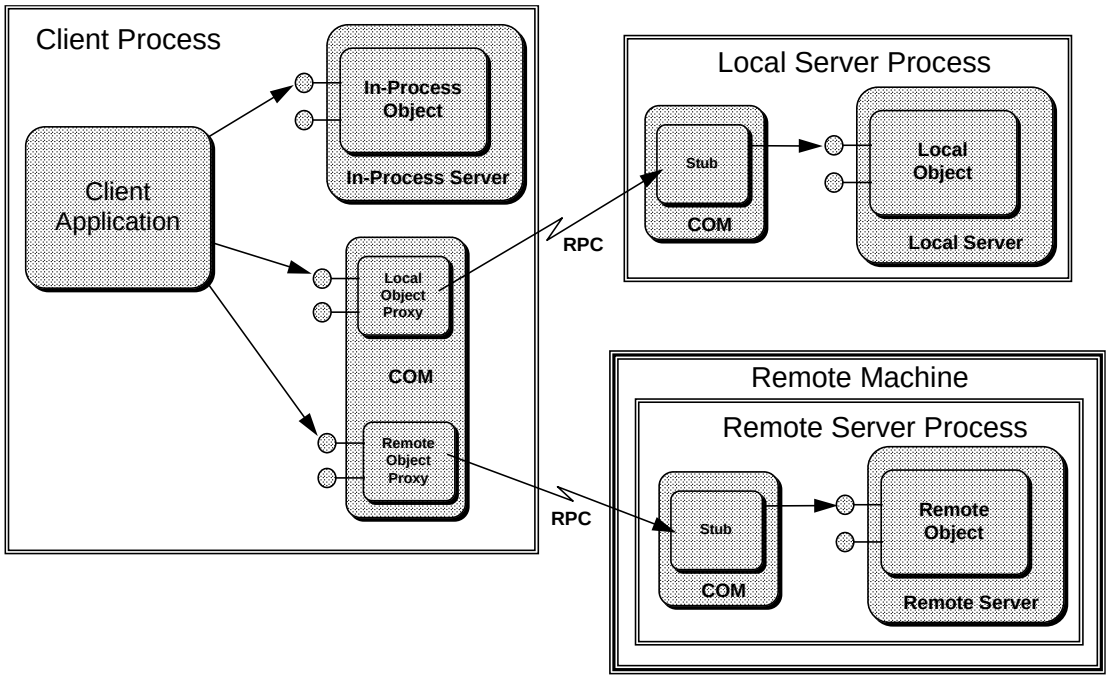


Figure 2-5: COM provides transparent access to local and remote servers through proxy and stub objects.

.5 Architecture for Distributed Objects

The COM architecture for object distribution is similar to the remoting architecture. When a client wants to connect to a server object, the name of the server is stored in the system registry. With distributed objects, the server can implemented as an in-process DLL, a local executable, or as executable or DLL running remotely. A component called the Service Control Manager (SCM) is responsible for locating the server and running it. The next section, "The Service Control Manager", explains the role of the SCM in greater depth and Chapter 15 contains the specification for it's interfaces.

Making a call to an interface method in a remote object involves the cooperation of several components. The interface proxy is a piece of interface-specific code that resides in the client's process space and prepares the interface parameters for transmittal. It packages, or marshals, them in such a way that they can be recreated and understood in the receiving process. The interface stub, also a piece of interface-specific code, resides in the server's process space and reverses the work of the proxy. The stub unpackages, or unmarshals, the sent parameters and forwards them on to the server. It also packages reply information to send back to the client.

The actual transmitting of the data across the network is handled by the RPC runtime library and the channel, part of the COM library. The channel works transparently with different channel types and supports both single and multi-threaded applications.

The flow of communication between the components involved in interface remoting is shown in Figure 2-6. On the client side of the process boundary, the client's method call goes through the proxy and then onto the channel. Note that the channel is part of the COM library. The channel sends the buffer containing the marshaled parameters to the RPC runtime library who transmits it across the process boundary. The RPC runtime and the COM libraries exist on both sides of the process.

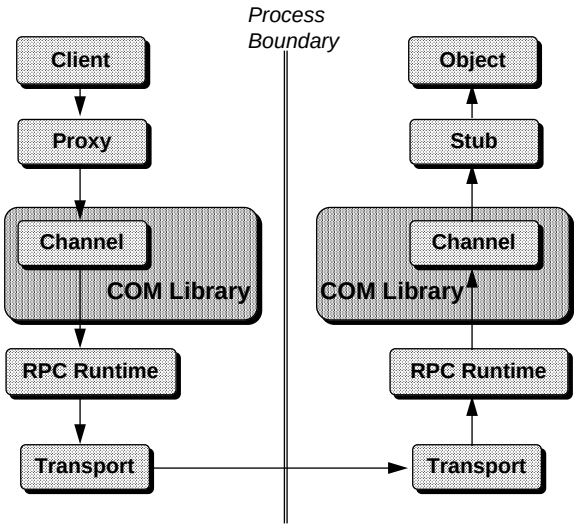


Figure 2-6. Components of COM's distributed architecture.

.6 The Service Control Manager

The Service Control Manager ensures that when a client request is made, the appropriate server is connected and ready to receive the request. The SCM keeps a database of class information based on the system registry that the client caches locally through the COM library. This is the basis for COM's implementation locator services as shown in Figure 2-7.

When a client makes a request to create an object of a CLSID, the COM Library contacts the local SCM (the one on the same machine) and requests that the appropriate server be located or launched, and a class factory returned to the COM Library. After that, the COM Library, or the client, can ask the class factory to create an object.

The actions taken by the local SCM depend on the type of object server that is registered for the CLSID:

In-Process The SCM returns the file path of the DLL containing the object server implementation. The COM library then loads the DLL and asks it for its class factory interface pointer.

Local The SCM starts the local executable which registers a class factory on startup. That pointer is then available to COM.

Remote The local SCM contacts the SCM running on the appropriate remote machine and forwards the request to the remote SCM. The remote SCM launches the server which registers a class factory like the local server with COM on that remote machine. The remote SCM then maintains a connection to that class factory and returns an RPC connection to the local SCM which corresponds to that remote class factory. The local SCM then returns that connection to COM which creates a class factory proxy which will internally forward requests to the remote SCM via the RPC connection and thus on to the remote server.

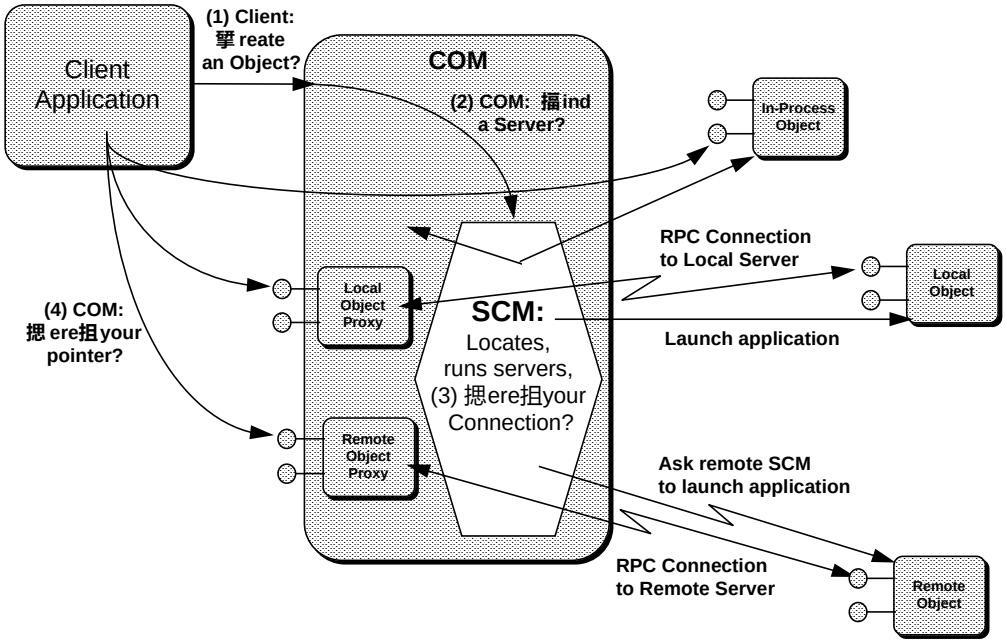


Figure 2-7: COM delegates responsibility of loading and launching servers to the SCM.

Note that if the remote SCM determines that the remote server is actually an in-process server, it launches a "surrogate" server that then loads that in-process server. The surrogate does nothing more than pass all requests on through to the loaded DLL.

.7 Application Security

The technology in COM provides security for applications, regardless of whether they run remotely. There is a default level of security that is provided to non-security-aware applications such as existing OLE applications. Beyond the default, applications that are security-aware can control who is granted access to their services and the type of access that is granted.

Default security insures that system integrity is maintained. When multiple users require the services of a single non-security-aware server, a separate instance for each user is run. Each client/server connection remains independent from the others, preventing clients from accessing each others' data. All non-security-aware servers are run as the security principal who caused them to run. An example involving four clients that all require server "X" is illustrated in Figure 2-8. Since two of the clients are the same user (User2), one instance of server X can service both clients.

The technology used in COM for distribution implements this security system with the authentication services provided by RPC. These services are accessed by applications through the COM library when a call is made to CoInitialize. This security system imposes a restriction on where non-security-aware applications can run. Since the system cannot start a session on another machine without the proper credentials, all servers that run in the client security context normally run where their client is running. The AtBits attribute associated with that class controls where a server is run.

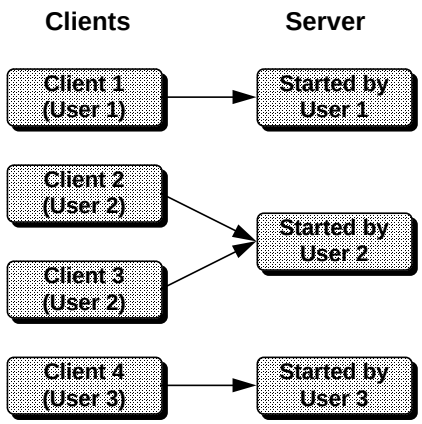


Figure 2-8. A non-security-aware server

Security-aware servers are those applications that do not allow global access to their services. These servers may run either where the client is running, where their data is stored, or elsewhere depending on a rich set of activation rules. Rather than running as one of their clients; security-aware servers are themselves security principals. Security-aware servers may participate in two-way authentication whereby clients can ask for verification. Securi-

ty-aware servers can use the services offered by the RPC security provider(s) or supply their own security implementation.

4 Object Reusability

An important goal of any object model is that component authors can reuse and extend objects provided by others as pieces of their own component implementations. Implementation inheritance is one way this can be achieved: to reuse code in the process of building a new object, you inherit implementation from it and override methods in the tradition of C++ and other languages. However, as a result of many years experience, many people believe traditional language-style implementation inheritance technology as the basis for object reuse is simply not robust enough for large, evolving systems composed of software components. (See page 16 for more information.) For this reason COM introduces other reusability mechanisms.

.1 COM Reusability Mechanisms

The key point to building reusable components is black-box reuse which means the piece of code attempting to reuse another component knows nothing, and does not need to know anything, about the internal structure or implementation of the component being used. In other words, the code attempting to reuse a component depends upon the *behavior* of the component and not the exact *implementation*.

To achieve black-box reusability, COM supports two mechanisms through which one object may reuse another. For convenience, the object being reused is called the "inner object" and the object making use of that inner object is the "outer object."

1. **Containment/Delegation**: the outer object behaves like an object client to the inner object. The outer object "contains" the inner object and when the outer object wishes to use the services of the inner object the outer object simply delegates implementation to the inner object's interfaces. In other words, the outer object uses the inner's services to implement itself. It is not necessary that the outer and inner objects support the same interfaces; in fact, the outer object may use an inner object's interface to help implement parts of a different interface on the outer object especially when the complexity of the interfaces differs greatly.

2. **Aggregation**: the outer object wishes to expose interfaces from the inner object as if they were implemented on the outer object itself. This is useful when the outer object would always delegate every call to one of its interfaces to the same interface of the inner object. Aggregation is a convenience to allow the outer object to avoid extra implementation overhead in such cases.

These two mechanisms are illustrated in Figures 2-9 and 2-10. The important part to both these mechanisms is how the outer object appears to its clients. As far as the clients are concerned, both objects implement interfaces *A*, *B*, and *C*. Furthermore, the client treats the outer object as a black box, and thus does not care, nor does it need to care, about the internal structure of the outer object—the client only cares about behavior.

Containment is simple to implement for an outer object: during its creation, the outer object creates whatever inner objects it needs to use as any other client would. This is nothing new—the process is like a C++ object that itself contains a C++ string object that it uses to perform certain string functions even if the outer object is not considered a "string" object in its own right.

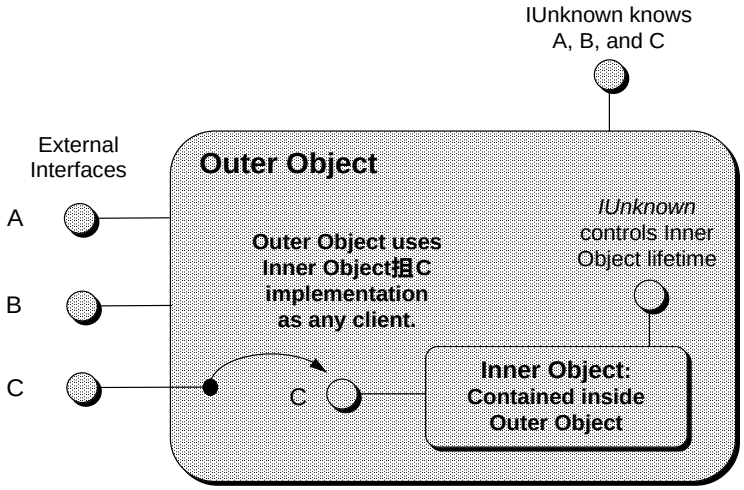


Figure 2-9: Containment of an inner object and delegation to its interfaces.

Aggregation is almost as simple to implement, the primary difference being the implementation of the three IUnknown functions: QueryInterface, AddRef, and Release. The catch is that from the client's perspective, any IUnknown function on the outer object must affect the outer object. That is, AddRef and Release affect the outer object and QueryInterface exposes all the interfaces available on the outer object. However, if the outer object simply exposes an inner object's interface as it's own, that inner object's IUnknown members called through that interface will behave differently than those IUnknown members on the outer object's interfaces, a sheer violation of the rules and properties governing IUnknown.

The solution is for the outer object to somehow pass the inner object some IUnknown pointer to which the inner object can re-route (that is, delegate) IUnknown calls in its own interfaces, and yet there must be a method through which the outer object can access the inner object's IUnknown functions that only affect the inner object. COM provides specific support for this solution as described in Chapter 6.

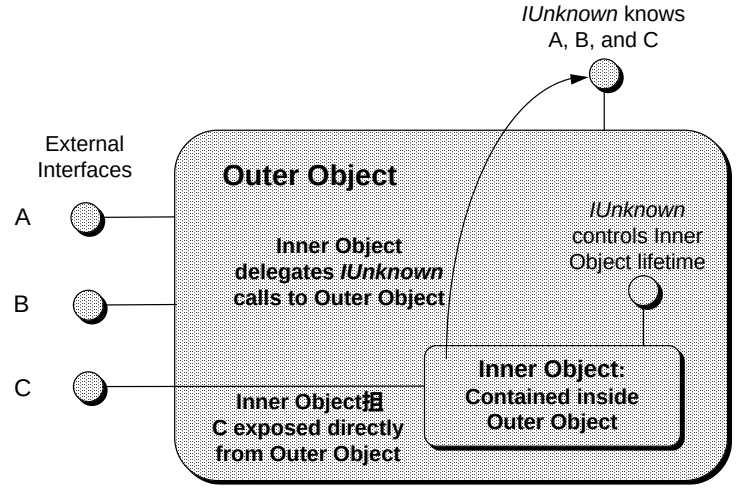


Figure 2-10: Aggregation of an inner object where the outer object exposes one or more of the inner object's interfaces as it's own.

5 Connectable Objects and Events

In the preceding discussions of interfaces it was implied that, from the object's perspective, the interfaces were "incoming". "Incoming," in the context of a client-object relationship, implies that the object "listens" to what the client has to say. In other words, incoming interfaces and their member functions receive input from the outside. COM also defines mechanisms where objects can support "outgoing" interfaces. Outgoing interfaces allow objects to have two-way conversations, so to speak, with clients. When an

object supports one or more outgoing interfaces, it is said to be *connectable.* One of the most obvious uses for outgoing interfaces is for event notification. This section describes Connectable Objects.[17]

A connectable object (also called a *source*) can have as many outgoing interfaces as it likes. Each interface is composed of distinct member functions, with each function representing a single *event, notification,* or *request.* Events and notifications are equivalent concepts (and interchangeable terms), as they are both used to tell the client that something interesting happened in the object. Events and notifications differ from a request in that the object expects response from the client. A request, on the other hand, is how an object asks the client a question and expects a response.

In all of these cases, there must be some client that listens to what the object has to say and uses that information wisely. It is the client, therefore, that actually implements these interfaces on objects called *sinks.* From the sink's perspective, the interfaces are incoming, meaning that the sink listens through them. A connectable object plays the role of a client as far as the sink is concerned; thus, the sink is what the object's client uses to listen to that object.

An object doesn't necessarily have a one-to-one relationship with a sink. In fact, a single instance of an object usually supports any number of connections to sinks in any number of separate clients. This is called *multicasting.*[18] In addition, any sink can be connected to any number of objects.

Chapter 11 covers the Connectable Object interfaces (IConnectionPoint and IConnectionPointContainer) in complete detail.

6 Persistent Storage

As mentioned in Chapter 1, the enhanced COM services define a number of storage-related interfaces, collectively called Persistent Storage or Structured Storage. By definition of the term *interface*, these interfaces carry no implementation. They describe a way to create a "file system within a file," and they provide some extremely powerful features for applications including incremental access, transactioning, and a sharable medium that can be used for data exchange or for storing the persistent data of objects that know how to read and write such data themselves. The following sections deal with the structure of storage and the other features.

.1 A File System Within A File

Years ago, before there were "disk operating systems," applications had to write persistent data directly to a disk drive (or drum) by sending commands directly to the hardware disk controller. Those applications were responsible for managing the absolute location of the data on the disk, making sure that it was not overwriting data that was already there. This was not too much of a problem seeing as how most disks were under complete control of a single application that took over the entire computer.

The advent of computer systems that could run more than one application brought about problems where all the applications had to make sure they did not write over each other's data on the disk. It therefore became beneficial that each adopted a standard of marking the disk sectors that were used and which ones were free. In time, these standards became the "disk operating system" which provided a "file system." Now, instead of dealing directly with absolute disk sectors and so forth, applications simply told the file system to write blocks of data to the disk. Furthermore, the file system allowed applications to create a hierarchy of information using directories which could contain not only files but other sub-directories which in turn contained more files, more sub-directories, etc.

The file system provided a single level of indirection between applications and the disk, and the result was that every application saw a file as a single contiguous stream of bytes on the disk. Underneath, however, the file system was storing the file in dis-contiguous sectors according to some algorithm that optimized read and write time for each file. The indirection provided from the file system freed applications from having to care about the absolute position of data on a storage device.

Today, virtually all system APIs for file input and output provide applications with some way to write information into a flat file that applications see as a single stream of bytes that can grow as large as necessary

[17] OLE Controls use the Connectable Objects mechanisms extensively.

[18] Note that this usage of the term *multicasting* may differ from what some readers are accustomed to. In some systems *multicasting* is used to describe a connection-less broadcast. Connectable objects are obviously connection oriented.

until the disk is full. For a long time these APIs have been sufficient for applications to store their persistent information. Applications have made some incredible innovations in how they deal with a single stream of information to provide features like incremental "fast" saves.

However, a major feature of COM is interoperability, the basis for integration between applications. *This integration brings with it the need to have multiple applications write information to the same file on the underlying file system.* This is exactly the same problem that the computer industry faced years ago when multiple applications began to share the same disk drive. The solution then was to create a file system to provide a level of indirection between an application "file" and the underlying disk sectors.

Thus, the solution for the integration problem today is another level of indirection: a file system *within* a file. Instead of requiring that a large contiguous sequence of bytes on the disk be manipulated through a single file handle with a single seek pointer, COM defines how to treat a single file system entity as a structured collection of two types of objects—storages and streams—that act like directories and files, respectively.

.2 Storage and Stream Objects

Within COM's Persistent Storage definition there are two types of storage elements: storage objects and stream objects. These are objects generally implemented by the COM library itself; applications rarely, if ever, need to implement these storage elements themselves.[19] These objects, like all others in COM, implement interfaces: IStream for stream objects, IStorage for storage objects as detailed in Chapter 8.

A stream object is the conceptual equivalent of a single disk file as we understand disk files today. Streams are the basic file-system component in which data lives, and each stream in itself has access rights and a single seek pointer. Through its *IStream* interface stream can be told to read, write, seek, and perform a few other operations on its underlying data. Streams are named by using a text string and can contain any internal structure you desire because they are simply a flat stream of bytes. In addition, the functions in the *IStream* interface map nearly one-to-one with standard file-handle based functions such as those in the ANSI C run-time library.

A storage object is the conceptual equivalent of a directory. Each storage, like a directory, can contain any number of sub-storages (sub-directories) and any number of streams (files). Furthermore, each storage has its own access rights. The IStorage interface describes the capabilities of a storage object such as enumerate elements (dir), move, copy, rename, create, destroy, and so forth. A storage object itself cannot store application-defined data except that it implicitly stores the names of the elements (storages and streams) contained within it.

Storage and stream objects, when implemented by COM as a standard on a system, are sharable between processes. This is a key feature that enables objects running in-process or out-of-process to have equal incremental access to their on-disk storage. Since COM is loaded into each process separately, it must use some operating-system supported shared memory mechanisms to communicate between processes about opened elements and their access modes.

.3 Application Design with Structured Storage

COM's structured storage built out of storage and stream objects makes it much easier to design applications that by their nature produce structured information. For example, consider a "diary" program that allows a user to make entries for any day of any month of any year. Entries are made in the form of some kind of object that itself manages some information. Users wanting to write some text into the diary would store a text object; if they wanted to save a scan of a newspaper clip they could use a bitmap objects, and so forth.

[19] This specification recommends that the COM implementation on a given platform (Windows, Macintosh, etc.) includes a standard storage implementation for use by all applications.

Without a powerful means to structure information of this kind, the diary application might be forced to manage some hideous file structure with an overabundance of file position cross-reference pointers as shown in Figure 2-11.

There are many problems in trying to put structured information into a flat file. First, there is the sheer tedium of managing all the cross-reference pointers in all the different structures of the file. Whenever a piece of information grows or moves in the file, every cross-reference offset referring to that information must be updated as well. Therefore even a small change in the size of one of the text objects or an addition of a day or month might precipitate changes throughout the rest of the file to update seek offsets. While not only tedious to manage, the application will have to spend enormous amounts of time moving information around in the file to make space for data that expands. That, or the application can move the newly enlarged data to the end of the file and patch a few seek offsets, but that introduces the whole problem of garbage collection, that is, managing the free space created in the middle of the file to minimize waste as well as overall file size.

The problems are compounded even further with objects that are capable of reading and writing their own information to storage. In the example here, the diary application would prefer to give each objects in it—text, bitmap, drawing, table, etc.—its own piece of the file in which the object can write whatever the it wants, however much it wants. The only practical way to do this with a single flat file is for the diary application to ask each object for a memory copy of what the object would like to store, and then the diary would write that information into a place in its own file. This is really the only way in which the diary could manage the location of all the information. Now while this works reasonably well for small data, consider an object that wants to store a 10MB bitmap scan of a true-color photograph—exchanging that much data through memory is horribly inefficient. Furthermore, if the end user wants to later make changes to that bitmap, the diary would have to load the bitmap *in entirety* from its file and pass it back to the object. This is again extraordinarily inefficient.[20]

COM's Persistent Storage technology solves these problems through the extra level of indirection of a file

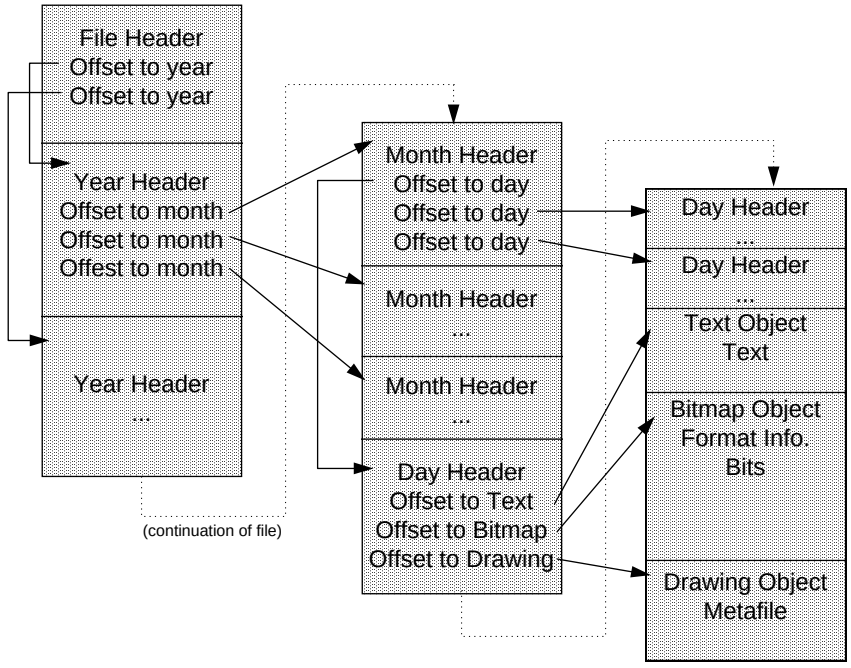


Figure 2-11: A flat-file structure for a diary application. This sort of structure is difficult to manage.

system within a file. With COM, the diary application can create a structured hierarchy where the root file itself has sub-storages for each year in the diary. Each year sub-storage has a sub-storage for each month, and each month has a sub-storage for each day. Each day then would have yet another sub-storage or per-

[20] This mechanism, in fact, was employed by compound documents in Microsoft's OLE version 1.0. The problems describe here were some of the major limitations of OLE 1.0 which provided much of the impetus for COM's Persistent Storage technology.

haps just a stream for each piece of information that the user stores in that day.[21] This configuration is illustrated in Figure 2-12.

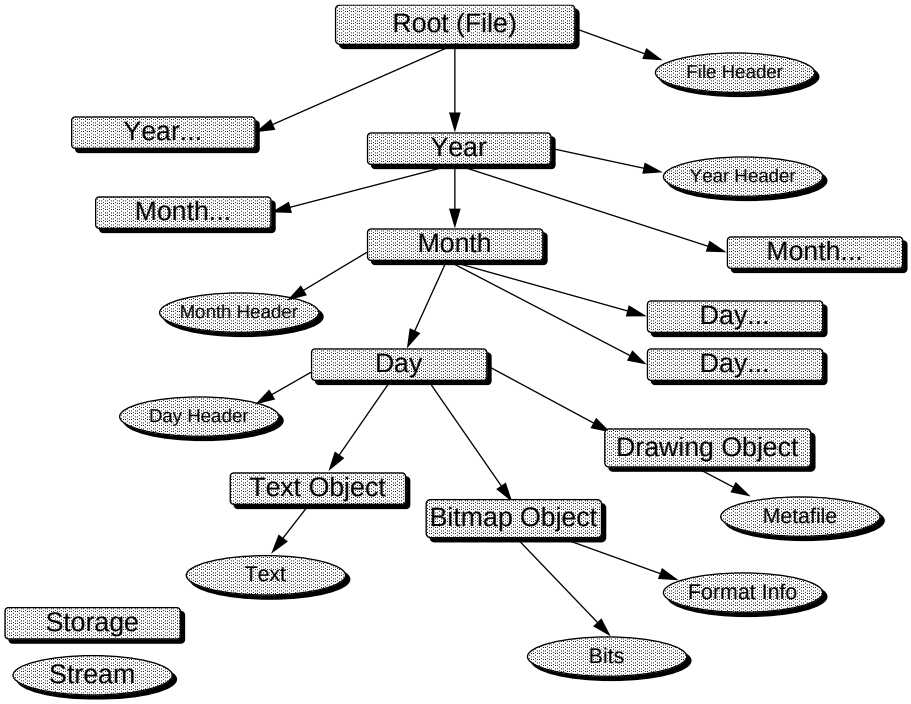


Figure 2-12: A structured storage scheme for a diary application. Every object that has some content is given its own storage or stream element for its own exclusive use.

This structure solves the problem of expanding information in one of the objects: the object itself expands the streams in its control and the COM implementation of storage figures out where to store all the information in the stream. The diary application doesn't have to lift a finger. Furthermore, the COM implementation automatically manages unused space in the entire file, again, relieving the diary application of a great burden.

In this sort of storage scheme, the objects that manage the content in the diary always have direct *incremental* access to their piece of storage. That is, when the object needs to store its data, it writes it *directly* into the diary file without having to involve the diary application itself. The object can, if it wants to, write *incremental changes* to that storage, thus leading to much better performance than the flat file scheme could possibly provide. If the end user wanted to make changes to that information later on, the object can then incrementally read as little information as necessary instead of requiring the diary to read all the information into memory first. Incremental access, a feature that has traditionally been very hard to implement in applications, is now the *default mode of operation*. All of this leads to much better performance.

.4 Naming Elements

Every storage and stream object in a structured file has a specific character name to identify it. These names are used to tell *IStorage* functions what element in that storage to open, destroy, move, copy, rename, etc. Depending on which component, client or object, actually defines and stores these names, different conventions and restrictions apply.

Names of root storage objects are in fact names of files in the underlying file system. Thus, they obey the conventions and restrictions that it imposes. Strings passed to storage-related functions which name files are passed on un-interpreted and unchanged to the file system.

Names of elements contained within storage objects are managed by the implementation of the particular storage object in question. All implementations of storage objects must at the least support element names

[21] The application would only create year, month, and day substorages for those days that had information in them, that is, the diary application would create sparse storage for efficiency.

that are 32 characters in length; some implementations may if they wish choose to support longer names. Names are stored case-preserving, but are compared case-insensitive.[22] As a result, applications which define element names must choose names which will work in either situation.

The names of elements inside an storage object must conform to certain conventions:

1. The two specific names "." and ".." are reserved for future use.
2. Element names cannot contain any of the four characters "\", "/", ":", or "!".

In addition, the name space in a storage element is partitioned in to different areas of ownership. Different pieces of code have the right to create elements in each area of the name space.

- The set of element names beginning with characters other than '\0x01' through '\0x1F' (that is, decimal 1 through decimal 31) are for use by the object whose data is stored in the *IStorage*. Conversely, the object must *not* use element names beginning with these characters.
- Element names beginning with a '\0x01' and '\0x02' are for the exclusive use of COM and other system code built on it such as OLE Documents.
- Element names beginning with a '\0x03' are for the exclusive use of the client which is managing the object. The client can use this space as a place to persistently store any information it wishes to associate with the object along with the rest of the storage for that object.
- Element names beginning with a '\0x04' are for the exclusive use of the COM structured storage implementation itself. They will be useful, for example, should that implementation support other interfaces in addition to IStorage, and these interface need persistent state.
- Element names beginning with '\0x05' and '\0x06' are for the exclusive use of COM and other system code built on it such as OLE Documents.
- All other names beginning with '\0x07' through '\0x1F' are reserved for future definition and use by the system.

In general, an element's name is not considered useful to an end-user. Therefore, if a client wants to store specific user-readable names of objects, it usually uses some other mechanism. For example, the client may write its own stream under one of its own storage elements that has the names of all the other objects within that same storage element. Another method would be for the client to store a stream named "\0x03Name" in each object's storage that would contain that object's name. Since the stream name itself begins with '\0x03' the client owns that stream even through the objects controls much of the rest of that storage element.

.5 Direct Access vs. Transacted Access

Storage and stream elements support two fundamentally different modes of access: direct mode and transacted mode. Changes made while in direct mode are immediately and permanently made to the affected storage object. In transacted mode, changes are buffered so that they may be saved ("committed") or reverted when modifications are complete.

If an outermost level IStorage is used in transacted mode, then when it commits, a robust two-phase commit operation is used to publish those changes to the underlying file on the file system. That is, great pains are taken are taken so as not to loose the user's data should an untimely crash occurs.

The need for transacted mode is best explained by an illustrative scenario. Imagine that a user has created a spreadsheet which contains a sound clip object, and that the sound clip is an object that uses the new persistent storage facilities provided in COM. Suppose the user opens the spreadsheet, opens the sound clip, makes some editing changes, then closes the sound clip at which point the changes are updated in the spreadsheet storage set aside for the sound clip. Now, at this instant, the user has a choice: save the spreadsheet or close the spreadsheet *without* saving. Either way, the next time the user opens the spreadsheet, the sound clip had better be in the appropriate state. This implies that at the instant before the save vs. close decision was made, both the old and the new versions of the sound clip had to exist. Further, since large

[22] Case sensitivity is a locale-sensitive operation: some characters compare case-insenstive-equal in some locales and -not-equal in others. In an IStorage implementation, the case-insenstive comparision is done with respect to the current locale in which the system is presently running. This has implications on the use of IStorage names for those who wish to create globally portable documents.

objects are precisely the ones that are expensive in time and space to copy, the new version should exist as a set of *differences* from the old.

The central issue is whose responsibility it is to keep track of the two versions. The client (the spreadsheet in this example) had the old version to begin with, so the question really boils down to how and when does the object (sound clip) communicate the new version to the spreadsheet. Applications today are in general already designed to keep edits separate from the persistent copy of an object until such time as the user does a save or update. Update time is thus the earliest time at which the transfer should occur. The latest is immediately before the client saves itself. The most appropriate time seems to be one of these two extremes; no intermediate time has any discernible advantage.

COM specifies that this communication happens at the earlier time. When asked to update edits back to the client, an object using the new persistence support will write any changes to its storage) exactly as if it were doing a save to its own storage completely outside the client. It is the responsibility of the client to keep these changes separate from the old version until *it* does a save (commit) or close (revert). Transacted mode on *IStorage* makes dealing with this requirement easy and efficient.

The transaction on each storage is nested in the transaction of its parent storage. Think of the act of committing a transaction on an IStorage instance as "publishing changes one more level outwards." Inner objects publish changes to the transaction of the next object outwards; outermost objects publish changes permanently into the file system.

Let's examine for a moment the implications of using instead the second option, where the object keeps all editing changes to itself until it is known that the user wants to commit the client (save the file). This may happen many minutes after the contained object was edited. COM must therefore allow for the possibility that in the interim time period the user closed the server used to edit the object, since such servers may consume significant system resources. To implement this second option, the server must presumably keep the changes to the old version around in a set of temporary files (remember, these are potentially *big* objects). At the client's commit time, every server would have to be restarted and asked to incorporate any changes back onto its persistent storage. This could be *very* time consuming, and could significantly slow the save operation. It would also cause reliability concern in the user's mind: what if for some reason (such as memory resources) a server cannot be restarted? Further, even when the client is closed *without* saving, servers have to be awakened to clean up their temporary files. Finally, if a object is edited a second time before the client is committed, in this option its the client can only provide the *old*, *original* storage, not the storage that has the first edits. Thus, the server would have to recognize on startup that some edits to this object were lying around in the system. This is an awkward burden to place on servers: it amounts to requiring that they *all* support the ability to do incremental auto-save with automatic recovery from crashes. In short, this approach would significantly and unacceptably complicate the responsibilities of the object implementors.

To that end, it makes the most sense that the standard COM implementation of the storage system support transactioning through IStorage and possibly IStream.

.6 Browsing Elements

By its nature, COM's structured storage separates applications from the exact layout of information within a given file. Every element of information in that file is access using functions and interfaces implemented by COM. Because this implementation is central, a file generated by some application using this structure can be browsed by some other piece of code, such as a system shell. In other words, any piece of code in the system can use COM to browse the entire hierarchy of elements within any structured file simply by navigating with the IStorage interface functions which provide directory-like services. If that piece of code also knows the format and the meaning of a specific stream that has a certain name, it could also *open* that stream and make use of the information in it, *without having to run the application that wrote the file.*

This is a powerful enabling technology for operating system shells that want to provide rich query tools to help end users look for information on their machine or even on a network. To make it really happen requires standards for certain stream names and the format of those streams such that the system shell can open the stream and execute queries against that information. For example, consider what is possible if all applications created a stream called "Summary Information" underneath the root storage element of the file. In this stream the application would write information such as the author of the document, the create/modify/last saved time-stamps, title, subject, keywords, comments, a thumbnail sketch of the first page,

etc. Using this information the system shell could find any documents that a certain user write before a certain date or those that contained subject matter matched against a few keywords. Once those documents are found, the shell can then extract the title of the document along with the thumbnail sketch and give the user a very engaging display of the search results.

This all being said, in the general the actual utility of this capability is perhaps significantly less than what one might first imagine. Suppose, for example, that I have a structured storage that contains some word processing document whose semantics and persistent representation I am unaware of, but which contains some number of contained objects, perhaps the figures in the document, that I can identify by their being stored and tagged in contained sub-storages. One might naively think that it would be reasonable to be able to walk in and browse the figures from some system-provided generic browsing utility. This would indeed work from a technical point of view; however, it is unlikely to be useable from a user interface perspective. The document may contain hundreds of figures, for example, that the user created and thinks about not with a name, not with a number, but only in the relationship of a particular figure to the rest of the document's information. *With what user interface could one reasonably present this list of objects to the user other than as some add-hoc and arbitrarily-ordered sequence?* There is, for example, no name associated with each object that one could use to leverage a file-system directory-browsing user interface design. In general, the *content* of a document can only be reasonably be presented to a human being using a tool that understands the semantics of the document content, and thus can show all of the information therein in its appropriate context.

.7 Persistent Objects

Because COM allows an object to read and write itself to storage, there must be a way through which the client tells objects to do so. The way is, of course, additional interfaces that form a storage contract between the client and objects. When a client wants to tell and object to deal with storage, it queries the object for one of the persistence-related interfaces, as suits the context. The interfaces that objects can implement, in any combination, are described below:

IPersistStorage	Object can read and write its persistent state to a storage object. The client provides the object with an IStorage pointer through this interface. This is the only IPersist* interface that includes semantics for incremental access.
IPersistStream	Object can read and write its persistent state to a stream object. The client provides the object with an IStream pointer through this interface.
IPersistFile	Object can read and write its persistent state to a file on the underlying system directly. This interface does not involve IStorage or IStream unless the underlying file is itself access through these interfaces, but the IPersistFile itself has no semantics relating to such structures. The client simply provides the object with a filename and orders to save or load; the object does whatever is necessary to fulfill the request.

These interfaces and the rules governing them are described in Chapter 12.

7 Persistent, Intelligent Names: Monikers

To set the context for why "Persistent, Intelligent Names" are an important technology in COM, think for a moment about a standard, mundane file name. That file name refers to some collection of data that happens to be stored on disk somewhere. The file name describes the somewhere. In that sense, the file name is really a name for a particular "object" of sorts where the object is defined by the data in the file.

The limitation is that a file name by itself is unintelligent; all the intelligence about what that filename means and how it gets used, as well as how it is stored persistently if necessary, is contained in whatever application is the client of that file name. The file name is nothing more than some piece of data in that client. This means that the client must have specific code to handle file names. This normally isn't seen as much of a problem—most applications can deal with files and have been doing so for a long time.

Now introduce some sort of name that describes a query in a database. The introduce others that describe a file and a specific range of data within that file, such as a range of spreadsheet cells or a paragraph is a document. Introduce yet more than identify a piece of code on the system somewhere that can execute some interesting operation. In a world where clients have to know what a name means in order to use it, those clients end up having to write specific code for each type of name causing that application to grow monolithically in size and complexity. This is one of the problems that COM was created to solve.

In COM, therefore, the intelligence of how to work with a particular name is encapsulated inside the name itself, where the name becomes an object that implements name-related interfaces. These objects are called *monikers.*[23] A moniker implementation provides an abstraction to some underlying connection (or "binding") mechanism. Each different moniker class (with a different CLSID) has its own semantics as to what sort of object or operation it can refer to, which is *entirely* up to the moniker itself. A section below describes some typical types of monikers. While a moniker class itself defines the operations necessary to locate some general type of object or perform some general type of action, each individual moniker *object* (each instantiation) maintains its own name data that identifies some other *particular* object or operation. The moniker class defines the functionality; a moniker object maintains the parameters.

With monikers, clients always work with names through an interface, rather than directly manipulating the strings (or whatever) themselves. This means that whenever a client wishes to perform any operation with a name, it calls some code to do it instead of doing the work itself. This level of indirection means that the moniker can transparently provide a whole host of services, and that the client can seamlessly interoperate over time with various different moniker implementations which implement these services in different ways.

.1 Moniker Objects

A moniker is simply an object that supports the IMoniker interface. IMoniker interface includes the IPersistStream interface;[24] thus, monikers can be saved to and loaded from streams. The persistent form of a moniker includes the data comprising its name and the CLSID of its implementation which is used during the loading process. This allows new kinds of monikers to be created transparently to clients.

The most basic operation in the IMoniker interface is that of *binding* to the object to which it points. The binding function in IMoniker takes as a parameter the interface identifier by which the client wishes to talk to the bound object, runs whatever algorithm is necessary in order to locate the object, then returns a pointer of that interface type to the client. The client can also ask to bind to the object's *storage* (for example, the IStorage containing the object) if desired, instead of to the running object through a slightly different IMoniker function. As binding may be an expensive and time-consuming process, a client can control how long it is willing to wait for the binding to complete. Binding also takes place inside a specific "bind context" that is given to the moniker. Such a context enables the binding process overall to be more efficient by avoiding repeated connections to the same object.

A moniker also supports an operation called "reduction" through which it re-writes itself into another equivalent moniker that will bind to the same object, but does so in a more efficient way. This capability is useful to enable the construction of user-defined macros or aliases as new kinds of moniker classes (such that when reduced, the moniker to which the macro evaluates is returned) and to enable construction of a kind of moniker which tracks data as it moves about (such that when reduced, the new moniker contains a reference to the new location). Chapter 9 will expand on the reduction concept.

Each moniker class can store arbitrary data its persistent representation, and can run arbitrary code at binding time. The client therefore only knows each moniker by the presence of a persistent representation and whatever label the client wishes to assign to each moniker. For example, a spreadsheet as a client may keep, from the user's perspective, a list of "links" to other spreadsheets where, in fact, each link was an arbitrary label for a moniker (regardless of whether the moniker is loaded or persistently on disk at the moment) where the moniker manages the real identity of the linked data. When the spreadsheet wants to resolve a link for the user, it only has to ask the moniker to bind to the object. After the binding is complete, the

[23] The word "moniker" is fairly obscure synonym for "nickname."

[24] One of the few instances of inheritance from one major interface to another, which the *IMoniker* designer later decided was actually less preferable to having a moniker implement *IMoniker* and *IPersistStream* separately. See the first footnote in Chapter 9.

spreadsheet then has an interface pointer for the linked object and can talk to it directly—the moniker falls out of the picture as its job is complete.

The label assigned to a moniker by a client does not have to be arbitrary. Monikers support the ability to produce a "display name" for whatever object they represent that is suitable to show to an end user. A moniker that maintains a file name (such that it can find an application to load that file) would probably just use the file name directly as the display name. Other monikers for things such as a query may want to provide a display name that is a little more readable than some query languages.

.2 Types of Monikers

As some of the examples above has hinted, monikers can have many types, or classes, depending on the information they contain and the type of objects they can refer to. A moniker class is really defined by the information it persistently maintains and the binding operation is uses on that information.

COM itself, however, only specifies one standard moniker called the *generic composite moniker*. The composite moniker is special in two ways. First, its persistent data is *completely* composed of the persistent data of other monikers, that is, a composite moniker is a collection of other monikers. Second, binding a composite moniker simply tells the composite to bind each moniker it contains in sequence. Since the composite's behavior and persistent state is defined by other monikers, it is a standard type of moniker that works identically on any host system; the composite is *generic* because it has no knowledge of its pieces except that they are monikers. Chapter 9 described the generic composite in more detail.

So what other types of monikers can go in a composite? Virtually any other type (including other composite monikers!). However, other types of monikers are not so generic and have more dependency on the underlying operating system or the scenarios in which such a moniker is used.

For example, Microsoft's OLE defines four other specific monikers—file, item, anti, pointer—that it uses specifically to help implement "linked objects" in its compound document technology. A file moniker, for example, maintains a file name as its persistent data and its binding process is one of locating an application that can load that file, launching the application, and retrieving from it an IPersistFile interface through which the file moniker can ask the application to load the file. Item monikers are used to describe smaller portions of a file that might have been loaded with a file moniker, such as a specific sheet of a three-dimensional spreadsheet or a range of cells in that sheet. To "link" to a specific cell range in a specific sheet of a specific file, the single moniker used to describe the link is a generic composite that is composed with a file moniker and two item monikers as illustrated in Figure 2-13. Each moniker in the composite is one step in the path to the final source of the link.

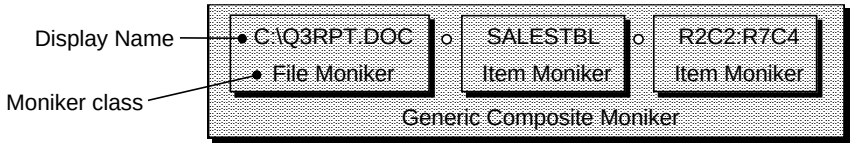


Figure 2-13: A composite moniker that is composed with a file moniker and two item monikers to describe the source of a link which is a cell range in a specific sheet of a spreadsheet file.

More complete descriptions of the file, item, anti, and pointer monikers from OLE are provided in Chapter 9 as examples of how monikers can be used. But monikers can represent virtually any type of information and operation, and are not limited to this basic set of OLE defined monikers.

.3 Connections and Reconnections

How does a client come by a moniker in the first place? In other words, how does a client establish a connection to some object and obtain a moniker that describes that connection? The answer depends on the scenario involved but is generally one of two ways. First, the source of the object may have created a moniker and made it available for consumption through a data transfer mechanism such (in the workstation case) as a clipboard or perhaps a drag & drop operation. Second, the client may have enough knowledge about a particular moniker class that it can synthesize a moniker for some object using other known information such that the client can forget about that specific information itself and thereafter deal only with monikers. So regardless of how a client obtains a moniker, it can simply ask the moniker to bind to establish a connection to the object referred to by the moniker.

Binding a moniker does not always mean that the moniker must run the object itself. The object might already be running within some appropriate scope (such as the current desktop) by the time the client wants to bind the moniker to it. Therefore the moniker need only connect to that running object.

COM supports this scenario through two mechanisms. The first is the *Running Object Table* in which objects register themselves and their monikers when they become running. This table is available to all monikers as they attempt to bind—if a moniker sees that a matching moniker in the table, it can quickly connect to the already running object.

8 Uniform Data Transfer

Just as COM provides interfaces for dealing with storage and object naming, it also provides interfaces for exchanging data between applications. So built on top of both COM and the Persistent Storage technology is Uniform Data Transfer, which provides the functionality to represent all data transfers through a single implementation of a *data object*. Data objects implement an interface called IDataObject which encompasses the standard operations of get/set data and query/enumerate formats as well as functions through which a client of a data object can establish a notification loop to detect data changes in the object. In addition, this technology enables use of richer descriptions of data formats and the use of virtually any storage medium as the transfer medium.

.1 Isolation of Transfer Protocols

The "Uniform" in the name of this technology arose from the fact that the IDataObject interface separates all the common exchange operations from what is called a *transfer protocol*. Existing protocols include facilities such as a "clipboard" or a "drag & drop" feature as well as compound documents as implemented in OLE. With Uniform Data Transfer, all protocols are concerned only with exchanging a pointer to an IDataObject interface. The source of the data—the server—need only implement one data object which is usable in any exchange protocol and that's it. The consumer—the client—need only implement one piece of code to request data from a data object once it receives an IDataObject pointer from any protocol. Once the pointer exchange has occurred, both sides deal with data exchange in a uniform fashion, through IDataObject.

This uniformity not only reduces the code necessary to source or consume data, but also greatly simplifies the code needed to work with the protocol itself. Before COM was first implemented in OLE 2, each transfer protocol available on Microsoft Windows had its own set of functions that tightly bound the protocol to the act of requesting data, and so programmers had to implement specific code to handle each different protocol and exchange procedure. Now that the exchange functionality is separated from the protocol, dealing with each protocol requires only a minimum amount of code which is absolutely necessary for the semantics of that protocol.

While of course extremely useful in the context of OLE Documents, Uniform Data Transfer is a generic service with applications far beyond OLE Documents.

.2 Data Formats and Transfer Mediums

Before Uniform Data Transfer, virtually all standard protocols for data transfer were quite weak at describing the data being transferred and usually required the exchange to occur through global memory. This was especially true on Microsoft Windows: the format was described by a single 16-bit "clipboard format" and the medium was always global memory.

The problem with the "clipboard format" is that it can only describe the structure of the data, that is, identify the layout of the bits. For example, the format CF_TEXT describes ASCII text. CF_BITMAP describes a device-dependent bitmap of so many colors and such and such dimensions, but was incapable of describing the actual device it depends upon. Furthermore, none of these formats gave any indication of what was actually in the data such as the amount of detail—whether a bitmap or metafile contained the full image or just a thumbnail sketch.

The problem with always using global memory as a transfer medium is apparent when large amounts of data are exchanged. Unless you have a machine with an obnoxious amount of memory, an exchange of, say, a 20MB scanned true-color bitmap through global memory is going to cause considerable swapping to vir-

tual memory on the disk. Restricting exchanges to global memory means that no application can choose to exchange data *on disk* when it will usually *reside on disk* even when being manipulated and will usually use virtual memory on disk anyway. It would be much more efficient to allow the source of that data to indicate that the exchange happens on disk in the first place instead of forcing 20MB of data through a virtual-memory bottleneck to just have it end up on disk once again.

Further, *latency* of the data transfer is sometimes an issue, particularly in network situations. One often needs or wants to start processing the *beginning* of a large set of data before the end the data set has even reached the destination machine. To accomplish this, some abstraction on the medium by which the data is transferred is needed.

To solve these problems, COM defines two new data structures: FORMATETC and STGMEDIUM. FORMATETC is a better clipboard format, for the structure not only contains a clipboard format but also contains a device description, a detail description (full content, thumbnail sketch, iconic, and 'as printed'), and a flag indicating what storage device is used for a particular rendering. Two FORMATETC structures that differ only by storage medium are, for all intents and purposes, two different formats. STGMEDIUM is then the better global memory handle which contains a flag indicating the medium as well as a pointer or handle or whatever is necessary to access that actual medium and get at the data. Two STGMEDIUM structures may indicate different mediums and have different references to data, but those mediums can easily contain the exact same data.

So FORMATETC is what a consumer (client) uses to indicate the type of data it wants from a data source (object) and is used by the source to describe what formats it can provide. FORMATETC can describe virtually any data, including other objects such a monikers. A client can ask a data object for an enumeration of its formats by requesting the data object's IEnumFORMATETC interface. Instead of an object blandly stating that it has "text and a bitmap" it can say it has "A device-independent string of text that is stored in global memory" and "a thumbnail sketch bitmap rendered for a 100dpi dot-matrix printer which is stored in an IStorage object." This ability to tightly describe data will, in time, result in higher quality printer and screen output as well as more efficiency in data browsing where a thumbnail sketch is much faster to retrieve and display than a full detail rendering.

STGMEDIUM means that data sources and consumers can now choose to use the most efficient exchange medium on a per-rendering basis. If the data is so big that it should be kept on disk, the data source can indicate a disk-based medium in it's preferred format, only using global memory as a backup if that's all the consumer understands. This has the benefit of using the *best* medium for exchanges as the default, thereby improving overall performance of data exchange between applications—if some data is already on disk, it does not even have to be loaded in order to send it to a consumer who doesn't even have to load it upon receipt. *At worst*, COM's data exchange mechanisms would be *as good as anything available today* where all transfers restricted to global memory. *At best*, data exchanges can be effectively instantaneous even for large data.

Note that two potential storage mediums that can be used in data exchange are storage objects and stream objects. Therefore Uniform Data Transfer as a technology itself builds upon the Persistent Storage technology as well as the basic COM foundation. Again, this enables each piece of code in an application to be leveraged elsewhere.

.3 Data Selection

A data object can vary to a number of degrees as to what exact data it can exchange through the IDataObject interface. Some data objects, such as those representing the clipboard or those used in a drag & drop operation, statically represent a specific selection of data in the source, such as a range of cells in a spreadsheet, a certain portion of a bitmap, or a certain amount of text. For the life of such static data objects, the data underneath them does not change.

Other types of data objects, however, may support the ability to dynamically change their data set. This ability, however, is not represented through the IDataObject interface itself. In other words, the data object has to implement some *other* interface to support dynamic data selection. An example of such objects are those that support OLE for Real-Time Market Data (WOSA/XRT) specification.[25] OLE for Real-Time

[25] OLE for Real-Time Market Data was formerly called the "WOSA Extensions for Real Time Market Data". More information on this and other industry specific extensions to OLE is available from Microsoft.

Market Data uses a data object and the IDataObject interface for exchange of data, but use the IDispatch interface from OLE Automation to allow consumers of the data to dynamically instruct the data object to change its working set. In other words, the OLE Automation technology (built on COM but not part of COM itself) allows the consumer to identify the specific market issues and the information on those issues (high, low, volume, etc.) that it wants to obtain from the data object. In response, the data object internally determines where to retrieve that data and how to watch for changes in it. The data object then notifies the consumer of changes in the data through COM's Notification mechanism.

.4 Notification

Consumers of data from an external source might be interested in knowing when data in that source changes. This requires some mechanism through which a data object itself asynchronously notifies a client connected to it of just such an event at which point a client can remember to ask for an updated copy of the data when it later needs such an update.

COM handles notifications of this kind through an object called an *advise sink* which implements an interface called IAdviseSink.[26] This sink is a body that absorbs asynchronous notifications from a data source. The advise sink object itself, and the IAdviseSink interface is implemented by the consumer of data which then hands an IAdviseSink pointer to the data object in question. When the data object detects a change, it then calls a function in IAdviseSink to notify the consumer as illustrated in Figure 2-14.

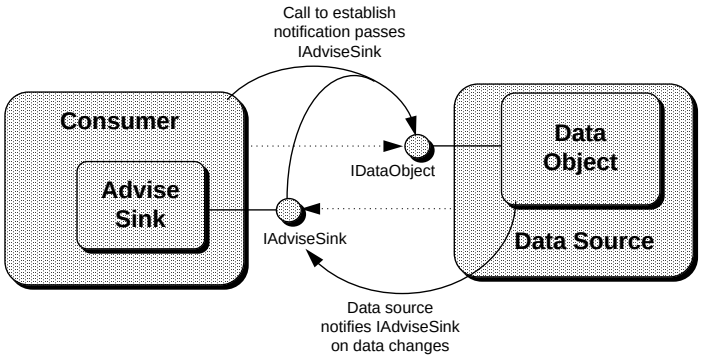


Figure 2-14: A consumer of data implements an object with the *IAdviseSink* interface through which data objects notify that consumer of data changes.

This is the most frequent situation where a client of one object, in this case the consumer, will itself implement an object to which the data object acts as a client itself. Notice that there are no circular reference counts here: the consumer object and the advise sink have different COM object identities, and thus separate reference counts. When the data object needs to notify the consumer, it simply calls the appropriate member function of IAdviseSink.

So IAdviseSink is more of a central collection of notifications of interest to a number of other interfaces and scenarios outside of IDataObject and data exchange. It contains, for example, a function for the event of a 'view' change, that is, when a particular view of data changes without a change in the underlying data. In addition, it contains functions for knowing when an object has saved itself, closed, or been renamed. All of these other notifications are of particular use in compound document scenarios and are used in OLE, but not COM proper. Chapter 14 will describe these functions but the mechanisms by which they are called are not part of COM and are not covered in this specification. Interested readers should refer to the OLE 2 Specifications from Microsoft.

Finally, data objects can establish notifications with multiple advise sinks. COM provides some assistance for data objects to manage an arbitrary number of IAdviseSink pointers through which the data object can pass each pointer to COM and then tell COM when to send notifications. COM in turn notifies all the advise sinks it maintains on behalf of the data object.

[26] Astute readers will wonder why Uniform Data Transfer is defined using the Connectable Objects interfaced described previously. The reason is simple: UDT was designed as part of the original OLE 2.0 specification in 1991, and Connectable Objects were not introduced until the release of the OLE Controls specification in 1993.

3. Objects And Interfaces

This chapter describes in detail the heart of COM: the notion of interfaces and their relationships to the objects on which they are implemented. More specifically, this chapter covers what an interface is (technically), interface calling conventions, object and interface identity, the fundamental interface called IUnknown, and COM's error reporting mechanism. In addition, this chapter describes how an object implements one or more interfaces as well as a special type of object called the "enumerator" which comes up in various contexts in COM.

As described in Chapters 1 and 2, the COM Library provides the fundamental implementation locator services to clients and provides all the necessary glue to help clients communicate transparently with object regardless of where those objects execute: in-process, out-of-process, or on a different machine entirely. All servers expose their object's services through interfaces, and COM provides implementations of the "proxy" and "stub" objects that make communication possible between processes and machines where RPC is necessary.

However, as we'll see in this chapter and those that follow, the COM Library also provides fundamental API functions for both clients and servers or, in general, any piece of code that uses COM, application or not. These API functions will be described in the context of where other applications or DLLs use them. A COM implementor reading this document will find the specifications for each function offset clearly from the rest of the text. These functions are implemented in the COM Library to standardize the parts of this specification that applications should not have to implement nor would want to implement. Through the services of the COM Library, all clients can make use of all objects in all servers, and all servers can expose their objects to all clients. Only by having a standard is this possible, and the COM Library enforces that standard by doing most of the hard work.

Not all the COM Library functions are truly fundamental. Some are just convenient wrappers to common sequences of other calls, sometimes called "helper functions." Others exist simply to maintain global lists for the sake of all applications. Others just provide a solid implementation of functions that could be implemented in every application, but would be tedious and wasteful to do so.

1 Interfaces

An interface, in the COM definition, is a contract between the user, or client, of some object and the object itself. It is a promise on the part of the object to provide a certain level of service, of functionality, to that client. Chapters 1 and 2 have already explained why interfaces are important COM and the whole idea of an object model. This chapter will now fill out the definition of an interface on the technical side.

.1 The Interface Binary Standard

Technically speaking, an interface is some data structure that sits between the client's code and the object's implementation through which the client requests the object's services. The interface in this sense is nothing more than a set of member functions that the client can call to access that object implementation. Those member functions are exposed outside the object implementor application such that clients, local or remote, can call those functions.

The client maintains a pointer to the interface which is, in actuality, a pointer to a pointer to an array of pointers to the object's implementations of the interface member functions. That's a lot of pointers; to clarify matters, the structure is illustrated in Figure 3-1.

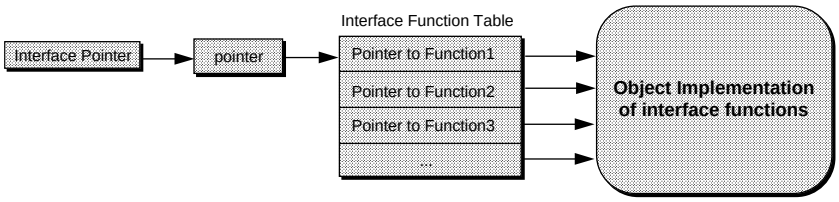


Figure 3-1: The interface structure: a client has a pointer to an interface which is a pointer to a pointer to an array (table) of pointers to the object's implementation.

By convention the pointer to the interface function table is called the pVtbl pointer. The table itself is generally referred to with the name vtbl for "virtual function table."

On a given implementation platform, a given method in a given interface (a particular IID, that is) has a fixed calling convention; this is decoupled from the implementation of the interface. In principle, this decision can be made on a method by method basis, though in practice on a given platform virtually all methods in all interfaces use the same calling convention. On Microsoft's 16-bit Windows platform, this default is the __far __cdecl calling convention; on Win32 platforms, the __stdcall calling convention is the default for methods which do not take a variable number of arguments, and __cdecl is used for those that do.

In contrast, just for note, COM API functions (not interface members) use the standard host system-call calling convention, which on both Microsoft Win16 and Win32 is the __far __pascal sequence.

Finally, and quite significantly, ***all strings passed through all COM interfaces*** (and, at least on Microsoft platforms, all COM APIs) ***are Unicode strings***. There simply is no other reasonable way to get interoperable objects in the face of (i) location transparency, and (ii) a high-efficiency object architecture that doesn't in all cases intervene system-provided code between client and server. Further, this burden is in practice not large.

When calling member functions, the caller must include an argument which is the pointer to the object instance itself. This is automatically provided in C++ compilers and completely hidden from the caller. The Microsoft Object Mapping[27] specifies that this pointer is pushed very last, immediately before the return address. The location of this pointer is the reason that the pIInterface pointer appears at the *beginning* of the argument list of the equivalent C function prototype: it means that the layout in the stack of the parameters to the C function prototype is exactly that expected by the member function implemented in C++, and so no re-ordering is required.

Usually the pointer to the interface itself is the pointer to the entire object structure (state variables, or whatever) and that structure immediately follows[28] the pVtbl pointer memory as shown in Figure 3-2.

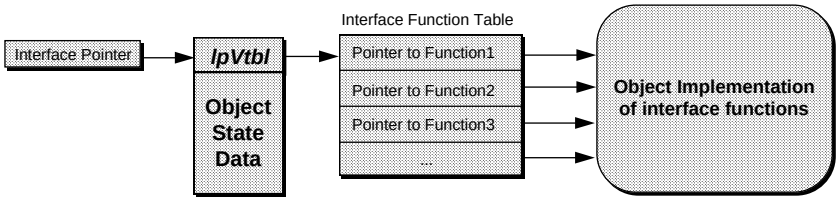


Figure 3-2: Convention places object data following the pointer to the interface function table.

Since the pVtbl is received as the this pointer in the interface function, the implementor of that function knows which object is being called—an object is, after all, some structure and functions to manipulate that structure, and the interface definition here supplies both.

In any case, this "vtbl" structure is called a binary standard because on the binary level, the structure is completely determined by the particular interface being used and the platform on which it is being invoked. It is independent of the programming language or tool used to create it. In other words, a program can be written in C to generate this structure to match what C++ does automatically. For more details, see the sec-

[27] The "Microsoft Object Mapping" is an open specification describing the detailed layout of C++ objects. It is supported by the MS C/C++ compiler, as well as C++ compilers from other vendors including Borland, Symantec, Watcom, , and others. This is also the location of the this pointer as placed by CFront when using the traditional right-to-left __cdecl calling sequence. Thus, we achieve a large degree of interoperability.

[28] Usually this data *follows* the pVtbl pointer, but this is not required. It is perfectly legal for object-specific data to precede the vtbl pointer, and this in fact will be common with many C++ compilers.

tion "C vs. C++" below. You could even create this structure in assembly if so inclined. Since compilers for other languages eventually reduce source code to assembly (as is the compiler itself) it is really a matter for compiler vendors to support this structure for languages such as Pascal, COBOL, Smalltalk, etc. Thus COM clients, objects, and servers can be written in any languages with appropriate compiler support.

Note that it is technically legal for the binary calling conventions for a given interface to vary according the particular implementation platform in question, though this flexibility should be exercised by COM system implementors only with very careful attention to source portability issues. It is the case, for example, that on the Macintosh, the pVtbl pointer does not point to the first function in the vtbl, but rather to a dummy pointer slot (which is ignored) immediately before the first function; all the function pointers are thus offset by an index of one in the vtbl.

An interface implementor is free to use the memory before and beyond the "as-specified-by-the-standard" vtbl for whatever purpose he may wish; others cannot assume anything about such memory.

.2 Interface Definition and Identity

Every interface has a name that serves as the programmatic compile-time type in code that uses that interface (either as a client or as an object implementor). The convention is to name each interface with a capital "I" followed by some descriptive label that indicates what functionality the interface encompasses. For example, IUnknown is the label of the interface that represents the functionality of an object when all else about that object is unknown.

These programmatic types are defined in header files provided by the designer of the interface through use of the Interface Description Language (IDL, see next section). For C++, an interface is defined as an abstract base, that is, a structure containing nothing but "pure virtual" member functions. This specification uses C++ *notation* to express the declaration of an interface. For example, the IUnknown interface is declared as:

```
interface IUnknown
    {
    virtual HRESULT    QueryInterface(IID& iid, void** ppv) =0;
    virtual ULONG      AddRef(void) =0;
    virtual ULONG      Release(void) =0;
    };
```

where "virtual" and "=0" describe the attribute of a "pure virtual" function and where the interface keyword is defined as:

```
#define    interface    struct
```

The programmatic name and definition of an interface defines a type such that an application can declare a pointer to an interface using standard C++ syntax as in IUnknown *.

In addition, this specification as a notation makes some use of the C++ reference mechanism in parameter passing, for example:

```
QueryInterface(const IID& iid, void**ppv);
```

Usually "const <type>&" is written as "REF<type>" as in REFIID for convenience. As you might expect, this example would appear in a C version of the interface as a parameter of type:

```
const IID * const
```

Input parameters passed by reference will themselves be const, as shown here. In-out or out- parameters will not.

The use of the interface keyword is more a documentation technique than any requirement for implementation. An interface, as a binary standard, is definable in any programming language as shown in the previous section. This specification's use of C++ syntax is just a convenience.[29] Also, for ease of reading, this specification generally omits parameter types in code fragments such as this but does document those parameters and types fully with each member function. Types do, of course, appear in header files with interfaces.

It is very important to note that the programmatic name for an interface is only a *compile-time* type used in application source code. Each interface must also have a *run-time* identifier. This identifier enables a caller to query (via QueryInterface) an object for a desired interface. Interface identifiers are GUIDs, that is, global-

[29] And, indeed, this syntax will at times be somewhat abused.

ly-unique 16 byte values, of type IID. The person who defines the interface allocates and assigns the IID as with any other GUID, and he informs others of his choice at the same time he informs them of the interface member functions, semantics, etc. Use of a GUID for this purpose guarantees that the IID will be unique in all programs, on all machines, for all time, the run-time identifier for a given interface will in fact have the same 16 byte value.

Programmers who define interfaces convey the interface identifier to implementors or clients of that interface along with the other information about the interface (in the form of header files, accompanying semantic documentation, etc.). To make application source code independent of the representation of particular interface identifiers, it is standard practice that the header file defines a constant for each IID where the symbol is the name of the interface prefixed with "IID_" such that the name can be derived algorithmically. For example, the interface IUnknown has an identifier called IID_IUnknown.

For brevity in this specification, this definition will not be repeated with each interface, though of course it is present in the COM implementation.

.3 Defining Interfaces: IDL

The Interface Description Language (IDL) is based on the Open Software Foundation (OSF) Distributed Computing Environment (DCE) specification for describing interfaces, operations, and attributes to define remote procedure calls. COM extends the IDL to support distributed objects.

A designer can define a new custom interface by writing an interface definition file. The interface definition file uses the IDL to describe data types and member functions of an interface. The interface definition file contains the information that defines the actual contract between the client application and server object. The interface contract specifies three things:

- *Language binding*—defines the programming model exposed to the application program using a particular programming language.
- *Application binary interface*—specifies how consumers and providers of the interface interoperate on a particular target platform.
- *Network interface*—defines how client applications access remote server objects via the network.

After completing the interface definition file, the programmer runs the IDL compiler to generate the interface header and the source code necessary to build the interface proxy and interface stub that the interface definition file describes. The interface header file is made available so client applications can use the interface. The interface proxy and interface stub are used to construct the proxy and stub DLLs. The DLL containing the interface proxy must be distributed with all client applications that use the new interface. The DLL containing the interface stub must be distributed with all server objects that provide the new interface.

It is important to note that the IDL is a tool that makes the job of defining interfaces easier for the programmer, and is one of possibly many such tools. It is not the key to COM interoperability. COM compliance does not require that the IDL compiler be used. However, as IDL is broadly understood and used, it provides a convenient means by which interface specifications can be conveyed to other programmers.

.4 C vs. C++ vs. ...

This specification documents COM interfaces using C++ syntax as a notation but (again) does not mean COM requires that programmers use C++, or any other particular language. COM is based on a *binary* interoperability standard, rather than a *language* interoperability standard. Any language supporting "structure" or "record" types containing double-indirected access to a table of function pointers is suitable.

However, this is not to say all languages are created equal. It is certainly true that since the binary vtbl standard is exactly what most C++ compilers generate on PC and many RISC platforms, C++ is a *convenient* language to use over a language such as C.

That being said, COM can declare interface declarations for both C++ and C (and for other languages if the COM implementor desires). The C++ definition of an interface, which in general is of the form:

```
interface ISomeInterface
    {
    virtual RET_T   MemberFunction(ARG1_T arg1, ARG2_T arg2 /*, etc */);
    [Other member functions]
```

```
    ...
    };
```

then the corresponding C declaration of that interface looks like

```
typedef struct ISomeInterface
    {
    ISomeInterfaceVtbl *   pVtbl;
    } ISomeInterface;

typedef struct ISomeInterfaceVtbl ISomeInterfaceVtbl;

struct ISomeInterfaceVtbl
    {
    RET_T (*MemberFunction)(ISomeInterface * this, ARG1_T arg1,
        ARG2_T arg2 /*, etc */);
    [Other member functions]
    } ;
```

This example also illustrates the algorithm for determining the signature of C form of an interface function given the corresponding C++ form of the interface function:

- Use the same argument list as that of the member function, but add an initial parameter which is the pointer to the interface. This initial parameter is a pointer to a C type of the same name as the interface.
- Define a structure type which is a table of function pointers corresponding to the vtbl layout of the interface. The name of this structure type should be the name of the interface followed by "Vtbl." Members in this structure have the same names as the member functions of the interface.

The C form of interfaces, when instantiated, generates exactly the same binary structure as a C++ interface does when some C++ class inherits the function signatures (but no implementation) from an interface and overrides each virtual function.

These structures show why C++ is more convenient for the object implementor because C++ will automatically generate the vtbl and the object structure pointing to it in the course of instantiating an object. A C object implementor must define and object structure with the pVtbl field first, explicitly allocate both object structure and interface Vtbl structure, explicitly fill in the fields of the Vtbl structure, and explicitly point the pVtbl field in the object structure to the Vtbl structure. Filling the Vtbl structure need only occur once in an application which then simplifies later object allocations. In any case, once the C program has done this explicit work the binary structure is indistinguishable from what C++ would generate.

On the client side of the picture there is also a small difference between using C and C++. Suppose the client application has a pointer to an ISomeInterface on some object in the variable *psome*. If the client is compiled using C++, then the following line of code would call a member function in the interface:

```
psome->MemberFunction(arg1, arg2, /* other parameters */);
```

A C++ compiler, upon noting that the type of psome is an ISomeInterface * will know to actually perform the double indirection through the hidden pVtbl pointer and will remember to push the psome pointer itself on the stack so the implementation of MemberFunction knows which object to work with. This is, in fact, what C++ compilers do for any member function call; C++ programmers just never see it.

What C++ actually does is be expressed in C as follows:

```
psome->lpVtbl->MemberFunction(psome, arg1, arg2, /* other parameters */);
```

This is, in fact, how a client written in C would make the same call. These two lines of code show why C++ is more convenient—there is simply less typing and therefore fewer chances to make mistakes. The resulting source code is somewhat cleaner as well. The key point to remember, however, is that *how the client calls an interface member depends solely on the language used to implement the client and is completely unrelated to the language used to implement the object*. The code shown above to call an interface function is the code necessary to work with the interface binary standard and not the object itself.

.5 Remoting Magic Through Vtbls

The double indirection of the *vtbl* structure has an additional, indeed enormous, benefit: the pointers in the table of function pointers do not need to point directly to the real implementation in the real object. This is the heart of Location Transparency.

It is true that in the in-process server case, where the object is loaded directly into the client process, the function pointers in the table are, in fact, the actual pointers to the actual implementation. So a function call from the client to an interface member directly transfers execution control to the interface member function.

However, this cannot possibly work for local, let alone remote, object, because pointers to memory are absolutely not sharable between processes. What must still happen to achieve transparency is that the client continues to call interface member functions *as if it were calling the actual implementation*. In other words, the client uniformly transfers control to some object's member function by making the call.

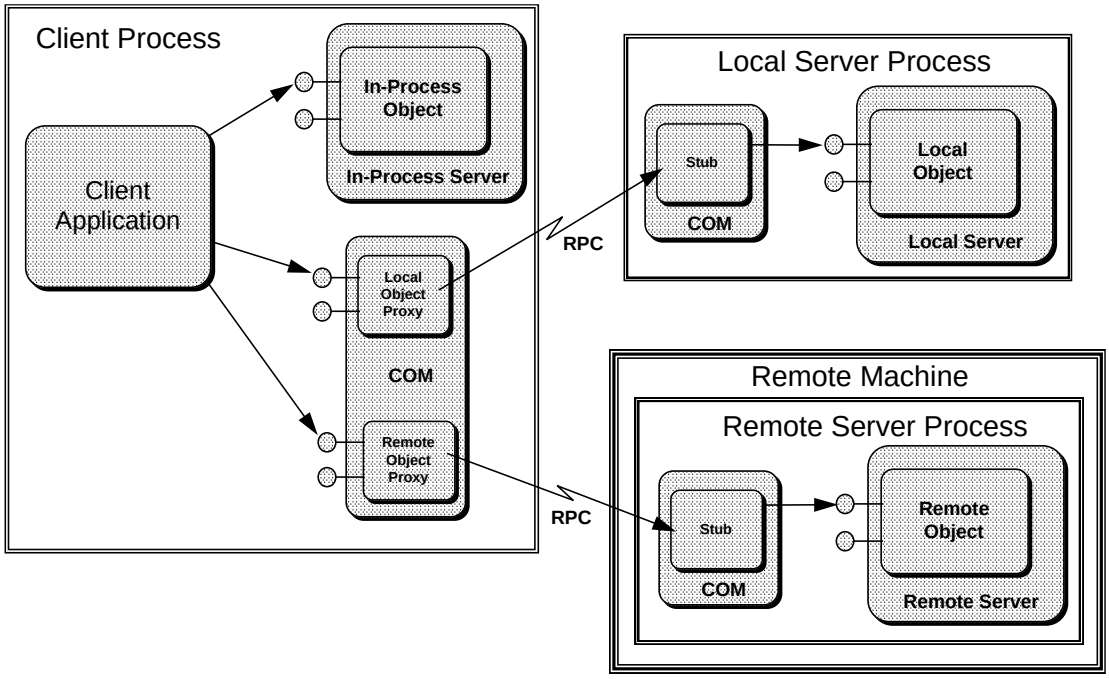


Figure 3-3: A client always calls interface members in some in-process object. If the actual object is local or remote, the call is made to a proxy object which then makes a remote procedure call to the actual object.

So what member function actually executes? The answer is that the interface member called is implemented by a proxy object that is always an in-process object that acts on behalf of the object being called. This proxy object knows that the actual object is running in a local or remote server and so it must somehow make a remote procedure call, through a standard RPC mechanism, to that object as shown in Figure 3-3.

The proxy object packages up the function parameters in some data packets and generates an RPC call to the local or remote object. That packet is picked up by a stub object in the server's process, on the local or a remote machine, which unpacks the parameters and makes the call to the real implementation of the member function. When that function returns, the stub packages up any out-parameters and the return value, sends it back to the proxy, which unpacks them and returns them to the original client. For exact details on how the proxy-stub and RPC mechanisms work, see Chapter 7.

The bottom line is that client and server always talk to each other as if everything was in-process. All calls from the client and all calls to the server do at some point, in fact, happen in-process. But because the vtbl structure allows some agent, like COM, to intercept all function calls and all returns from functions, that agent can redirect those calls to an RPC call as necessary. All of this is completely transparent to the client and server, hence Location Transparency.[30]

[30] Of course, if a client timed the call it might be able to discern a performance penalty if it had both in-process and out-of-process objects to compare.

2 Globally Unique Identifiers

As mentioned earlier in this document, the GUID, from which are also obtained CLSID, IIDs, and any other needed unique identifier, is a 128-bit, or 16-byte, value. The term GUID as used in this specification is completely synonymous and interchangeable with the term "UUID" as used by the DCE RPC architecture; they are indeed one and the same notion. In binary terms, a GUID is a data structure defined as follows, where DWORD is 32-bits, WORD is 16-bits, and BYTE is 8-bits:

```
typedef struct GUID      {
    DWORD  Data1;
    WORD   Data2;
    WORD   Data3;
    BYTE   Data4[8];
    } GUID;
```

This structure provides applications with some way of addressing the parts of a GUID for debugging purposes, if necessary. This information is also needed when GUIDs are transmitted between machines of different byte orders.

For the most part, applications never manipulate GUIDs directly—they are almost always manipulated either as a constant, such as with interface identifiers, or as a variable of which the absolute value is unimportant. For example, a client might enumerate all object classes registered on the system and display a list of those classes to an end user. That user selects a class from the list which the client then maps to an absolute CLSID value. The client does not care what that value is—it simply knows that it uniquely identifies the object that the user selected.

The GUID design allows for coexistence of several different allocation technologies, but the one by far most commonly used incorporates a 48-bit machine unique identifier together with the current UTC time and some persistent backing store to guard against retrograde clock motion. It is in theory capable of allocating GUIDs at a rate of 10,000,000 per second per machine for the next 3240 years, enough for most purposes.

For further information regarding GUID allocation technologies, see pp585-592 of [CAE RPC].[31]

3 The IUnknown Interface

This specification has already mentioned the IUnknown interface many times. It is the fundamental interface in COM that contains basic operations of not only all objects, but all interfaces as well: reference counting and QueryInterface. All interfaces in COM are polymorphic with IUnknown, that is, if you look at the first three functions in any interface you see QueryInterface, AddRef, and Release. In other words, IUnknown is base interface from which all other interfaces inherit.

Any single object usually only requires a single implementation of the IUnknown member functions. This means that by virtue of implementing any interface on an object you completely implement the IUnknown functions. You do not generally need to explicitly inherit from nor implement IUnknown as its own interface: when queried for it, simply typecast another interface pointer into an IUnknown* which is entirely legal with polymorphism.

In some specific situations, more notably in creating an object that supports aggregation, you may need to implement one set of IUnknown functions for all interfaces as well as a stand-alone IUnknown interface. The reasons and techniques for this are described in the "Object Reusability" section of Chapter 6.

In any case, any object implementor will implement IUnknown functions, and we are now in a position to look at them in their precise terms.

.1 IUnknown Interface

IUnknown supports the capability of getting to other interfaces on the same object through QueryInterface. In addition, it supports the management of the existence of the interface instance though AddRef and Release.

[31] Though be aware that the use of the term GUID on page 587 is regrettably *not* the same as its usage in this specification. In this specification, the term GUID is used to refer to all identifiers that are "interoperable" with UUIDs as defined on p586; p587 uses the term to refer to one specific central-authority allocation scheme. Apologies to those who may be confused by this state of affairs.

The following is the definition of IUnknown using the IDL notation; for details on the syntax of IDL see Chapter 15.[32]

```
[
  object,
  uuid(00000000-0000-0000-C000-000000000046),
  pointer_default(unique)
]
interface IUnknown
{
    HRESULT     QueryInterface([in] REFIID iid, [out] void **ppv) ;
    ULONG       AddRef(void) ;
    ULONG           Release(void);
}
```

.1 IUnknown::QueryInterface

HRESULT IUnknown::QueryInterface(iid, ppv)

Return a pointer within this object instance that implements the indicated interface. Answer NULL if the receiver does not contain an implementation of the interface.

It is required that any query for the specific interface IUnknown[33] always returns the *same actual pointer value*, no matter through which interface derived from IUnknown it is called. This enables the following identity test algorithm to determine whether two pointers in fact point to the same object: call QueryInterface(IID_IUnknown, ...) on both and compare the results.

In contrast, queries for interfaces *other* than IUnknown are *not* required to return the same actual pointer value each time a QueryInterface returning one of them is called. This, among other things, enables sophisticated object implementors to free individual interfaces on their objects when they are not being used, recreating them on demand (reference counting is a per-interface notion, as is explained further below). This requirement is the basis for what is called *COM identity*.

It is required that the set of interfaces accessible on an object via QueryInterface be static, not dynamic, in the following precise sense.[34] Suppose we have a pointer to an interface

```
ISomeInterface * psome = (some function returning an ISomeInterface *);
```

where ISomeInterface derives from IUnknown. Suppose further that the following operation is attempted:

```
IOtherInterface * pother;
HRESULT     hr;
hr=psome->QueryInterface(IID_IOtherInterface, &pother);          //line 4
```

Then, the following must be true:

- If hr==S_OK, then if the QueryInterface in "line 4" is attempted a second time from the same psome pointer, then S_OK must be answered again. This is independent of whether or not pother->Release was called in the interim. In short, if you can get to a pointer once, you can get to it again.
- If hr==E_NOINTERFACE, then if the QueryInterface in line 4 is attempted a second time from the same psome pointer, then E_NOINTERFACE must be answered again. In short, if you didn't get it the first time, then you won't get it later.

Furthermore, *QueryInterface* must be reflexive, symmetric, and transitive with respect to the set of interfaces that are accessible. That is, given the above definitions, then we have the following:

Symmetric:	psome->QueryInterface(IID_ISomeInterface, ...) must succeed
Reflexive:	If in line 4, pother was successfully obtained, then pother->QueryInterface(IID_ISomeInterface, ...) must succeed.

[32] Throughout this document IDL notation is used to precisely describe interfaces and other types. The actual IDL files contain additional IDL specifies that are used by the IDL compiler to optimize the generation of marshaling code, but have no bearing on the actual interface contract.

[33] That is, a QueryInterface invocation where iid is 00000000-0000-0000-C000-000000000046.

[34] While this set of rules may seem surprising to some, they are needed in order that remote access to interface pointers can be provided with a reasonable degree of efficiency (without this, interface pointers could not be cached on a remote machine). Further, as QueryInterface forms the fundamental architectural basis by which clients reason about the capabilities of an object with which they have come in contact, stability is needed to make any sort of reasonable reasoning and capability discovery possible.

Transitive: If in line 4, pother was successfully obtained, and we do

```
IYetAnother * pyet;
pother->QueryInterface(IID_IYetAnother, &pyet);        //Line 7
```

and pyet is successfully obtained in line 7, then

```
pyet->QueryInterface(IID_ISomeInterface, ...)
```

must succeed.

Here, "must succeed" means "must succeed barring catastrophic failures." As was mentioned above, it is specifically *not* the case that two QueryInterface calls on the same pointer asking for the same interface must succeed and return exactly the same *pointer value* (except in the IUnknown case as described previously).

Argument	Type	Description
iid	REFIID	The interface identifier desired.
ppv	void**	Pointer to the object with the desired interface. In the case that the interface is not supported or another error occurred, *ppv must be set to NULL.

Return Value	Meaning
S_OK	Success. The interface is supported
E_NOINTERFACE	The interface is not supported
E_UNEXPECTED	An unknown error occurred.

.2 IUnknown::AddRef

ULONG IUnknown::AddRef(void)

Increments the reference count in this interface instance.

Objects implementations are required to support a certain minimum size for the counter that is internally maintained by AddRef. In short, this counter must be at least 31 bits large. The precise rule is that the counter must be large enough to support 2^{31}-1 outstanding pointer references to all the interfaces on a given object taken as a whole. Just make it a 32 bit unsigned integer, and you'll be fine.

Argument	Type	Description
return value	ULONG	The resulting value of the reference count. This value is returned solely for diagnostic/testing purposes; it absolutely holds no meaning for release code since in certain situations it is unstable

.3 IUnknown::Release

ULONG IUnknown::Release(void)

Release a reference to this interface instance.

If AddRef has been called on this object (through any IUnknown members of its interfaces) *n* times and this is the *n*th call to Release, then the interface instance will free itself.

Release cannot indicate failure; if a client needs to know that resources have been freed etc., it must use a method in some interface on the object with higher level semantics before calling release.

Argument	Type	Description
return value	ULONG	The resulting value of the reference count. This value is returned solely for diagnostic/testing purposes; it only has meaning when the return is zero meaning that the object cannot be considered valid in any way by the caller. Non-zero values are meaningless to the caller.

.2 Reference Counting

Objects accessed through interfaces use a reference counting mechanism to ensure that the lifetime of the object includes the lifetime of references to it. This mechanism is adopted so that independent components can obtain and release access to a single object, and not have to coordinate with each other over the lifetime management. In a sense, the object provides this management, so long as the client components conform to the rules. Within a single component that is completely under the control of a single development organiza-

tion, clearly that organization can adopt whatever strategy it chooses. The following rules are about how to manage and communicate interface instances between components, and are a reasonable starting point for a policy within a component.

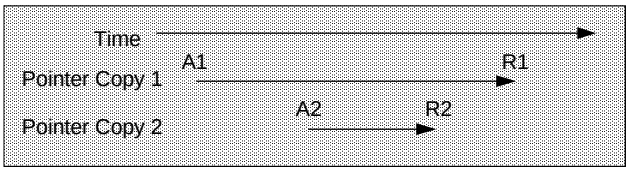
Note that the reference counting paradigm applies only to pointers to interfaces; pointers to data are not referenced counted.

It is important to be very clear on exactly when it is necessary to call AddRef and Release through an interface pointer. By its nature, pointer management is a cooperative effort between separate pieces of code, which must all therefore cooperate in order that the overall management of the pointer be correct. The following discussion should hopefully clarify the rules as to when AddRef and Release need to be called in order that this may happen. Some special reference counting rules apply to objects which are aggregated; see the discussion of aggregation in Chapter 6.

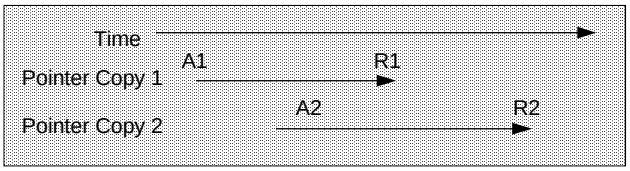
The conceptual model is the following: interface pointers are thought of as living in pointer variables, which for the present discussion will include variables in memory locations and in internal processor registers, and will include both programmer- and compiler-generated variables. In short, it includes all internal computation state that holds an interface pointer. Assignment to or initialization of a pointer variable involves creating a *new copy* of an already existing pointer: where there was one copy of the pointer in some variable (the value used in the assignment/initialization), there is now two. An assignment to a pointer variable *destroys* the pointer copy presently in the variable, as does the destruction of the variable itself (that is, the scope in which the variable is found, such as the stack frame, is destroyed).

Rule 1: AddRef must be called for every new copy of an interface pointer, and Release called every destruction of an interface pointer except where subsequent rules explicitly permit otherwise.

This is the default case. In short, unless special knowledge permits otherwise, the worst case must be assumed. The exceptions to Rule 1 all involve knowledge of the relationships of the lifetimes of two or more copies of an interface pointer. In general, they fall into two categories.[35]



Category 1. Nested lifetimes



Category 2. Staggered overlapping lifetimes

In Category 1 situations, the AddRef A2 and the Release R2 can be omitted, while in Category 2, A2 and R1 can be eliminated.

Rule 2: Special knowledge on the part of a piece of code of the relationships of the beginnings and the endings of the lifetimes of two or more copies of an interface pointer can allow AddRef/Release pairs to be omitted.

The following rules call out specific common cases of Rule 2. The first two of these rules are particularly important, as they are especially common.

Rule 2a: *In-parameters to functions.* The copy of an interface pointer which is passed as an actual parameter to a function has a lifetime which is nested in that of the pointer used to initialize the value. The actual parameter therefore need not be separately reference counted.

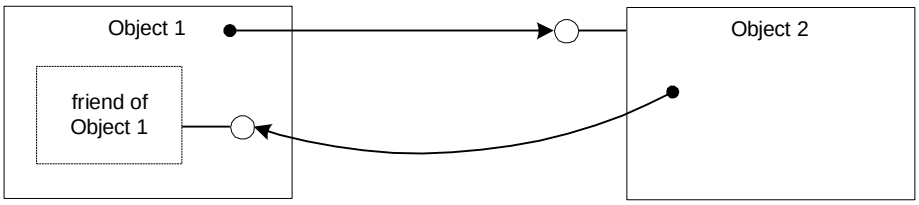
Rule 2b: *Out-parameters from functions, including return values.* This is a Category 2 situation. In order to set the out parameter, the function itself by Rule 1 must have a stable copy of the

[35] There are in fact more general cases than illustrated here involving n-way rather than 2-way interactions of matched AddRef / Release pairs, but that will not be elaborated on here.

interface pointer. On exit, the responsibility for releasing the pointer is transferred from the callee to the caller. The out-parameter thus need not be separately reference counted.

Rule 2c: *Local variables.* A function implementation clearly has omniscient knowledge of the lifetimes of each of the pointer variables allocated on the stack frame. It can therefore use this knowledge to omit redundant AddRef/Release pairs.

Rule 2d: *Backpointers*. Some data structures are of the nature of containing two components, A and B, each with a pointer to the other. If the lifetime of one component (A) is known to contain the lifetime of the other (B), then the pointer from the second component back to the first (from B to A) need not be reference counted. Often, avoiding the cycle that would otherwise be created is important in maintaining the appropriate deallocation behavior. However, such non-reference counted pointers should be used *with extreme caution.*In particular, as the remoting infrastructure cannot know about the semantic relationship in use here, such backpointers cannot be remote references. In almost all cases, an alternative design of having the backpointer refer a second "friend" object of the first rather than the object itself (thus avoiding the circularity) is a superiour design. The following figure illustrates this concept.[36]



The following rules call out common non-exceptions to Rule 1.

Rule 1a: *In-Out-parameters to functions.* The caller must AddRef the actual parameter, since it will be Released by the callee when the out-value is stored on top of it.

Rule 1b: *Fetching a global variable.* The local copy of the interface pointer fetched from an existing copy of the pointer in a global variable must be independently reference counted since called functions might destroy the copy in the global while the local copy is still alive.

Rule 1c: *New pointers synthesized out of "thin air."* A function which synthesizes an interface pointer using special internal knowledge rather than obtaining it from some other source must do an initial AddRef on the newly synthesized pointer. Important examples of such routines include instance creation routines, implementations of IUnknown::QueryInterface, etc.

Rule 1d: *Returning a copy of an internally stored pointer.* Once the pointer has been returned, the callee has no idea how its lifetime relates to that of the internally stored copy of the pointer. Thus, the callee must call AddRef on the pointer copy before returning it.

Finally, when implementing or using reference counted objects, a technique sometimes termed "artificial reference counts" sometimes proves useful. Suppose you're writing the code in method Foo in some interface IInterface. If in the implementation of Foo you invoke functions which have even the remotest chance of decrementing your reference count, then such function may cause you to release before it returns to Foo. The subsequent code in Foo will crash.

A robust way to protect yourself from this is to insert an AddRef at the beginning of Foo which is paired with a Release just before Foo returns:

```
void IInterface::Foo(void) {
    this[37]->AddRef();
    /*
     * Body of Foo, as before, except short-circuit returns
     * need to be changed.
     */
    this->Release();
    return;
    }
```

[36] The connection point interfaces introduced in the OLE Controls specification are a real world example of this concept.

[37] "This" is the appropriate thing to AddRef in an object implementation using the approach of multiply inheriting from the suite of interfaces supported by the object; more complex implementation strategies will need to modify this appropriately.

These "artificial" reference counts guarantee object stability while processing is done.

4 Error Codes and Error Handling

COM interface member functions and COM Library API functions use a specific convention for error codes in order to pass back to the caller both a useful return value and along with an indication of status or error information. For example, it is highly useful for a function to be capable of returning a Boolean result (true or false) as well as indicate failure or success—returning true and false means that the function executed successfully, and true or false is the answer whereas an error code indicates the function failed completely.

But before we get into error handling in COM, we'll first take a small digression. Many readers might here be wondering about exceptions. How do exceptions relate to interfaces? In short, *it is strictly illegal to throw an exception across an interface invocation*; all such cross-interface exceptions which are thrown are in fact bugs in the offending interface implementation. Why have such a policy? The first, straightforward, pragmatic reason is the technical reality that there simply isn't an ubiquitous exception model or semantic that is broadly supported across languages and operating systems that one could choose to permit; recall that location transparency and language independence are important design goals of COM. Further, simplicity is also an important design goal. It is well-understood that, quite apart from COM *per se*, the exceptions that may be legally thrown from a function implementation in the public interface of an encapsulated module must necessarily from part of the contract of that function implementation. Thus, a thrown exception across such a boundary is merely an alternative mechanism by which values may be returned from the function. In COM, we instead make use of the simpler, ubiquitous, already-existing return-value mechanism for returning information from a function as our error reporting mechanism: simply returning HRESULTs, which are the topic of this section.

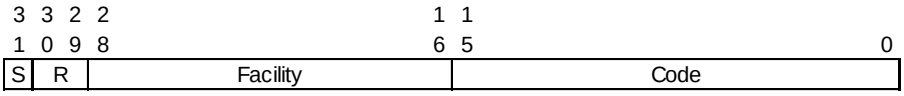
This all being said, it would be absolutely perfectly reasonable for the implementor of a tool for using or implementing COM interfaces to within the body of code managed by his tool turn errors returned from invoked COM interfaces into local exceptions and, conversely, to turn internally generated exceptions into error-returns across an interface boundary. This is yet another example of the clear architectural difference that needs to be made between the rules and design of the underlying COM system architecture and the capabilities and design freedom afforded to tools that support that architecture.

.1 HRESULT

The key type involved in COM error reporting is HRESULT.[38] In addition, the COM Library provides a few functions and macros to help applications of any kind deal with error information. An HRESULT is a simple 32-bit value:

```
typedef LONG HRESULT;
```

An HRESULT is divided up into an internal structure that has four fields with the following format (numbers indicate bit positions):



S: (1 bit) Severity field:

- 0 *Success*. The function was successful; it behaved according to its proscribed semantics.
- 1 *Error*. The function failed due to an error condition.

R: (2 bits) Reserved for future use; must be set to zero by present programs generating HRESULTs; present code should not take action that relies on any particular bits being set or cleared this field.

[38] The name "HRESULT" is retained for historical reasons. Readers familiar with programming COM on the Windows platform will note that HRESULT is analogous to SCODE.

Facility: (13 bits) Indicates which group of status codes this belongs to. New facilities must be allocated by a central coordinating body since they need to be universally unique.[39] However, the need for new facility codes is very small. Most cases can and should use FACILITY_ITF. See the section "Use of FACILITY_ITF" below.

Code: (16 bits) Describes what actually took place, error or otherwise.

COM presently defines the following facility codes:

Facility Name	Facility Value	Description
FACILITY_NULL	0	Used for broadly applicable common status codes that have no specific grouping. S_OK belongs to this facility, for example.
FACILITY_ITF	4	Used for by far the majority of result codes that are returned from an interface member function. Use of this facility indicates that the meaning of the error code is defined solely by the definition of the particular interface in question; an HRESULT with exactly the same 32-bit value returned from another interface might have a different meaning
FACILITY_RPC	1	Used for errors that result from an underlying remote procedure call implementation. In general, this specification does not explicitly document the RPC errors that can be returned from functions, though they nevertheless can be returned in situations where the interface being used is in fact remoted
FACILITY_DISPATCH	2	Used for IDispatch-interface-related status codes.
FACILITY_STORAGE	3	Used for persistent-storage-related status codes. Status codes whose code (lower 16 bits) value is in the range of DOS error codes (less than 256) have the same meaning as the corresponding DOS error.
FACILITY_WIN32	7	Used to provide a means of mapping an error code from a function in the Win32 API into an HRESULT. The semantically significant part of a Win32 error is 16 bits large.
FACILITY_WINDOWS	8	Used for additional error codes from Microsoft-defined interfaces.
FACILITY_CONTROL	10	Used for OLE Controls-related error values.

A particular HRESULT value by convention uses the following naming structure:

<**Facility**>_<**Sev**>_<**Reason**>

where <**Facility**> is either the facility name or some other distinguishing identifier, <**Sev**> is a single letter, one of the set { S, E } indicating the severity (success or error), and <**Reason**> is a short identifier that describes the meaning of the code. Status codes from FACILITY_NULL omit the <**Facility**>_ prefix. For example, the status code E_NOMEMORY is the general out-of memory error. All codes have either S_ or E_ in them allowing quick visual determination if the code means success or failure.

The general "success" HRESULT is named S_OK, meaning "everything worked" as per the function specification. The value of this HRESULT is zero. In addition, as it is useful to have functions that can succeed but return Boolean results, the code S_FALSE is defined are success codes intended to mean "function worked and the result is false."

```
#define  S_OK       0
#define  S_FALSE    1
```

A list of presently-defined standard error codes and their semantics can be found in Appendix A.

From a general interface design perspective, "success" status codes should be used for circumstances where the consequence of "what happened" in a method invocation is most naturally understood and dealt with by client code by looking at the out-values returned from the interface function: NULL pointers, etc. "Error" status codes should in contrast be used in situations where the function has performed in a manner that would naturally require "out of band" processing in the client code, logic that is written to deal with situations in which the interface implementation truly did not behave in a manner under which normal client code can make normal forward progress. The distinction is an imprecise and subtle one, and indeed many existing interface definitions do not for historical reasons abide by this reasoning. However, with this approach, it becomes feasible to implement automated COM development tools that appropriately turn the error codes into exceptions as was mentioned above.

[39] As of this writing, said body is Microsoft Corporation.

Interface functions in general take the form:

```
HRESULT ISomeInteface::SomeFunction(ARG1_T arg1, ... , ARGN_T argn, RET_T * pret);
```

Stylistically, what would otherwise be the return value is passed as an out-value through the last argument of the function. COM development tools which map error returns into exceptions might also consider mapping the last argument of such a function containing only one out-parameter into what the programmer sees as the "return value" of the method invocation.

The COM remoting infrastructure only supports reporting of RPC-induced errors (such as communication failures) through interface member functions that return HRESULTs. For interface member functions of other return types (e.g.: void), such errors are silently discarded. To do otherwise would, to say the least, significantly complicate local / remote transparency.

.1 Use of FACILITY_ITF

The use of FACILITY_ITF deserves some special discussion with respect to interfaces defined in COM and interfaces that will be defined in the future. Where as status codes with other facilities (FACILITY_NULL, FACILITY_RPC, etc.) have universal meaning, status codes in FACILITY_ITF have their meaning completely determined by the interface member function (or API function) from which they are returned; *the same 32-bit value in FACILITY_ITF returned from two different interface functions may have completely different meanings*.

The reasoning behind this distinction is as follows. For reasons of efficiency, it is unreasonable to have the primary error code data type (HRESULT) be larger than 32 bits in size. 32 bits is not large enough, unfortunately, to enable COM to develop an allocation policy for error codes that will universally avoid conflict between codes allocated by different non-communicating programmers at different times in different places (contrast, for instance, with what is done with IIDs and CLSIDs). Therefore, COM structures the use of the 32 bit SCODE in such a way so as to allow the a central coordinating body[40] to define *some* universally defined error codes while at the same time allowing other programmers to define new error codes without fear of conflict by limiting the places in which those field-defined error codes can be used. Thus:

1. Status codes in facilities other than FACILITY_ITF can only be defined by the central coordinating body.
2. Status codes in facility FACILITY_ITF are defined solely by the *definer of the interface* or API by which said status code is returned. That is, in order to avoid conflicting error codes, a human being needs to coordinate the assignment of codes in this facility, and we state that he who defines the interface gets to do the coordination.

COM itself defines a number of interfaces and APIs, and so COM defines many status codes in FACILITY_ITF. By design, none of the COM-defined status codes in fact have the same value, even if returned by different interfaces, though it would have been legal for COM to do otherwise.

Likewise, it is possible (though not required) for designers of COM interface suites to coordinate the error codes across the interfaces in that suite so as to avoid duplication. The designers of the OLE 2 interface suite, for example, ensured such lack of duplication.

Thus, with regard to which errors can be returned by which interface functions, it is the case that, in the extreme,

- It is legal that any COM-defined *error* code may in fact be returned by any COM-defined interface member function or API function. This includes errors presently defined in FACILITY_ITF. Further, COM may in the future define new failure codes (but not *success* codes) that may also be so ubiquitously returned.

  Designers of interface suites may if they wish choose to provide similar rules across the interfaces in their suites.
- Further, any *error* in FACILITY_RPC or other facility, even those errors not presently defined, may be returned.

Clients must treat error codes that are unknown to them as synonymous with E_UNEXPECTED, which in general should be and is presently a legal error return value from each and every interface member function

[40] As of this writing, said body is Microsoft Corporation.

in all interfaces; interface designers and implementors *are responsible to insure* that any newly defined error codes they should choose to invent or return will be such that that existing clients with code treating generic cases as synonymous with E_UNEXPECTED this will have reasonable behavior.

In short, if you know the function you invoked, you know as a client how to unambiguously take action on any error code you receive. The interface implementor is responsible for maintaining your ability to do same.

Normally, of course, only a small subset of the COM-defined status codes will be usefully returned by a given interface function or API, but the immediately preceding statements are in fact the actual interoperability rules for the COM-defined interfaces. This specification endeavors to point out which error codes are particularly useful for each function, but code must be written to correctly handle the general rule.

The present document is, however, precise as to which *success* codes may legally be returned.

Conversely, it is *only* legal to return a status code from the implementation of an interface member function which has been sanctioned by the designer of that interface as being legally returnable; otherwise, there is the possibility of conflict between these returned code values and the codes in-fact sanctioned by the interface designer. Pay particular attention to this when propagating errors from internally called functions. Nevertheless, as noted above, callers of interfaces must to guard themselves from imprecise interface implementations by treating any otherwise unknown returned error code (in contrast with success code) as synonymous with E_UNEXPECTED: experience shows that programmers are notoriously lax in dealing with error handling. Further, given the third bullet point above, this coding practice is *required* by clients of the COM-defined interfaces and APIs. Pragmatically speaking, however, this is little burden to programmers: normal practice is to handle a few special error codes specially, but treat the rest generically.

All the COM-defined FACILITY_ITF codes will, in fact, have a *code* value which lies in the region 0x0000 — 0x01FF. Thus, while it is indeed legal for the definer of a new function or interface to make use of any codes in FACILITY_ITF that he chooses in any way he sees fit, it is highly recommended that only *code* values in the range 0x0200 — 0xFFFF be used, as this will reduce the possibility of accidental confusion with any COM-defined errors. It is also highly recommended that designers of new functions and interfaces consider defining as legal that most if not all of their functions can return the appropriate status codes defined by COM in facilities other than FACILITY_ITF. E_UNEXPECTED is a specific error code that most if not all interface definers will wish to make universally legal.

.2 COM Library Error-Related Macros and Functions

The following macros and functions are defined in the COM Library include files to manipulate status code values.

```
#define SEVERITY_SUCCESS        0
#define SEVERITY_ERROR          1

#define SUCCEEDED(Status)       ((HRESULT)(Status) >= 0)
#define FAILED(Status)          ((HRESULT)(Status)<0)

#define HRESULT_CODE(hr)        ((hr) & 0xFFFF)
#define HRESULT_FACILITY(hr)        (((hr) >> 16) & 0x1fff)
#define HRESULT_SEVERITY(hr)        (((hr) >> 31) & 0x1)

#define MAKE_HRESULT(sev,fac,code) \
    ((HRESULT) (((unsigned long)(sev)<<31) | ((unsigned long)(fac)<<16) | ((unsigned long)(code))) )
```

.1 SUCCEEDED

SUCCEEDED(HRESULT Status)

The SUCCEEDED macro returns TRUE if the severity of the status code is either success or information; otherwise, FALSE is returned.

.2 FAILED

FAILED(HRESULT Status)

The FAILED macro returns TRUE if the severity of the status code is either a warning or error; otherwise, FALSE is returned.

.3 HRESULT_CODE

HRESULT_CODE(HRESULT hr)

HRESULT_CODE returns the error code part from a specified status code.

.4 HRESULT_FACILITY

HRESULT_FACILITY(HRESULT hr)

HRESULT_FACILITY extracts the facility from a specified status code.

.5 HRESULT_SEVERITY

HRESULT_SEVERITY(HRESULT hr)

HRESULT_SEVERITY extracts the severity field from the specified status code.

.6 MAKE_HRESULT

MAKE_HRESULT(SEVERITY sev, FACILITY fac, HRESULT hr)

MAKE_HRESULT makes a new status code given a severity, a facility, and a status code.

5 Enumerators and Enumerator Interfaces

A frequent programming task is that of iterating through a sequence of items. The COM interfaces are no exception: there are places in several interfaces described in this specification where a client of some object needs to iterate through a sequence of items controlled by the object. COM supports such enumeration through the use of "enumerator objects." Enumerators cleanly separate the caller's desire to loop over a set of objects from the callee's knowledge of how to accomplish that function.

Enumerators are just a concept; there is no actual interface called IEnumerator or IEnum or the like. This is due to the fact that the function signatures in an enumerator interface must include the type of the things that the enumerator enumerates. As a consequence, separate interfaces exist for each kind of thing that can be enumerated. However, the difference in the type being enumerated is the *only* difference between each of these interfaces; they are all used in fundamentally the same way. In other words, they are "generic" over the element type. This document describes the semantics of enumerators using a generic interface IEnum and the C++ parameterized type syntax where ELT_T, which stands for "**EL**emen**T T**ype"[41] is representative of the type involved in the enumeration:

```
[
    object,
    uuid(<IID_IEnum <ELT_T>>),   // IID_IEnum<ELT_T>
    pointer_default(unique)
]
interface IEnum<ELT_T> : IUnknown
{
    HRESULT Next( [in] ULONG celt, [out] IUnknown **rgelt, [out] ULONG *pceltFetched );
    HRESULT Skip( [in] ULONG celt );
    HRESULT Reset( void );
    HRESULT Clone( [out] IEnum<ELT_T>**ppenum );
}
```

A typical use of an enumerator is the following.

```
//Somewhere there's a type called "String"
```

[41] "elt" by itself in the function prototypes is just "element"

```
typedef char * String;

//Interface defined using template syntax
typedef IEnum<char *>   IEnumString;
...
interface IStringManager {
      virtual IEnumString* EnumStrings(void) = 0;
      };
...
void SomeFunc(IStringManager * pStringMan)      {
      char *          psz;
      IEnumString * penum;
      penum=pStringMan->EnumStrings();
      while (S_OK==penum->Next(1, &psz, NULL))
            {
            //Do something with the string in psz and free it
            }
      penum->Release();
      return;
      }
```

.*1* IEnum::Next

HRESULT IEnum::Next(celt, rgelt, pceltFetched)

Attempt to get the next celt items in the enumeration sequence, and return them through the array pointed to by rgelt. If fewer than the requested number of elements remain in the sequence, then just return the remaining ones; the actual number of elements returned is passed through *pceltFetched (unless it is NULL). If the requested celt elements are in fact returned, then return S_OK; otherwise return S_FALSE. An error condition other than simply "not that many elements left" will return an SCODE which is a failure code rather than one of these two success values.

To clarify:

- If S_OK is returned, then on exit the all celt elements requested are valid and returned in rgelt.
- If S_FALSE is returned, then on exit only the first *pceltFetched entries of rgelt are valid. The contents of the remaining entries in the rgelt array are indeterminate.
- If an error value is returned, then on exit no entries in the rgelt array are valid; they are all in an indeterminate state.

Argument	Type	Description
celt	ULONG	The number of elements that are to be returned.
rgelt[42]	ELT_T*	An array of size at least celt in which the next elements are to be returned.
pceltFetched	ULONG*	May be NULL if celt is one. If non-NULL, then this is set with the number of elements actually returned in rgelt.

Return Value	Meaning
S_OK	Success. The requested number of elements were returned.
S_FALSE	Success. Fewer than the requested number of elements were returned.
E_UNEXPECTED	An unknown error occurred.

.2 IEnum::Skip

HRESULT IEnum::Skip(celt)

Attempt to skip over the next celt elements in the enumeration sequence. Return S_OK if this was accomplished, or S_FALSE if the end of the sequence was reached first.

[42] Think of "rgelt" as short for "range of elt", signifying an array.

Argument	Type	Description
celt	ULONG	The number of elements that are to be skipped.

Return Value	Meaning
S_OK	Success. The requested number of elements were skipped.
S_FALSE	Success. Some skipping was done, but the end of the sequence was hit before the requested number of elements could be skipped.
E_UNEXPECTED	An unknown error occurred.

.3 IEnum::Reset

HRESULT IEnum::Reset(void)

Reset the enumeration sequence back to the beginning.

Note that there is no intrinsic guarantee that *exactly* the same set of objects will be enumerated the second time as was enumerated the first. Though clearly very desirable, whether this is the case or not is dependent on the collection being enumerated; some collections will simply find it too expensive to maintain this condition. Consider enumerating the files in a directory, for example, while concurrent users may be making changes.

Return Value	Meaning
S_OK	Success. The enumeration was reset to its beginning.
E_UNEXPECTED	An unknown error occurred.

.4 IEnum::Clone

HRESULT IEnum::Clone(ppenum)

Return another enumerator which contains exactly the same enumeration state as this one. Using this function, a client can remember a particular point in the enumeration sequence, then return to it at a later time. Notice that the enumerator returned is of the same actual interface as the one which is being cloned.

Caveats similar to the ones found in IEnum::Reset regarding enumerating the same sequence twice apply here as well.

Argument	Type	Description
ppenum	IEnum<ELT_T>**	The place in which to return the clone enumerator.

Return Value	Meaning
S_OK	Success. The enumeration was reset to its beginning.
E_UNEXPECTED	An unknown error occurred.

6 Designing and Implementing Objects

Objects can come in all shapes and sizes and applications will implement objects for various purposes with or without assigning the class a CLSID. COM servers implement objects for the sake of serving them to clients. In some cases, such as data change notification, a client itself will implement a classless object to essentially provide callback functions for the server object.

In all cases there is only one requirement for all objects: implement at least the IUnknown interface. An object is not a COM object unless it implements at least one interface which at minimum is IUnknown. Not all objects even need a unique identifier, that is, a CLSID. In fact, *only* those objects that wish to allow COM to locate and launch their implementations really need a CLSID. All other objects do not.

IUnknown implemented by itself can be useful for objects that simply represent the existence of some resource and control that resource's lifetime without providing any other means of manipulating that resource. By and large, however, most interesting objects will want to provide more services, that is, additional interfaces through which to manipulate the object. This all depends on the purpose of the object and the context in which clients (or whatever other agents) use it. The object may wish to provide some data exchange capabilities by implementing IDataObject, or may wish to indicate the contract through which it can serialize it's information by implementing one of the IPersist flavors of interfaces. If the object is a moniker, it will implement an interface called IMoniker that we'll see in Chapter 9. Objects that are used specifically for

handling remote procedure calls implement a number of specialized interfaces themselves as we'll see in Chapter 7.

The bottom line is that you decide what functionality the object should have and implement the interface that represents that functionality. In some cases there are no standard interfaces that contain the desired functionality in which case you will want to design a custom interface. You may need to provide for remoting that interface as described in Chapter 7.

The following chapters that discuss COM clients and servers use as an example an object class designed to render ASCII text information from text stored in files. This object class is called "TextRender" and it has a CLSID of {12345678-ABCD-1234-5678-9ABCDEF00000}[43] defined as the symbol CLSID_TextRender in some include file. Note again that an object class does not have to have an associated CLSID. This example has one so we can use it to demonstrate COM clients and servers in Chapters 5 and 6.

The TextRender object can read and write text to and from a file, and so implements the IPersistFile interface to support those operations. An object can be initialized (see Chapter 5, "Initializing the Object") with the contents of a file through IPersistFile::Load. The object class also supports rendering the text data into straight text as well as graphically as metafiles and bitmaps. Rendering capabilities are handled through the IDataObject interface, and IDataObject::SetData when given text forms a second initializing function.[44] The operation of TextRender objects is illustrated in Figure 3-4:

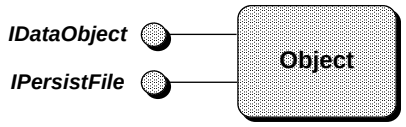


Figure 3-4: An object with *IDataObject* and *IPersistFile* Interfaces.

The "Object Reusability" section of Chapter 6 will show how we might implement this object when another object that provides some the desired functionality is available for reuse. But for now, we want to see how to implement this object on its own.

.1 Implementing Interfaces: Multiple Inheritance

There are two different strategies for implementing interfaces on an object: multiple inheritance and interface containment. Which method works best for you depends first of all on your language of choice (languages that don't have an inheritance notion cannot support multiple inheritance, obviously) but if you are implementing an object in C++, which is a common occurrence, your choice depends on the object design itself.

Multiple inheritance works best for most objects. Declaring an object in this manner might appear as follows:

```
class CTextRender : public IDataObject, public IPersistFile {
    private:
        ULONG       m_cRef;         //Reference Count
        char *      m_pszText;      //Pointer to allocated text
        ULONG       m_cchText;      //Number of characters in m_pszText

        //Other internal member functions here

    public:
        [Constructor, Destructor]

        /*
         * We must override all interface member functions we
         * inherit to create an instantiatable class.
         */

        //IUnknown members shared between IDataObject and IPersistFile
        HRESULT QueryInterface(REFIID iid, void ** ppv);
        ULONG AddRef(void);
```

[43] Do not use this CLSID for your own purposes–it is simply an example. See the section "Identifying and Registering the Object" below.

[44] In other words, the client may initialize the object by telling it to read text from a file or by handing text to it through IDataObject::SetData. Either way, the object now has some text to render graphically or to save to a file.


```
        ULONG Release(void);

        //IDataObject Members overrides
        HRESULT GetData(FORAMTETC *pFE, STGMEDIUM *pSTM);
        [Other members]
        ...

        //IPersistFile Member overrides
        HRESULT Load(char * pszFile, DWORD grfMode);
        [Other members]
        ...
};
```

This object class inherits from the interfaces it wishes to implement, declares whatever variables are necessary for maintaining the object state, and overrides all the member functions of all inherited interfaces, remembering to include the IUnknown members that are present in all other interfaces. The implementation of the single QueryInterface function of this object would use typecasts to return pointers to different vtbl pointers:

```
HRESULT CTextRender::QueryInterface(REFIID iid, void ** ppv) {
    *ppv=NULL;

    //This code assumes an overloaded == operator for GUIDs exists
    if (IID_IUnknown==iid)
        *ppv=(void *)(IUnknown *)this;

    if (IID_IPersitFile==iid)
        *ppv=(void *)(IPersistFile *)this;

    if (IID_IDataObject==iid)
        *ppv=(void *)(IDataObject *)this;

    if (NULL==*ppv)
        return E_NOINTERFACE;          //iid not supported.

    // Any call to anyone's AddRef is our own, so we can just call that directly
    AddRef();
    return NOERROR;
    }
```

This technique has the advantage that all the implementation of all interfaces is gathered together in the same object and all functions have quick and direct access to all the other members of this object. In addition, there only needs to be one implementation of the IUnknown members. However, when we deal with aggregation in Chapter 6 we will see how an object might need a *separate* implementation of IUnknown by itself.

.2 Implementing Interfaces: Interface Containment

There are at times reasons why you may not want to use multiple inheritance for an object implementation. First, you may not be using C++. That aside, you may want to individually track reference counts on each interface separate from the overall object for debugging or for resource management purposes—reference counting is from a client perspective an interface-specific operation. This can uncover problems in a client you might also be developing, exposing situations where the client is calling AddRef through one interface but matching it with a Release call through a different interface. The third reason that you would use a different method of implementation is when you have two interfaces with the same member function names with possibly identical function signatures or when you want to avoid function overloading. For example, if you wanted to implement IPersistFile, IPersistStorage, and IPersistStream on an object, you would have to write overloaded functions for the Load and Save members of each which might get confusing. Worse, if two interface designers should happen to define interfaces that have like-named methods with like parameter lists but incompatible semantics, such overloading isn't even possible: two separate functions need to be implemented, but C++ unifies the two method definitions. Note that as in general interfaces may be defined by independent parties that do not communicate with each other, such situations are inevitable.

The other implementation method is to use "interface implementations" which are separate C++ objects that each inherit from and implement one interface. The real object itself singly inherits from IUnknown and maintains (or contains) pointers to each interface implementation that it creates on initialization. This keeps all the interfaces separate and distinct. An example of code that uses the containment policy follows:

```
class CImplPersistFile : public IPersistFile {
    private:
        ULONG       m_cRef;         //Interface reference count for debugging

        //"Backpointer" to the actual object.
        class CTextRender * m_pObj;

    public:
        [Constructor, Destructor]

        //IUnknown members for IPersistFile
        HRESULT QueryInterface(REFIID iid, void ** ppv);
        ULONG AddRef(void);
        ULONG Release(void);

        //IPersistFile Member overrides
        HRESULT Load(char * pszFile, DWORD grfMode);
        [Other members]
        ...
    }

class CImplDataObject : public IDataObject
    private:
        ULONG       m_cRef;         //Interface reference count for debugging

        //"Backpointer" to the actual object.
        class CTextRender * m_pObj;

    public:
        [Constructor, Destructor]

        //IUnknown members for IDataObject
        HRESULT QueryInterface(REFIID iid, void ** ppv);
        ULONG AddRef(void);
        ULONG Release(void);

        //IPersistFile Member overrides
        HRESULT GetData(FORMATETC *pFE,STGMEDIUM *pSTM);
        [Other members]
        ...
    }


class CTextRender : public IUnknown
    {
    friend class CImplDataObject;
    friend class CImplPersistFile;

    private:
        ULONG       m_cRef;         //Reference Count
        char *      m_pszText;      //Pointer to allocated text
        ULONG       m_cchText;      //Number of characters in m_pszText

        //Contained interface implementations
        CImplPersistFile * m_pImplPersistFile;
        CImplDataObject *   m_pImplDataObject;

        //Other internal member functions here

    public:
        [Constructor, Destructor]

        HRESULT QueryInterface(REFIID iid, void ** ppv);
        ULONG AddRef(void);
        ULONG Release(void);
```



```
};
```

In this technique, each interface implementation must maintain a backpointer to the real object in order to access that object's variables (normally this is passed in the interface implementation constructor). This may require a *friend* relationship (in C++) between the object classes; alternatively, these friend classes can be implemented as nested classes in CTextRender.

Notice that the IUnknown member functions of each interface implementation do not need to do anything more than delegate directly to the IUnknown functions implemented on the CTextRender object. The implementation of QueryInterface on the main object would appear as follows:

```
HRESULT CTextRender::QueryInterface(REFIID iid, void ** ppv)
    {
    *ppv=NULL;

    //This code assumes an overloaded == operator for GUIDs exists
    if (IID_IUnknown==iid)
        *ppv=(void *)(IUnknown *)this;

    if (IID_IPersitFile==iid)
        *ppv=(void *)(IPersistFile *)m_pImpIPersistFile;

    if (IID_IDataObject==iid)
        *ppv=(void *)(IDataObject *)m_pImpIDataObject;

    if (NULL==*ppv)
        return E_NOINTERFACE;          //iid not supported.

    //Call AddRef through the returned interface
    ((IUnknown *)ppv)->AddRef();
    return NOERROR;
    }
```

This sort of delegation structure makes it very easy to redirect each interface's IUnknown members to some other IUnknown, which is necessary in supporting aggregation as explained in Chapter 6. But overall the implementation is not much different than multiple inheritance and both methods work equally well. Containment of interface implementation is more easily translatable into C where classes simply become equivalent structures, if for any reason such readability is desirable (such as making the source code more comprehensible to C programmers who do not know C++ and do not understand multiple inheritance). In the end it really all depends upon your preferences and has no significant impact on performance nor development.

This page intentionally left blank.

4. COM Applications

All applications, that is, running programs that define a task or a process be they client or servers, have specific responsibilities. This chapter examines the roles and responsibilities of all COM applications and the necessary COM library support functions for those responsibilities.

In short, any application that makes use of COM, client or server, has three specific responsibilities to insure proper operation with other components:

1. On application startup, verify that the COM Library version is new enough to support the functionality expected by the application. In general, an application can use an updated version of the library but not an older one or one that has undergone a major version change.
2. On application startup, initialize the COM Library.
3. On application shutdown, uninitialize the COM Library to allow it to free resources and perform any cleanup operations as necessary.

Each of these responsibilities requires support from the COM Library itself as detailed in the following sections. For convenience, initialization and uninitialization are described together. Additional COM Library functions related to initialization and memory management are also given in this chapter.

1 Verifying the COM Library Version

The COM Library defines a major version number and a minor version number and provide these in a header file that is compiled with the COM application. Any application must then compare these compiled numbers with the version of the available library and if the available library is incompatible the application cannot use COM. Similarly, a DLL should check the library version in its initialization code and fail loading if the library is incompatible or otherwise disable its COM functionality. The current major and minor version numbers are retrieved from COM Library with the function CoBuildVersion.

.1 CoBuildVersion

DWORD CoBuildVersion(void)

Return the major and the minor version number of the Component Object Model library.

Argument	Type	Description
return value	DWORD	A 32 bit value whose high-order 16 bits are the major version number (rmm) and whose low-order 16 bits are the minor version number (rup).

An application or DLL can run against only one major version of the COM Library but can run against any minor version (possibly disabling specific minor features that are not available in a builds before a given minor number). Therefore during startup (initialization for DLLs), all COM applications must include code similar to the following:

```
DWORD      dwBuildVersion;
dwBuildVersion=CoBuildVersion();
if (HIWORD(dwBuildVersion)!=rmm)
    //Error:   Can't run against wrong major version
if (LOWORD(dwBuildVersion) < rup)
    //Disable features dependent on the rup version of COM (or simply fail)
//Continue initialization
```

2 Library Initialization / Uninitialization

Once the application has determined that it can run against the currently available version of the COM Library, it must initialize the library through a function called CoInitialize. Calls made to CoInitialize must be matched with calls to CoUninitialize to allow the COM Library to perform any final cleanup.

.1 CoInitialize

HRESULT CoInitialize(pReserved)

Initialize the Common Object Model library so that it can be used. With the exception of CoBuildVersion, this function must be called by applications before any other function in the library. Calls to CoInitialize must be balanced by corresponding calls to CoUninitialize. Typically, CoInitialize is called only once by the process that wants to use the COM library, although multiple calls can be made. Subsequent calls to CoInitialize return S_FALSE.

Argument	Type	Description
pReserved	void*	Reserved for future use. Presently, must be NULL.

Return Value	Meaning
S_OK	Success. Initialization has succeeded. This was the first initialization call in this process.
S_FALSE	Success. Initialization has succeeded, but this was not the first initialization call in this process.
E_UNEXPECTED	An unknown error occurred.

.2 CoUninitialize

void CoUninitialize(void)

Shuts down the Component Object Model library, thus freeing any resources that it maintains. Since CoInitialize and CoUninitialize calls must be balanced, only the CoUninitialize call that corresponds to the CoInitialize call that actually did the initialization will uninitialize the library.

3 Memory Management

As was articulated earlier in this specification, when ownership of allocated memory is passed through an interface, COM requires[45] that the memory be allocated with a specific "task allocator." Most general purpose access to the task allocator is provided through the IMalloc interface instance returned from CoGetMalloc. Simple shortcut allocation and freeing APIs are also provided in the form of CoTaskMemAlloc and CoTaskMemFree.

.1 IMalloc Interface

The *IMalloc* interface is an abstraction of familiar memory-allocation primitives that fit into the COM interface model. Like all other interface, it is derived from IUnknown and correspondingly includes the AddRef, Release, and QueryInterface member functions. The first three IMalloc-specific functions in this interface are merely simple abstractions of the familiar C-library functions malloc, realloc, and free.

```
[
    local,
    object,
    uuid(00000002-0000-0000-C000-000000000046)
]
interface IMalloc : IUnknown {
    void *    Alloc([in] ULONG cb);
    void *    Realloc([in] void * pv, [in] ULONG cb);
    void      Free([in] void* pv);
    ULONG     GetSize([in] void * pv);
    int       DidAlloc([in] void * pv);
    void      HeapMinimize(void);
    };
```

[45] In general, though, precisely, one can invent interfaces which choose to violate this rule. However, such interfaces are, for example, unlikely to have their remoting proxies and stubs generated with common tools.

.1 IMalloc::Alloc

void * IMalloc::Alloc(cb)

Allocate a memory block of at least cb bytes. The initial contents of the returned memory block are undefined. Specifically, it is not guaranteed that the block is zeroed. The block actually allocated may be larger than cb bytes because of space required for alignment and for maintenance information. If cb is 0, Alloc allocates a zero-length item and returns a valid pointer to that item. This function returns NULL if there is insufficient memory available.

Callers must always check the return from the this function, even if the amount of memory requested is small.

Argument	Type	Description
cb	ULONG	The number of bytes to allocate.
return value	void *	The allocated memory block, or NULL if insufficient memory exists.

.2 IMalloc::Free

void IMalloc::Free(pv)

Deallocate a memory block. The pv argument points to a memory block previously allocated through a call to IMalloc::Alloc or IMalloc::Realloc. The number of bytes freed is the number of bytes with which the block was originally allocated (or reallocated, in the case of Realloc). After the call, the pv parameter is invalid, and can no longer be used. pv may be NULL, in which case this function is a no-op.

Argument	Type	Description
pv	void *	Pointer to the block to free. May be NULL.

.3 IMalloc::Realloc

void * IMalloc::Realloc(pv, cb)

Change the size of a previously allocated memory block. The pv argument points to the beginning of the memory block. If pv is NULL, Realloc functions in the same way as IMalloc::Alloc and allocates a new block of cb bytes. If pv is not NULL, it should be a pointer returned by a prior call to IMalloc::Alloc.

The cb argument gives the new size of the block in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes, although the new block may be in a different location. Because the new block can be in a new memory location, the pointer returned by Realloc is not guaranteed to be the pointer passed through the pv argument. If pv is not NULL and *cb* is 0, then the memory pointed to by pv is freed.

Realloc returns a void pointer to the reallocated (and possibly moved) memory block. The return value is NULL if the size is zero and the buffer argument is not NULL, or if there is not enough available memory to expand the block to the given size. In the first case, the original block is freed. In the second, the original block is unchanged.

The storage space pointed to by the return value is guaranteed to be suitably aligned for storage of any type of object. To get a pointer to a type other than void, use a type cast on the return value.

Argument	Type	Description
pv	void *	Pointer to the block to reallocate. May be NULL.
cb	ULONG	The new size in bytes to allocate. May be zero.
return value	void *	The reallocated memory block, or NULL.

.4 IMalloc::GetSize

ULONG IMalloc::GetSize(pv)

Return the size, in bytes, of the memory block allocated by a previous call to IMalloc::Alloc or IMalloc::Realloc on this memory manager.

Argument	Type	Description
pv	void *	The pointer to be tested. May be NULL, in which case -1 is returned.
return value	ULONG	The size of the allocated memory block

.5 IMalloc::DidAlloc

int IMalloc::DidAlloc(pv)

This function answers as whether or not the indicated memory pointer pv was allocated by the given allocator, if the allocator is able to determine that fact (many memory allocators will not be able to do so).

The values 1 (one) and 0 (zero) are returned as "did alloc" and "did not alloc" answers respectively; -1 (minus one) is returned if the IMalloc implementation is unable to determine whether it allocated the pointer or not.

Argument	Type	Description
pv	void *	The pointer to be tested. May be NULL, in which case -1 is returned.
return value	int	-1, 0, 1

.6 IMalloc::HeapMinimize

void IMalloc::HeapMinimize()

Minimize the heap as much as possible for this allocator by, for example, releasing unused memory in the heap to the operating system. This is useful in cases when a lot of allocations have been freed (using IMalloc::Free) and the application wants to release the freed memory back to the operating system so that it is available for other purposes.

.2 COM Library Memory Management Functions

.1 CoGetMalloc

HRESULT CoGetMalloc(dwMemContext, ppMalloc)

This function retrieves from the COM library either the task memory allocator an optionally-provided shared memory allocator. The particular allocator of interest is indicated by the dwMemContext parameter. Legal values for this parameter are taken from the enumeration MEMCTX:

```
typedef enum tagMEMCTX {
        MEMCTX_TASK = 1,                // task (private) memory
        MEMCTX_SHARED = 2,              // shared memory (between processes)
        MEMCTX_MACSYSTEM = 3,           // on the mac, the system heap
        // these are mostly for internal use...
        MEMCTX_UNKNOWN = -1,            // unknown context (when asked about it)
        MEMCTX_SAME = -2,               // same context (as some other pointer)
    } MEMCTX;
```

MEMCTX_TASK returns the task allocator. If CoInitialize has not yet been called, NULL we be stored in ppMalloc and CO_E_NOTINITIALIZED returned from the function.

MEMCTX_SHARED returns an optionally-provided shared allocator; if the shared allocator is not supported, E_INVALIDARG is returned. When supported, the shared allocator returned by this function is an COM-provided implementation of IMalloc interface, one which allocates memory in such a way that it can be accessed by other process on the current machine simply by conveying the pointer to said applications.[46] Further, memory allocated by this shared allocator in one application may be freed by the shared allocator in another. Except when a NULL pointer is passed, the shared memory allocator never answers -1 to IMalloc::DidAlloc; it always indicates that either did or did not allocate the passed pointer.

[46] That is, the memory resides at the same address in all processes.

Argument	Type	Description
dwMemContext	DWORD	A value from the enumeration MEMCTX.
ppMalloc	IMalloc **	The place in which the memory allocator should be returned.

Return Value	Meaning
S_OK	Success. The requested allocator was returned.
CO_E_NOTINITIALIZED	The COM library has not been initialized.
E_INVALIDARG	An invalid argument was passed.
E_UNEXPECTED	An unknown error occurred.

.2 CoGetCurrentProcess

DWORD CoGetCurrentProcess(void)

Return a value unique to the current process. More precisely, return a value unique to the current process to the degree that it will not be reused until 2^{32} further processes have been created on the current workstation.

Argument	Type	Description
return value	DWORD	A value unique to the current process.

.3 CoTaskMemAlloc

LPVOID CoTaskMemAlloc(cb)

Semantically identical to retrieving the current task allocator with CoGetMalloc, invoking IMalloc::Alloc on that pointer with the same parameters, then releasing the IMalloc pointer.

Argument	Type	Description
cb	ULONG	The number of bytes to allocate.
return value	void *	The allocated memory block, or NULL if insufficient memory exists.

.4 CoTaskMemFree

void CoTaskMemFree(pv)

Semantically identical to retrieving the current task allocator with CoGetMalloc, invoking IMalloc::Free on that pointer with the same parameters, then releasing the IMalloc pointer.

Argument	Type	Description
pv	void *	Pointer to the block to free. May be NULL.

.5 CoTaskMemRealloc

void CoTaskMemRealloc(pv, cb)

Semantically identical to retrieving the current task allocator with CoGetMalloc, invoking IMalloc::Realloc on that pointer with the same parameters, then releasing the IMalloc pointer.

Argument	Type	Description
pv	void *	Pointer to the block to reallocate. May be NULL.
cb	ULONG	The new size in bytes to allocate. May be zero.
return value	void *	The reallocated memory block, or NULL.

4 Memory Allocation Example

An object may need to pass memory between it and the client at some point in the object's lifetime—this applies to in-process as well as out-of-process servers. When such a situation arises the object must use the task allocator as described in Chapter 2. That is, the object must allocate memory whose ownership is transferred from one party to another through an interface function by using the local task allocator.

CoGetMalloc provides a convenient way for objects to allocate working memory as well. For example, when the TextRender object (see Chapter 3, "Designing and Implementing Objects") under consideration in this

document loads text from a file in the function IPersistFile::Load (that is, CTextRender::Load) it will want to make a memory copy of that text. It would use the task allocator for this purpose as illustrated in the following code (unnecessary details of opening files and reading data are omitted for simplicity):

```
//Implementation of IPersistFile::Load
HRESULT CTextRender::Load(char *pszFile, DWORD grfMode) {
    int         hFile;
    DWORD       cch;
    IMalloc * pIMalloc;
    HRESULT     hr;

    /*
     * Open the file and seek to the end to set the
     * cch variable to the length of the file.
     */

    hr=CoGetMalloc(MEMCTX_TASK, &pIMalloc);

    if (FAILED(hr))
        //Close file and return failure

    psz=pIMalloc->Alloc(cch);
    pIMalloc->Release();

    if (NULL==psz)
        //Close file and return failure

    //Read text into psz buffer and close file

    //Save memory pointer and return success
    m_pszText=psz;
    return NOERROR;
    }
```

If an object will make many allocations throughout it's lifetime, it makes sense to call CoGetMalloc once when the object is created, store the IMalloc pointer in the object (m_pIMalloc or such), and call IMalloc::Release when the object is destroyed. Alternatively, the APIs CoTaskMemAlloc and its friends may be used.

5. COM Clients

As described in earlier chapters, a COM Client is simply any piece of code that makes use of another object through that object's interfaces. In this sense, a COM Client may itself be a COM Server acting in the capacity of a client by virtue of using (or reusing) some other object.

If the client is an application, that is, an executable program as opposed to a DLL, then it must follow all the requirements for a COM Application as detailed in Chapter 4. That aside, clients have a number of ways to actually get at an object to use as discussed in a previous chapter. The client may call a specific function to create an object, it might ask an existing object to create another, or it might itself implement an object to which some other code hands yet another object's interface pointer. Not all of these objects must have CLSID.

This chapter, however, is concerned with those clients that want to create an object based on a CLSID, because at some point or another, many operations that don't directly involve a CLSID do eventually resolve to this process. For example, moniker binding internally uses a CLSID but shields clients from that fact. In any case, whatever client code uses a CLSID will generally perform the following operations in order to make use of an object:

1. Identify the class of object to use.
2. Obtain the "class factory" for the object class and ask it to create an uninitialized instance of the object class, returning an interface pointer to it.
3. Initialize the newly created object by calling an initialization member function of the "initialization interface," that is, one of a generally small set of interfaces that have such functions.
4. Make use of the object which generally includes calling QueryInterface to obtain additional working interface pointers on the object. The client must be prepared for the potential absence of a desired interface.
5. Release the object when it is no longer needed.

The following sections cover the functions and interfaces involved in each of these steps. In addition, the client may want to more closely manage the loading and unloading of server modules (DLLs or EXEs) for optimization purposes, so this chapter includes a section of such management.

As far as the client is concerned, the COM Library exists to provide fundamental implementation locator and object creation services and to handle remote procedure calls to local or remote objects (in addition to memory management services, of course). How a server facilitates these functions is the topic of Chapter 6.

Before examining the details of object creating and manipulation, realize that after the object is created and the client has its first interface pointer to that object, the *client cannot distinguish an in-process object from a local object from a remote object* by virtue of examining the interface pointer or any other interfaces on that object. That is, all objects appear identically to the client such that after creation, all requests made to the object's services are made by calling interface member functions. Period. There are not special exceptions that a client must make at run-time based on the distance of the object in question. The COM Library provides any underlying glue to insure that a call made to a local or remote object is, in fact, marshaled properly to the other process or the other machine, respectively. *This operation is transparent to the client*, who always sees any call to an object as a function call to the objects interfaces as if that object were in-process. This consistency is a key benefit for COM clients as it can treat all objects identically regardless of their actual execution context. If you are interested in understanding how this transparency is achieved, please see Chapter 7, "Communicating via Interfaces: Remoting" for more details. There you will find that all clients do, in fact, always call an in-process object first, but in local and remote cases that in-process object is just a proxy that takes care of generating a remote procedure call.

1 Identifying the Object Class

A central feature of COM is that a client can opaquely locate and dynamically load the specific piece code that knows how to manipulate a specific class of object. This is accomplished through the COM-supplied implementation locator services through which COM associates a class identifier, that is, CLSID, with the

server module for that object class. Therefore the COM Library is responsible for defining how this association occurs which usually involves a system-wide persistent registry of CLSIDs and their corresponding servers. For example, under Microsoft Windows the COM Library stores the pathnames of in-process server DLLs and local server EXEs in the system registry under the text string of the object's CLSID.

The practical upshot of all this for client applications is that the client need not know nor care how this information is maintained or how the COM Library performs the association from CLSID to server. In the same manner the client need not perform any additional work to establish communication with a local or remote object as such steps are also handled in COM transparently.

This does leave the question of how the client determines what CLSID to hand to COM in the first place. There is no single answer, for it varies from situation to situation. In some cases the object to use has a well-known and fixed CLSID that is compiled into the client application. In other cases the client may have a constant text string (compiled, that is) that represents a CLSID and uses some means to associate that name with a CLSID. Another example may be that the client has some previously saved information that directly or indirectly translates to a CLSID, such as a piece of storage (where the CLSID is serialized into a stream) or a moniker (where the CLSID is implied by the data which the moniker references). Finally, there may be some means through which the client displays a list of available objects to the end-user where each item in the list corresponds to a specific CLSID. In such cases the list is generated by browsing the registry for all existing object classes. Other examples are clearly possible, particularly in network situations.

2 Creating the Object

Given a CLSID the client must now create an object of that class in order to make use of its services. It does so using two steps:

1. Obtain the "class factory" for the CLSID.
2. Ask the class factory to instantiate an object of the class, returning an interface pointer to the client.

After these steps, illustrated in Figure 5-1, the client is free to do whatever it wishes with the object through whatever interfaces the object supports. In fact, *everything* done with the object is accomplished through calls to interface member functions—APIs that seems to affect objects through other means are merely wrappers to common sequences of interface calls.

Before examining each of these steps, let's take a look at what a class factory is in the first place.

.1 The Class Factory Object: IClassFactory Interface

The class factory is another object itself that exists to *manufacture* objects (hence the name "factory") of a specific class (hence the qualifier "class").[47] A class factory object is implemented by a server module, either a DLL or EXE, and supports the IClassFactory interface described below. For the purposes of COM Clients, the IClassFactory interface is and interface on an object used by a client. For information on implementation, see Chapter 6, "COM Servers."

[47] Note that IClassFactory would be more appropriately be named IObjectFactoy since using it one creates objects, not classes. But IClassFactory remains for historical reasons.

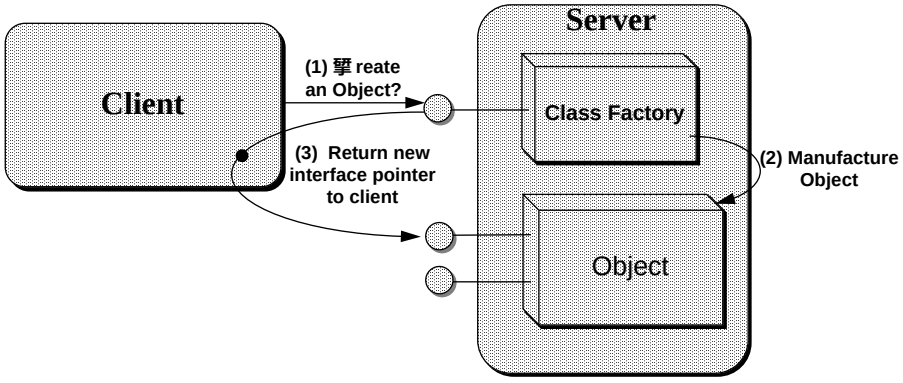


Figure 5-1 A client asks a class factory in the server to create an object.

The IClassFactory interface is implemented by COM servers on a "class factory" object for the purpose of creating new objects of a particular class. The interface also provides for a COM client to keep the server in memory even when it is not servicing any object. A class factory has a one-to-one correspondence with a CLSID (although actual implementations can be made generic to service multiple classes if the COM server so chooses).

```
[
    object,
    uuid(00000001-0000-0000-C000-000000000046), // IID_IClassFactory
    pointer_default(unique)
]
interface IClassFactory : IUnknown
{
      HRESULT CreateInstance([in] IUnknown * pUnkOuter, [in] REFIID iid, [out] void * ppv);
      HRESULT LockServer([in]BOOL fLock);
}
```

.1 IClassFactory::CreateInstance

HRESULT IClassFactory::CreateInstance(pUnkOuter, iid, ppvObject)

Create an uninitialized instance, that is, object, of the class associated with the class factory, returning an interface pointer of type iid on the object to the caller in the out-parameter ppvObject.

If the object is being created as part of an aggregate—that is, the client of the object in this case is also an object server itself—then pUnkOuter contains the IUnknown pointer to the "outer unknown." See "Object Reusability" in Chapter 6 for more information. Class implementations need to be consciously designed to be aggregatable and accordingly not all classes are so designed.

Argument	Type	Description
pUnkOuter	IUnknown *	The controlling unknown of the aggregate object if this object is being created as part of an aggregate. If NULL, then the object is not being aggregated, which is the case when the object is being created from a pure client. If non-NULL and the class does not support aggregation, then the function returns CLASS_E_NOAGGREGATION.
iid	REFIID	The identifier of the first interface desired by the caller through which it will communicate with the object; usually the "initialization interface."
ppv	void **	The place in which the first interface pointer is to be returned.

Return Value	Meaning
S_OK	Success. A new instance was created.
E_NOAGGREGATION	Use of aggregation was requested, but this class does not support it.
E_OUTOFMEMORY	Memory could not be allocated to service the request.
E_UNEXPECTED	An unknown error occurred.

.2 IClassFactory::LockServer

HRESULT IClassFactory::LockServer(fLock)

This function can be called by a client to keep a server in memory even when it is servicing no objects. Normally a server will unload itself (an EXE server) or allow the COM library to unload it (a DLL server) when the server has no objects left to serve. If the client so desires, it can lock the server in memory to prevent it from being loaded and unloaded multiple times, which can improve performance of object instantiations. Most clients have no need to call this function. It is present primarily for the benefit of sophisticated clients with special performance needs from certain classes.

It is an error to call LockServer(TRUE) and then call Release without first releasing the lock with LockServer(FALSE). Whoever locks the server is responsible for unlocking it, and once the class factory is released, there is no mechanism by which the caller can be guaranteed to later connect to the same class factory. All calls to IClassFactory::LockServer must be counted, not only the last one. Calls will be balanced; that is, for every LockServer(TRUE) call, there will be a LockServer(FALSE) call. If the lock count and the class object reference count are both zero, the class object can be freed.

For more information on the use of LockServer, see the "Server Management" section below. For more information on implementing this function, see Chapter 6 under "The Class Factory: Implementation and Exposure."

Argument	Type	Description
fLock	BOOL	True if a lock is being added to the class factory; false if one is being removed.

Return Value	Meaning
S_OK	Success.
E_UNEXPECTED	An unknown error occurred.

3 Obtaining the Class Factory Object for a CLSID

Now that we understand what a class factory is and what functions it performs through the IClassFactory interface we can examine how a client obtains the class factory. This depends only slightly on whether the object in question is in-process, local, or remote. For the most part, all cases are handled through the same implementation locator service in the COM library and the same API functions. The implications are greater for servers as shown in Chapter 6.

For all objects on the same machine as the client, including object handlers, the client generates a call to the COM Library function CoGetClassObject. This function, described below, does whatever is necessary to obtain a class factory object for the given CLSID and return one of that class factory's interface pointers to the client. After that the client may calls IClassFactory::CreateInstance to instantiate objects of the class.

We say here that the client must *generate* a call to CoGetClassObject because it is not always necessary to call this function directly. When a client only wants to create a single object of a given class there is no need to go through the process of calling CoGetClassObject, IClassFactory::CreateInstance, and IClassFactory::Release. Instead it can use API function CoCreateInstance described below which conveniently wraps these three more fundamental steps into one function.

.1 CoGetClassObject

HRESULT CoGetClassObject(clsid, grfContext, pServerInfo, iid, ppv)

Locate and connect to the class factory object associated with the class identifier clsid. If necessary, the COM Library dynamically loads executable code in order to accomplish this. The interface by which the caller wishes to talk to the class factory object is indicated by iid; this is usually IID_IClassFactory but can, of course, be any other object-creation interface.[48] The class factory's interface is returned in ppv with one reference count on it on behalf of the caller, that is, the caller is responsible for calling Release after it has finished using the class factory object.

[48] For example, the remoting architechture described in Chapter 7 uses a different type of "factory" interface.

Different pieces of code can be associated with one CLSID for use in different execution contexts such as in-process, local, or object handler. The context in which the caller is interested is indicated by the grfContext parameter, a group of flags taken from the enumeration CLSCTX:

```
typedef enum tagCLSCTX {
    CLSCTX_INPROC_SERVER      = 1,
    CLSCTX_INPROC_HANDLER     = 2,
    CLSCTX_LOCAL_SERVER       = 4,
    CLSCTX_REMOTE_SERVER      = 16.
    } CLSCTX;
```

The several contexts are tried in the sequence in which they are listed here. Multiple values may be combined (using bitwise OR) indicating that multiple contexts are acceptable to the caller:

```
#define CLSCTX_INPROC    (CLSCTX_INPROC_SERVER | CLSCTX_INPROC_HANDLER)
#define CLSCTX_SERVER    (CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER | CLSCTX_REMOTE_SERVER)
#define CLSCTX_ALL       (CLSCTX_INPROC_SERVER | CLSCTX_INPROC_HANDLER | CLSCTX_LOCAL_SERVER |
        CLSCTX_REMOTE_SERVER)
```

These context values have the following meanings which apply to all remote servers as well:

Value	Action Taken by the COM Library
CLSCTX_INPROC_SERVER	Load the in-process code (DLL) which creates and completely manages the objects of this class. If the DLL is on a remote machine, invoke a surrogate server as well to load the DLL.
CLSCTX_INPROC_HANDLER	Load the in-process code (DLL) which implements client-side structures of this class when instances of it are accessed remotely. An object handler generally implements object functionality which can only be implemented from an in-process module, relying on a local server for the remainder of the implementation.[49]
CLSCTX_LOCAL_SERVER	Launch the separate-process code (EXE) which creates and manages the objects of this class.[50]
CLSCTX_REMOTE_SERVER	Launch the separate-process code (EXE) on another machine which creates and manages objects of this class.

The COM Library should attempt to load in-process servers first, then in-process handlers, then local servers, then remote servers. This order helps to minimize the frequency with which the library has to launch separate server applications which is generally a much more time-consuming operation than loading a DLL, especially across the network.

When specifying CLSCTX_REMOTE_SERVER, the caller may pass a COMSERVERINFO structure to indicate the machine on which to run the server module, which is defined as follows:

```
typedef struct tagCOMSERVERINFO {
    OLECHAR      *szRemoteSCMBindingHandle;
    } COMSERVERINFO;[51]
```

The COM Library implementation of this CoGetClassObject relies on the system registry to map the CLSID to the server module to load or launch, but this process is opaque to the client application. If, however, COM cannot make any association then the function fails with the code REGDB_E_CLASSNOTREG. If this function launches a server application it must wait until that server registers its class factory or until a

[49] For example, in OLE 2, built on top of COM, there is an interface called IViewObject through which a client can ask an object to draw its graphical presentation directly to a Windows device context (HDC) through IViewObject::Draw. However, an HDC cannot be shared between processes, so this interface can only be implemented inside as part of an in-process object. When an object server wishes to provide optimized graphical output but does not wish to completely implement the object in-process, it can use a lightweight object handler to implement just the drawing functionality where it must reside, relying on the local server for the full object implementation.

[50] In some cases the object server may already be running and may allow its class factory to be used multiple times in which case the COM Library simply establishes another connection to the existing class factory in that server, eliminating the need to launch another instance of the server applications entirely. While this can improve performance significantly, it is the option of the server to decide if its class factory is single- or multiple-use. See the function CoRegisterClassObject in Chapter 6 for more information.

[51] This abstraction is still under design.

time-out occurs (duration determined by COM, something on the order of a minute of processing time). See the CoRegisterClassObject function in Chapter 6 under "Exposing the Class Factory from Local Servers."

The arguments to this function are as follows:

Argument	Type	Description
clsid	REFCLSID	The class of the class factory to obtain.
grfContext	DWORD	The context in which the executable code is to run.
pServerInfo	COMSERVERINFO*	Identifies the machine on which to activate the executable code. Must be NULL when grfContext does not contain CLSCTX_REMOTE_SERVER. When NULL and grfContext contains CLSCTX_REMOTE_SERVER, COM uses the default machine location for this class.
iid	REFIID	The interface on the class factory object desired by the caller.
ppv	void **	The place in which to put the requested interface.

Return Value	Meaning
S_OK	Success.
REGDB_E_CLASSNOTREG	An implementation of the requested class could not be located.
E_OUTOFMEMORY	Memory could not be allocated to service the request.
E_UNEXPECTED	An unknown error occurred.

The following code fragment demonstrates how a client would call CoGetClassObject and create an in-process instance of the TextRender object with CLSID_TextRender using the class factory to request an IUnknown pointer for the object. In this example the client is explicitly limiting COM to use only in-process servers:

```
IClassFactory *     pCF;
IUnknown *          pUnkObj;
HRESULT             hr;

hr=CoGetClassObject(CLSID_TextRender, CLSCTX_INPROC_SERVER, NULL, IID_IClassFactory, (void *)pCF);
if (FAILED(hr))
    //Could not obtain class factory, creation fails completely.

/*
 * Create the object. If this call succeeds the pUnkObj will
 * be valid and have a reference count on it on behalf of the caller
 * which the caller must Release.
 */
hr=pCF->CreateInstance(NULL, IID_IUnknown, (void *)pUnkObj);

//Caller must call Release regardless of CreateInstance result
pCF->Release();

if (FAILED(hr))
    //Object creation failed: interface may not be supported

/*
 * Now use the object in whatever capacity the caller desires.
 * The first step will be initialization.
 */

//Release the object when finished with it.
pUnkObj->Release();
```

Since the process of calling CoGetClassObject, IClassFactory::CreateInstance, and IClassFactory::Release is so common in practice, the COM Library provides a wrapper API function for this sequence called CoCreateInstance. This allows the client to avoid the whole issue of class factory objects entirely. However, CoCreateInstance only creates one object at a time; if the client wants to create multiple objects of the same class at once, it is more efficient to obtain the class factory directly and call IClassFactory::CreateInstance multiple times, avoiding excess calls to CoGetClassObject and IClassFactory::Release.

.2 CoCreateInstance

HRESULT CoCreateInstance(clsid, pUnkOuter, grfContext, iid, ppvObj)

Create an uninitialized instance of the class clsid, asking for interface iid using the execution contexts given in grfContext. If the object is being used as part of an aggregation then pUnkOuter contains a pointer to the controlling unknown. These parameters behave as those of the same name in CoGetClassObject (clsid) and IClassFactory::CreateInstance (pUnkOuter, grfContext, iid, ppv),

CoCreateInstance is simply a wrapper function for CoGetClassObject and IClassFactory that is implemented (conceptually) as follows:

```
HRESULT CoCreateInstance(REFCLSID clsid, IUnknown * pUnkOuter,
    DWORD grfContext, REFIID iid, void * ppvObj)
    {
    IClassFactory * pCF;
    HRESULT         hr;

    hr=CoGetClassObject(clsid, grfContext, NULL, IID_IClassFactory, (void *)pCF);

    if (FAILED(hr))
        return hr;

    hr=pCF->CreateInstance(pUnkOuter, iid, (void *)ppv);
    pCF->Release();

    /*
     * If CreateInstance fails, ppv will be set to NULL. Otherwise
     * ppv has the interface pointer and hr contains NOERROR.
     */
    return hr;
    }
```

Argument	Type	Description
clsid	REFCLSID	The class of which an instance is desired
pUnkOuter	IUnknown*	The controlling unknown, if any.
grfContext	DWORD	The CLSCTX to be used.
iid	REFIID	The initialization interface desired
ppv	void**	The place at which to return the desired interface.

Return Value	Meaning
S_OK	Success.
Any error that can be returned from CoGetClassObject or IClassFactory::CreateInstance	Semantics as in those functions.
E_UNEXPECTED	An unknown error occurred.

.3 CoCreateInstanceEx

HRESULT CoCreateInstanceEx(clsid, pUnkOuter, grfContext, pServerInfo, dwCount, rgMultiQI)

Create an uninitialized instance of the class clsid on a specific machine, asking for a set of interface iids in pResult using the execution contexts given in grfContext. If the object is being used as part of an aggregation then pUnkOuter contains a pointer to the controlling unknown.

To help optimize round-trips to a remote machine during instantiation, this API allow the client to specify a set of interfaces to return on the object via the rgMultiQI array of MULTI_QI structures, defined as follows:

```
typedef struct tagMULTI_QI {
    REFIID      riid;     // interface to return
    void*       pvObj;    // location to return interface pointer
    HRESULT     hr;       // location to return result of QueryInterface for riid
    } MULTI_QI;
```

The semantics of using this API and passing a MULTI_QI array are identical to the following sequence of operations, but incur less overhead for the client, the server, and the network:

```
IClassFactory  *pCF;
IUnknown       *punk;
```

```
COMSERVERINFO csi;

CoGetClassObject(clsid, CLSCTX_SERVER, &csi, IID_IClassFactory, (void**)&pCF);
pCF->CreateInstance(NULL, IID_IUnknown, (void**)&punk);
for (DWORD i=0; i<dwCount; i++)
      rgMultiQI[I].hr = punk->QueryInterface(rgMultiQI[i].riid, &rgMultiQI[i].pvObj);
punk->Release();
```

Argument	Type	Description
clsid	REFCLSID	The class of which an instance is desired
pUnkOuter	IUnknown*	The controlling unknown, if any.
grfContext	DWORD	The CLSCTX to be used.
pServerInfo	COMSERVERINFO*	Identifies the machine on which to activate the executable code. Must be NULL when grfContext does not contain CLSCTX_REMOTE_SERVER. When NULL and grfContext contains CLSCTX_REMOTE_SERVER, COM uses the default machine location for this class.
dwCount	DWORD	The number of MULTI_QI structures in the rgMultiQI array.
rgMultiQI	MULTI_QI*	An array of MULTI_QI structures. On input, each element should be cleared and the riid member set to an IID being requested. On output, one or more of the interfaces may be retrieved, and individual pvObj members will be non-NULL.

Return Value	Meaning
S_OK	Success.
CO_S_NOTALLINTERFACES	Not all of dwCount interfaces requested in the MULTI_QI array were successfully retrieved. Examine individual pvObj members of MULTI_QI to determine exactly which interfaces were returned.
Any error that can be returned from CoGetClassObject or IClassFactory::CreateInstance	Semantics as in those functions.
E_UNEXPECTED	An unknown error occurred.

4 Initializing the Object

After the client has successfully created an object of a given class it must initialize that object. By definition, any new object created using IClassFactory::CreateInstance (or variant or wrapper thereof) is uninitialized. Initialization generally happens through a single call to a member function of the "initialization interface." This interface is usually the one requested by the client in its call to create the object, but this is not required. Before an object is initialized, the only calls that are guaranteed to work on the object (besides the initializing functions themselves) are the IUnknown functions (of any interface) unless otherwise explicitly specified in the definition of an interface. In addition, QueryInterface is only guaranteed to work for IUnknown and any initialization interface, but not guaranteed for a non-initialization interface.

Some objects will not require initialization before they are function through all of their interfaces. Those that do require initialization will define, either explicitly through documentation of the object or implicitly through the scenarios in which the object is used, which member of which interface can be used for initialization.

For example, objects that can serialize their persistent data to a file will implement the IPersistFile interface (see "Persistent Storage Interfaces for Objects" in Chapter 8). The function IPersistFile::Load, which instructs the object to load its data from a file, is the initialization function and IPersistFile is the initialization interface. Other examples are objects that can serialize to storages or streams, where the objects implement the initialization interfaces IPersistStorage or IPersistStream, respectively (again, see Chapter8). The Load functions in these interfaces are initialization functions as is IPersistStorage::InitNew, which initializes a new object with storage instead of loading a previously saved version.

5 Managing the Object

Once an object is initialized, it is entirely up to the client to determine what it intends to do with that object. It is often the case that the initializing interface is not the "working" interface through which the client will primarily use the object. The creation sequence only nets the client a single interface pointer that has a limited scope of functionality. If the client wishes to perform an operation outside that scope, it must call the known interface's QueryInterface function to ask for another interface on the same object.

For example, say a client has created and initialized an object but now wishes to obtain a graphical presentation, say a bitmap, from that object by calling IDataObject::GetData (see Chapter 10 for details on this function). The client must call QueryInterface to obtain an IDataObject pointer before calling the function.

It is important to note that *all operations on that object will occur through calls to the member functions of the object's various interfaces*. Any additional API functions that the client might call to affect the object itself are usually wrapper functions of common sequences of interface function calls. There simply is no other way to affect the object other than through it's interfaces.

Because a client must ask for an interface before it can possibly ask the object to perform the actions defined in the interface, the client cannot ask the object to perform an action the object does not support. This is a primary strength of the QueryInterface function as described in the early chapters of this document. Calling QueryInterface for access to an object's functionality is not problematic nor inconvenient because the client usually makes the call specifically at the point where the client wants to perform some action on the object. That is, clients generally do not call QueryInterface for all possible interfaces after the object is created so as to have all the pointers on hand—instead, the client calls QueryInterface before attempting to perform some action with the object.

In practice this means that the client must be prepared for the failure of a call to QueryInterface. Instead of being a complete pain to implementation, such preparation defines a mechanism through which the client can make dynamic choices based on the functionality of the object itself on an object-by-object basis.

For example, consider a client application that has created a number of objects and it now wants to save the application's state, which includes saving the state of each object. Let's say the client is using structured storage for its native file representation, so its first choice will be to assign an individual storage element in that file for each object. Each object can then store structured information itself and it indicates its ability to do by implementing the IPersistStorage interface. However, some object may not know how to write to a storage but know how to write to a stream and indicate the capability by implementing IPersistStream. Yet others may only know how to write information to a file themselves and thus implement IPersistFile. Finally, some objects may not know how to serialize themselves at all, but can provide a binary memory copy of the their native data through IDataObject.

In this case the client's strategy will be as follows: if an object supports IPersistStorage, then give it an IStorage instance and ask it to save its data into it by calling IPersistStorage::Save. If that object does not provide such support, check if it supports IPersistStream, and if so, create a client-controlled stream for it (in perhaps a separate client-controlled storage element) and pass that IStream pointer to the object through IPersistStream::Save. If the object does not support streams, then check for IPersistFile. If the object supports serialization to a file, then have the object write its data into a temporary file by calling IPersistFile::Save, then make a binary copy of that file in a client-controlled stream element within a client-controlled storage element. If all else fails, attempt to retrieve the object's binary data from IDataObject::GetData using the first format the object supports, and write that binary data into a client-controlled stream in a client-controlled storage.

Code for such a strategy would be structured something like the following pseudo-code for a "save object" function in the client:

```
BOOL SaveObject(IUnknown * pUnkObj)
    {
    pUnkObj->QueryInterface(IID_IPersistStorage)

    if (success)
        {
        create a storage element for the object
        call IPersistStorage::Save
        call IPersistStorage::Release
```

```
        return TRUE
        }

    //All other cases use a client-controlled stream
    create a stream element for the object in some storage

    //IPersistStorage not supported, try IPersistStream
    pUnkObj->QueryInterface(IID_IPersistStream)

    if (success)
        {
        call IPersistStream::Save
        call IPersistStream::Release
        return TRUE
        }

    //IPersistStream not supported, try IPersistFile
    pUnkObj->QueryInterface(IID_IPersistFile)

    if (success)
        {
        //Save to a temp file
        call IPersistFile::Save("objdata.tmp");
        call IPersistFile::Release
        read data from temp file
        write data to the stream
        return TRUE
        }

    //All else failed, try IDataObject
    pUnkObj->QueryInterface(IID_IDataObject)

    if (success)
        {
        call IDataObject::EnumFormatEtc
        call IEnumFORMATETC to get the first format (assume it's native)
        call IEnumFORMATETC::Release

        call IDataObject::GetData for the format, asking for global memory
        call IDataObject::Release

        Lock global memory and write to stream
        Free global memory
        return TRUE
        }

    //Everything failed, so give up
    destroy stream we created:   not using it.
    return FALSE
    }
```

In this example the client is prepared for many different types of objects and how they might provide persistent information (and using IDataObject::GetData here is stretching the concept somewhat, but shows that the client has many choices). Based on the results of QueryInterface the client decides at run-time how to save each individual object.

Reloading these objects would be a similar procedure, but the client would know, from the structure of its storage and other information it saved about the objects itself, which method to use to reload the object from the storage. The client wants to insure that it uses the same method to load the object that it did for saving it originally, that is, use the same interface instead of querying for the best one. The reason is that while the data was passively stored on disk, the object that wrote that data might have been updated such that where it once only supported IPersistStream, for example, it now supports IPersistStorage. In that case the client should ask it to load the data using IPersistStream::Load.

However, when the client goes to save the object again, it will now successfully find that the object supports IPersistStorage and can now have the object save into a storage element instead. (The container would also insure that the old client-controlled stream was deleted as it is no longer in use for that object.) This demonstrates how an object can be updated and new interfaces supported without *any* recompilation on the

part of existing clients while at the same time *suddenly working with clients on a higher level of integration than before*. In order to remain compatible the object must insure that it supports the older interfaces (such as IPersistStream) but is free to add new contracts—new interfaces such as IPersistStorage—as it wants to provide new functionality.

The point of this example, which is also true for clients that use any other interfaces an object might support in other scenarios, is that the client is empowered to make dynamic decisions on a per-object basis through the QueryInterface function. Containers programmed to be dynamic as such allow object to improve independently while insuring that the container will work as good—and generally better—as it always has with any given object. All of this is due to the powerful and important QueryInterface mechanism that for all intents and purposes is the single most important aspect of true system component software.

6 Releasing the Object

The final operation required in a COM client when dealing with an object from some other server is to free that object when the client no longer needs it. This is achieved by calling the Release member function of all interfaces obtained during the course of using the object.

Recall that a function that creates or synthesizes a new interface pointer is responsible for calling AddRef through that pointer before returning it to the caller of the function. This applies to the IClassFactory::CreateInstance function as well as CoCreateInstance (and for that matter, CoGetClassObject, too, which is why you must call IClassFactory::Release after creating the object). Therefore, as far as the client is concerned, the object will have a reference count of one after creation. The object may, in fact, have a higher reference count if it is also being used from other clients as well, but each client is only responsible and cognizant of the reference counts added on its behalf.

The other primary function that creates new interface pointers is QueryInterface. Every call the client makes to QueryInterface to obtain another interface pointer will internally generate another call to AddRef in that object, incrementing the reference count. Therefore, in addition to calling Release through the interface pointer obtained in the creation sequence, the client must also call Release through any interface pointer obtained from QueryInterface (this is illustrated in the pseudo-code of the previous section).

The bottom line is that the client is responsible for matching any operation that generates a call to AddRef through a given interface pointer with a call to Release through that same interface pointer. It is not necessary to call Release in the opposite order of calls to AddRef; it is just necessary to match the pairs. Failure to do so will cause memory leaks as objects are not freed and servers are not allowed to shut down properly. This is no different that forgetting to free memory obtained through malloc.

Finally, although the client matches its calls to AddRef and Release, the actual object may still continue to run and the server may continue to execute as well without any objects in service. The object will continue if other clients are using that same object and thus have reference counts on it. Only when all clients have released their references will that object free itself. The server will, of course, continue to execute as long as there is an object to serve, but the client does have some power over keeping a server running even without objects. That is the purpose of Server Management functions in COM.

7 Server Management

As mentioned in previous sections, a client has the ability to manage servers on the server level to keep them running even when they are not serving any objects. The client's primary mechanism for this is the IClassFactory::LockServer function described above. By calling this function with the TRUE parameter, the client places a 'lock' on the server. As long as the server either has objects created *or* has one or more locks on it, the server will continue to execute. When the server detects a zero object and zero lock condition, it can unload itself (which differs between DLL and EXE servers, as described in Chapter 7).

A client can place more than one lock on a server by calling IClassFactory::LockServer(TRUE) more than once. Each call to LockServer(TRUE) must be matched with a call to LockServer(FALSE)—the server maintains a lock count for the server as it maintains a reference count for its served objects. But while AddRef and Release affect objects, LockServer affects the server itself.

LockServer affects all servers—in-process, local, and remote—identically. The client does have some additional control over in-process objects as it normally would for other DLLs through the functions

CoLoadLibrary, CoFreeUnusedLibraries, and CoFreeAllLibraries, as described below. Normally only CoFreeUnusedLibraries is called from a client whereas the others are generally used inside the COM Library to implement other API functions. In addition, the COM Library supplies one additional function that has meaning in this context, CoIsHandlerConnected, that tells the container if an object handler is currently working in association with a local server as described in its entry below.

.1 CoFreeUnusedLibraries

void CoFreeUnusedLibraries(void)

This function and unloads any DLLs that have been loaded as a result of COM object creation calls but which are no longer in use. Client applications can call this function periodically to free up resources.

.2 CoIsHandlerConnected

BOOL CoIsHandlerConnected(pUnk)

Determines if the specified handler is connected to its corresponding object in a running local server. The result of this function might be used in a client application to determine if certain operations might result in launching a server application allowing the client to make performance decisions.

Argument	Type	Description
pUnk	IUnknown *	Specifies the object in question.
return value	BOOL	True if a handler is connected to a running server with the full object implementation, FALSE if the handler is not connected.

6. COM Servers

As described in earlier chapters, a COM Server is some module of code, a DLL or an EXE, that implements one or more object classes (each with their own CLSID). A COM server structures the object implementations such that COM clients can create an use objects from the server using the CLSID to identify the object through the processes described in Chapter 5.

In addition, COM servers themselves may be clients of other objects, usually when the server is using those other objects to help implement part of its own objects. This chapter will cover the various methods of using an object as part of another through the mechanisms of containment and aggregation.

Another feature that servers might support is the ability to emulate a different server of a different CLSID. The COM Library provides a few API functions to support this capability that are covered at the end of this chapter.

If the server is an application, that is, an executable program, then it must follow all the requirements for a COM Application as detailed in Chapter 4. If the server is a DLL, that is, an in-process server or an object handler, it must at least verify the library version and may, if desired, insure that the COM Library is initialized. That aside, all servers generally perform the following operations in order to expose their object implementations:

1. Allocate a class identifier—a CLSID—for each supported class and provide the system with a mapping between the CLSID and the server module.
2. Implement a class factory object with the IClassFactory interface for each supported CLSID.
3. Expose the class factory such that the COM Library can locate it after loading (DLL) or launching (EXE) the server.
4. Provide for unloading the server when there are no objects being served and no locks on the server (IClassFactory::LockServer).

Of course, there must be some object to serve, so the first section of this chapter discusses the basic structure of an object and some considerations for design. The sections that follow then cover the functions involved in each of these steps for the different styles of servers—DLL and EXE—which apply regardless of whether the server is running on a remote machine. Also included is a discussion of object handlers (special-case in-process objects) before the discussion of aggregation. Note that no new interfaces are introduced in this chapter as the fundamental ones, IUnknown and IClassFactory, have already been covered.

As far as the server is concerned, the COM Library exists to drive the server's class factory to create objects and to handle remote method calls from clients in other processes or on other machines and to marshal the object's return values back to the client. Whereas client applications are unaware of the object's execution context once the object is created, the server is, of course, always aware of that context. An in-process object is always loaded into the client's process space. A local or remote object always runs in a process other than the client, or on a different machine. However, the actual object itself can be written such that it does not need to care about the execution context, leaving the specifics to the structure of the server module instead. This chapter will cover one such strategy.

Finally, recall from the beginning of Chapter 5 that a client always makes a call into some in-process object whenever it calls any interface member function. If the actual object in the server is local or remote, that object is merely a proxy that generates the appropriate remote method call to the true object. This does not mean a server has to understand RPC, however, as the server always sees these calls as direct calls from a piece of code in the server process. The mechanism that achieves this, described in Chapter 7, "Communicating via Interfaces: Remoting," is that the RPC call is picked up in the server process by an "stub" object which translate the RPC information into the direct call to the server's object. From the server's point of view, the client called it directly.

1 Identifying and Registering an Object Class

A major strength of COM is the use of globally unique identifiers to essentially name each object class that exists, not only on the local machine but universally across all machines and all platforms. The algorithm that guarantees this is encompassed in the COM Library function CoCreateGuid as described in Chapter 3.

An object implementor must obtain a GUID to assign to the object server as its CLSID for each implemented class.

.1 System Registry of Classes for the Local Machine

A CLSID to identify an object implementation is not very useful unless clients have a way of finding the CLSID. From Chapter 5 we know that there are a number of ways a client may come to know a CLSID. First of all, that client may be compiled to specifically depend on a specific CLSID, in which case it obtained the server's header files with the DEFINE_GUID macros present. But for the most part, clients will want to obtain CLSIDs at run-time, especially when that client displays a list of available objects to and end-user and creates an object of the selected type at the user's request. So there must be a way to dynamically locate and load CLSIDs for accessible objects.

Furthermore, there has to be some system-wide method for the COM Library to associate a given CLSID, regardless of how the client obtained it, to the server code that implements that class. In other words, the COM Library requires some persistent store of CLSID-to-server mappings that it uses to implement its locator services. It is up to the COM Library implementor, not the implementor of clients or servers, to define the store and how server applications would register their CLSIDs and server module names in that store.

The store must distinguish between in-process, local, and remote objects as well as object handlers in addition to any environment-specific differences. The COM implementation on Microsoft Windows uses the Windows system registry (also called the registration database, or RegDB for short) as a store for such information. In that registry there is a root key called "CLSID" (spelled out in those letters) under which servers are responsible to create entries that point to their modules. Usually these entries are created at installation time by the application's setup code, but can be done at run-time if desired.

When a server is installed under Windows, the installation program will create a subkey under "CLSID" for each class the server supports, using the standard string representation of the CLSID as the key name (including the curly braces).[52] So the first key that the TextRender object would create appears as follows (CLSID is the root key the indentation of the object class implies a sub-key relationship with the one above it):

```
CLSID
    {12345678-ABCD-1234-5678-9ABCDEF00000} = TextRender Example
```

Depending on the type of same-machine server that handles this CLSID there will be one or more subkeys created underneath the ASCII CLSID string:

Server Flavor	Subkey Name	Value
In-Process	InprocServer32	Pathname of the server DLL
Local	LocalServer32	Pathname of the server EXE
Object Handler	InprocHandler32	Pathname to the object handler DLL.

So, for example, if the TextRender object was implemented in a TEXTREND.DLL, its entries would appear as:

```
CLSID
    {12345678-ABCD-1234-5678-9ABCDEF00000} = TextRender Example
        InprocServer32 = c:\objects\textrend.dll
```

If it were implemented in an application, TEXTREND.EXE, and worked with an object handler in TEXTHAND.DLL, the entries would appear as:

```
CLSID
    {12345678-ABCD-1234-5678-9ABCDEF00000} = TextRender Example
        InprocHandler32 = c:\handlers\texthand.dll
        LocalServer32 = c:\objects\textrend.exe
```

Over time, the registry will become populated with many CLSIDs and many such entries.

[52] Under Microsoft Windows, this key is created using the standard Windows API for registry manipulation. Other COM implementations may include their own functions as necessary, as long as it's consistent on a given platform. Such functions are not part of this specification

.2 Remote Objects: AtBits Key

As described in the last section, a prerequisite to server implementation is generating a CLSID for that server. This CLSID is registered in the system registry and referenced in the server code. The full path name of the server DLL or EXE is registered in association with the CLSID.

The remote server can actually run either on the machine where the server code is stored or on the same machine as its connected client (assuming the class is registered on the remote machine and there is a compatible binary image available). Servers that use the default security provided with the system must run where its client is running. To indicate the mode of operation, the Microsoft Windows implementation of COM includes the subkey "AtBits" that is registered along with the server's CLSID. To register a server to run where the persistent state of the object is stored, set AtBits to "Y." To register the server to run where the client is running, either set it to "N" or leave the attribute out altogether. The default is to run the server where the client is running. The registration example below shows how the TextRender object would allow itself to be activated remotely.

```
CLSID
    {12345678-ABCD-1234-5678-9ABCDEF00000} = TextRender Example
        LocalServer = c:\objects\textrend.exe
        AtBits = Y
```

.3 Self-Registering Servers

COM servers which are installed as part of an application setup program are usually registered by the setup program. However, to facilitate the registration of smaller grained servers, the notion of a self-registering server is introduced.

.1 Self-Registering DLL's

In-process COM servers (DLL's on the Windows and Macintosh platforms) support self-registration through several DLL entry points with well-known names. The DLL entry points for registering and unregistering a server are defined as follows:

HRESULT DllRegisterServer(void);

HRESULT DllUnregisterServer(void);

Both of these entry points are required for a DLL to be self-registering. The implementation of the DllRegisterServer entry point adds or updates registry information for all the classes implemented by the DLL. The DllUnregisterServer entry point removes its information from the registry.

.2 Self-Registering EXE's

There isn't an easy way for EXE's to publish entry points with well-known names, so a direct translation of DllRegisterServer isn't possible. Instead, EXE's support self-registration using special command line flags. EXE's that support self-registration must mark their resource fork in the same way as DLL's, so that the EXE's support for the command line flags is detectable. Launching an EXE marked as self-registering with the /REGSERVER command line argument should cause it to do whatever OLE installation is necessary and then exit. The /UNREGSERVER argument is the equivalent to DllUnregisterServer. The /REGSERVER and /UNREGSERVER strings should be treated case-insensitively, and that the character '-' can be substituted for '/'.

Other than guaranteeing that it has the correct entry point or implements the correct command line argument, an application that indicates it is self-registering must build its registration logic so that it may be called any number of times on a given system even if it is already installed. Telling it to register itself more than once should not have any negative side effects. The same is true for unregistering.

On normal startup (without the /REGSERVER command line option) EXE's should call the registration code to make sure their registry information is current. EXE's will indicate the failure or success of the self-registration process through their return code by returning zero for success and non-zero for failure.

.3 Identifying Self-Registering Servers

Applications need to check to see if a given server module is self-registering without actually loading the DLL or EXE for performance reasons and to avoid possible negative side-affects of code within the module being executed without the module first being registered. To accomplish this, the DLL or EXE must be tagged with a version resource that can be read without actually causing any code in the module to be executed. On Windows platforms, this involves using the version resource to hold a self-registration keyword. Since the VERSIONINFO section is fixed and cannot be easily extended, the following string is added to the "StringFileInfo", with an empty key value:

```
VALUE "OLESelfRegister", ""
```

For example:

```
VS_VERSION_INFO      VERSIONINFO
  FILEVERSION        1,0,0,1
  PRODUCTVERSION     1,0,0,1
  FILEFLAGSMASK      VS_FFI_FILEFLAGSMASK
#ifdef _DEBUG
  FILEFLAGS          VS_FF_DEBUG|VS_FF_PRIVATEBUILD|VS_FF_PRERELEASE
#else
  FILEFLAGS          0 // final version
#endif
  FILEOS             VOS_DOS_WINDOWS16
  FILETYPE           VFT_APP
  FILESUBTYPE        0    // not used
BEGIN
  BLOCK "StringFileInfo"
  BEGIN
    BLOCK "040904E4" // Lang=US English, CharSet=Windows Multilingual
    BEGIN
    VALUE "CompanyName",       "\0"
    VALUE "FileDescription", "BUTTON OLE Control DLL\0"
    VALUE "FileVersion",       "1.0.001\0"
    VALUE "InternalName",      "BUTTON\0"
    VALUE "LegalCopyright",   "\0"
    VALUE "LegalTrademarks", "\0"
    VALUE "OriginalFilename","BUTTON.DLL\0"
    VALUE "ProductName",       "BUTTON\0"
    VALUE "ProductVersion",   "1.0.001\0"
    VALUE "OLESelfRegister", "" // New keyword
    END
  END
  BLOCK "VarFileInfo"
  BEGIN
    VALUE "Translation", 0x409, 1252
  END
END
```

To support self-registering servers, an application can add a "Browse" button to its object selection user interface, which pops up a standard File Open dialog. After the user chooses a DLL or EXE the application can check to see if it is marked for self-registration and, if so, call its DllRegisterServer entry point (or execute the EXE with the /REGSERVER command line switch). The DLL or EXE should register itself at this point.

2 Implementing the Class Factory

The existence of a CLSID available to clients implies that there is a class factory that is capable of manufacturing objects of that class. The server, DLL or EXE, associated with the class in the registry is responsible to provide that class factory and expose it to the COM Library to make COM's creation mechanisms work for client. The specific mechanisms to expose the class factory is covered shortly, but first, let's examine how a class factory may be implemented[53].

[53] Note that the example code given below illustrates one of many ways a class factory object can be implemented.

.1 Defining the Class Factory Object

First of all, you need to define an object that implements the IClassFactory interface (or other factory-type interface if applicable). As you would define any other object, you can define a class factory. The following is an example class factory for our TextRender objects in C++:

```
class CTextRenderFactory : public IClassFactory
    {
    protected:
        ULONG               m_cRef;

    public:
        CTextRenderFactory(void);
        ~CTextRenderFactory(void);

        //IUnknown members
        HRESULT QueryInterface(REFIID, pLPVOID);
        ULONG AddRef(void);
        ULONG Release(void);

        //IClassFactory members
        HRESULT CreateInstance(IUnknown *, REFIID iid, void **ppv
        HRESULT LockServer(BOOL);
    };
```

Implementing the member functions of this object are fairly straightforward. AddRef and *Release* do their usual business, with Release calling *delete this* when the count is decremented to zero. Note that the zero-count event in *Release* has no effect other than to destroy the object—it does not cause the server to unload as that is the prerogative of LockServer. In any case, the QueryInterface implementation here will return pointers for IUnknown and IClassFactory.

.1 IClassFactory::CreateInstance

The class factory-specific functions are really all that are interesting. CreateInstance in this example will create an instance of the CTextRender object and return an interface pointer to it as shown below. Note that if pUnkOuter is non-NULL, that is, another object is attempting to aggregate, this code will fail with CLASS_E_NOAGGREGATION (this limitation will be revisited when later when aggregation is discussed).

```
//A global variable that counts objects being served
ULONG      g_cObj=0;

HRESULT CTextRenderFactory::CreateInstance(IUnknown * pUnkOuter, REFIID iid, void ** ppv) {
    CTextRender *          pObj;
    HRESULT                hr;

    *ppv=NULL;
    hr=E_OUTOFMEMORY;
    if (NULL!=pUnkOuter)
        return CLASS_E_NOAGGREGATION;

    //Create the object passing function to notify on destruction.
    pObj=new CTextRender(pUnkOuter, ObjectDestroyed);
    if (NULL==pObj)
        return hr;

    [Usually some other object initialization done here]

    //Obtain the first interface pointer (which does an AddRef)
    hr=pObj->QueryInterface(iid, ppv);

    //Kill the object if initial creation or FInit failed.
    if (FAILED(hr))
        delete pObj;
    else
        g_cObj++;

    return hr;
    }
```

There are two interesting points to this code, which is fairly standard for server implementations. First of all, note the call to the object's QueryInterface after creation. This accomplishes two things: first, since objects are generally constructed with a reference count of zero (common practice) then this QueryInterface call, if successful, has the effect of calling AddRef as well, making the object have a reference count of one. Second, it lets the object determine if it supports the interface requested in iid and if it does, it fills in ppv for us.

The second key point is that COM defines no standard mechanism for counting instantiated objects (there is no need for such a generic service), so this implementation example maintains a count of the objects in service using the global variable g_cObj. This count generally needs to be global so that other global functions can access it (see "Providing for Server Unloading" below). When CreateInstance successfully creates a new object it increments this count. When an object (not the class factory but the one the class factory creates) destroys itself in it's implementation of CTextRender::Release, it should decrement this count to match the increment in CreateInstance.

It is not necessary, however, for the object to have direct access to this variable, and there are techniques to avoid such access.. The example above passes a pointer to a function called ObjectDestroyed to the CTextRender constructor such that when the object destroys itself in it's Release it will call ObjectDestroyed to affect the server's object count:

```
void ObjectDestroyed(void) {
    g_cObj--;
    [Initiate unloading if g_cObj is zero and there are no locks]
    return;
    }

CTextRender::CTextRender(void (* pfnDestroy)(void)) {
    m_cRef=0;
    m_pfnDestroy=pfnDestroy;
    [Other initialization]
    return;
    }

ULONG CTextRender::Release(void) {
    ULONG       cRefT;
    cRefT=--m_cRef;
    if (0L==m_cRef) {
        if (NULL!=m_pfnDestroy)
            (*m_pfnDestroy)();
        delete this;
        }
    return cRefT;
    }
```

The object might also be given a pointer to the class factory object itself (which the object will call AddRef through, of course) that accomplishes the same thing. Regardless of the design, the point is that the object can be designed so as to be unaware of the exact object counting mechanism, having instead some mechanism to notify the server as a whole about the destroy event. A standard mechanism for this is not part of COM.

You might have noticed that the ObjectDestroyed function above contained a note that if there are no objects and no locks on the server, then the server can initiate unloading. What really happens here depends on the type of server, DLL or EXE, and will be covered under "Providing for Server Unloading."

.2 IClassFactory::LockServer

The other interesting member function of a class factory is LockServer. Here the server increments of decrements a lock count depending on the fLock parameter. If the last lock is removed and there are no objects in server, the server initiates unloading which again, is specific to the type of server and a topic for a later section. In any case, COM does not define a standard method for tracking the lock count. Since other code outside of the class factory may need access to the lock count, a global variable works well:

```
//Global server lock count.
ULONG       g_cLock=0;
```

The implementation of LockServer is correspondingly simple:

```
HRESULT CTextRenderFactory::LockServer(BOOL fLock)
    {
```



```
if (fLock)
      g_cLock++;
else
      {
      g_cLock--;
      [Initiate unloading if there are no objects and no locks]
      }

return NOERROR;
}
```

It is perfectly reasonable to double the use of g_cObj for counting locks as well as objects. You might want to keep them separate for debugging purposes.

3 Exposing the Class Factory

With a class factory implementation the server must now expose it such that the COM Library can locate the class factory from within CoGetClassObject after it has loaded the DLL[54] server or launched the EXE server. The exact method of exposing the class factory differs for each server type. The following sections cover each type in detail which apply to DLLs and EXEs running on the local or remote machine in relation to the client. There are also some considerations for DLL servers running remotely under a surrogate server that are covered in this section.

.1 Exposing the Class Factory from DLL Servers

To expose its class factory, an in-process server only needs to export[55] a function explicitly named DllGetClassObject. The COM Library will attempt to locate this function in the DLL's exports[56] and call it from within CoGetClassObject when the client has specified CLSCTX_INPROC_SERVER. Note that a DLL server can in addition expose a class factory at a later time using the function CoRegisterClassObject discussed for EXE servers below. This would only be used after the DLL was already loaded for some other reason.

.1 DllGetClassObject

HRESULT DllGetClassObject(clsid, iid, ppv)

This is not a function in the COM Library itself; rather, it is a function that is exported from DLL servers.

In the case that a call to the COM API function CoGetClassObject results in the class object having to be loaded from a DLL, CoGetClassObject uses the DllGetClassObject that must be exported from the DLL in order to actually retrieve the class.

Argument	Type	Description
clsid	REFCLSID	The class of the class factory being requested.
iid	REFIID	The interface with which the caller wants to talk to the class factory. Most often this is IID_IClassFactory but is not restricted to it.
ppv	void **	The place in which to put the interface pointer.

Return Value	Meaning
S_OK	Success.
E_NOINTERFACE	The requested interface was not supported on the class object.
E_OUTOFMEMORY	Memory could not be allocated to service the request.
E_UNEXPECTED	An unknown error occurred.

Note that since DllGetClassObject is passed the CLSID, a single implementation of this function can handle any number of classes. That also means that a single in-process server can implement any number of clas-

[54] Again, the term DLL is used generically to describe any shared library as supported by a given COM platform.

[55] Under Microsoft Windows this means listing the function in the EXPORTS section of a module definitions file or using the __declspec(dllexport) keyword at compile-time. Other platforms may differ as to requirements here, but in any case the function must be visible to other modules within the same process, but not across processes.

[56] Under Windows, CoGetClassObject, after loading the DLL with CoLoadLibrary, call the Windows API GetProcAddress("DllGetClassObject") to obtain the pointer to the actual function in the DLL.

ses. The implementation of DllGetClassObject only need create the proper class factory for the requested CLSID.

Most implementation of this function for a single class look very much like the implementation of IClassFactory::CreateInstance as illustrated in the code below:

```
HRESULT DllGetClassObject(REFCLSID clsid, REFIID iid, void **ppv) {
    CTextRenderFactory * pCF;
    HRESULT              hr=E_OUTOFMEMORY;

    if (!CLSID_TextRender!=clsid)
        return E_FAIL;
    pCF=new CTextRenderFactory();
    if (NULL==pCF)
        return E_OUTOFMEMORY;

    //This validates the requested interface and calls AddRef
    hr=pCF->QueryInterface(iid, ppv);
    if (FAILED(hr))
        delete pCF;
    else
        ppv=pCF;
    return hr;
  }
```

As is conventional with object implementations, including class factories, construction of the object sets the reference count to zero such that the initial QueryInterface creates the first actual reference count. Upon successful return from this function, the class factory will have a reference count of one which must be released by the caller (COM or the client, whoever gets the interface pointer).

The structure of a DLL server with its object and class factory is illustrated in Figure 6-1 below. This figure also illustrates the sequence of calls and events that happen when the client executes the standard object creation sequence of CoGetClassObject and IClassFactory::CreateInstance.

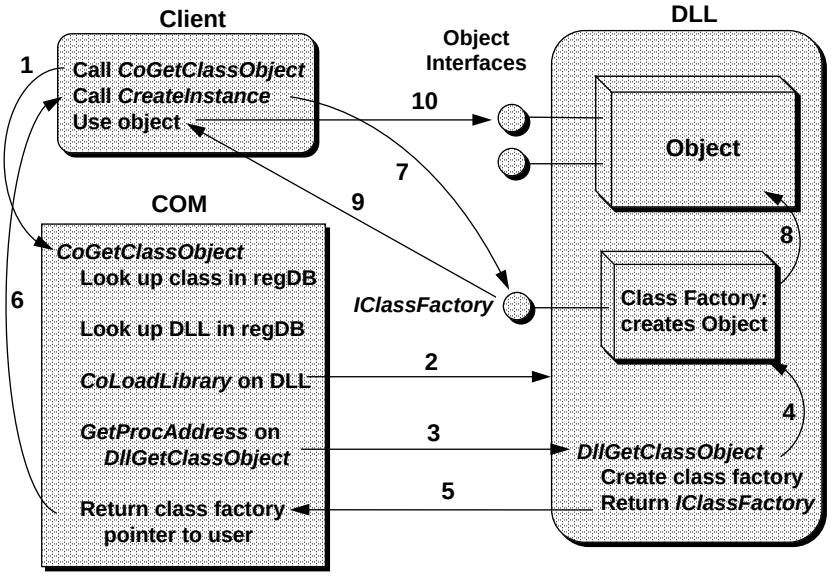


Figure 6-1: Creation sequence of an object from a DLL server. Function calls not in COM are from the Windows API.

.2 Exposing the Class Factory from EXE Servers

To expose a class factory from a server application is a different matter than for a DLL server for the reason that the application executes in a different process from the client. Thus, the COM Library cannot just obtain a pointer to an exported function and call that function to retrieve the class factory.

When COM launches an application from within CoGetClassObject it must wait for that application to register a class factory for the desired CLSID through the function CoRegisterClassObject.[57] Once that class factory appears to COM it can return an interface pointer (actually a pointer to the proxy) to the client. CoGetClassObject may time out if the server application takes too long.

The server can differentiate between times it is launched stand-alone and when it is launched from within COM. When COM launches the application it includes a switch "/Embedding"[58] on the server's command line. If the flag is present, the server must register its class factory with CoRegisterClassObject. If the flag is absent, the server may or may not choose to register depending on the object class.

Note that a server application can support any number of object classes by calling CoRegisterClassObject on startup. In fact, a server must register *all* supported class factories because the application is not told which CLSID was requested in the client.

Where CoRegisterClassObject registers a servers factories with COM on startup, the function CoRevokeClassObject unregisters those same factories on application shutdown so they are no longer available, meaning COM must launch the server again for those class factories. Each call to CoRegisterClassObject must be matched with a call to CoRevokeClassObject.

.1 CoRegisterClassObject

HRESULT CoRegisterClassObject(clsid, pUnk, grfContext, grfFlags, pdwRegister)

Registers the specified server class factory identified with *pUnk* with COM in order that it may be connected to by COM Clients. When a server application starts, it creates each class factory it supports and passes them to this function. When a server application exits, it revokes all its registered class objects with CoRevokeClassObject.

Note that an in-process object could call this function to expose a class factory only when the DLL is already loaded in another process and did not want to expose a class factory until it was loaded for some other reason.

The grfContext flag identifies the execution context of the server and is usually CLSCTX_LOCAL_SERVER. The grfFlags is used to control how connections are made to the class object. Values for this parameter are the following:

```
typedef enum tagREGCLS
    {
    REGCLS_SINGLEUSE = 0,
    REGCLS_MULTIPLEUSE = 1,
    REGCLS_MULTI_SEPARATE = 2
    } REGCLS;
```

Value	Description
REGCLS_SINGLEUSE	Once one client has connected to the class object with CoGetClassObject, then the class object should be removed from public view so that no other clients can similarly connect to it; new clients will use a new instance of the class factory, running a new copy of the server application if necessary. Specifying this flag does not affect the responsibility of the server to call CoRevokeClassObject on shutdown.
REGCLS_MULTIPLEUSE	Many CoGetClassObject calls can connect to the same class factory. When a class factory is registered from a local server (grfContext is CLSCTX_LOCAL_SERVER) and grfFlags includes REGCLS_MULTIPLEUSE, then it is the case that the same class factory will be *automatically also registered* as the in-process server (CLSCTX_IN-PROC_SERVER) for its own process
REGCLS_MULTI_SEPARATE	The same as REGCLS_MULTIPLEUSE, except that registration as a local server does *not* automatically also register as an in-process server in that same process (or any other, for that matter).

Thus, registering as

CLSCTX_LOCAL_SERVER, REGCLS_MULTIPLEUSE

is the equivalent to registering as

[57] This function is called in the COM Library loaded in the server's process space so it actually establishes the remote proxy and stub necessary to perform remote procedure calls.

[58] Case-insensitive. This name originated in OLE 1.0 and has been maintained for such historical reasons and compatibility.

(CLSCTX_INPROC_SERVER | CLSCTX_LOCAL_SERVER), REGCLS_MULTI_SEPARATE

but is different than registering as

CLSCTX_LOCAL_SERVER, REGCLS_MULTI_SEPARATE.

By using REGCLS_MULTI_SEPARATE, an object implementation can cause different class factories to be used according to whether or not it is being created from within the same process as it is implemented.

The following table summarizes the allowable flag combinations and the registrations that are effected by the various combinations:

	REGCLS_-SINGLEUSE	REGCLS_-MULTIPLEUSE	REGCLS_-MULTI_SEPARATE	Other
CLSCTX_IN-PROC_SERVER	*error*	In-Process	In-Process	*error*
CLSCTX_LO-CAL_SERVER	Local	In-Process/Local	**Just Local**	*error*
Both of the above	*error*	In-Process/Local	In-Process/Local	*error*
Other	*error*	*error*	*error*	*error*

The key difference is in the middle columns and the middle rows. In the REGCLS_MULTIPLEUSE column, they are the same (registers multiple use for both InProc and local); in the REGCLS_MULTI_SEPARATE column, the local server case is local only.

The arguments to this function are as follows:

Argument	Type	Description
rclsid	REFCLSID	The CLSID of the class factory being registered.
pUnk	IUnknown *	The class factory whose availability is being published.
grfContext	DWORD	As in CoGetClassObject.
grfFlags	DWORD	REGCLS values that control the use of the class factory.
pdwRegister	DWORD *	A place at which a token is passed back with which this registration can be revoked in CoRevokeClassObject.

Return Value	Meaning
S_OK	Success.
CO_E_OBJISREG	Error. The indicated class is already registered.
E_OUTOFMEMORY	Memory could not be allocated to service the request.
E_UNEXPECTED	An unknown error occurred.

.2 CoRevokeClassObject

HRESULT CoRevokeClassObject(dwRegister)

Informs the COM Library that a class factory previously registered with CoRegisterClassObject is no longer available for use. Server applications call this function on shutdown after having detected the necessary unloading conditions.

- There are no instances of the class in existence, that is, the object count is zero.
- The class factory has a zero number of locks from IClassFactory::LockServer.
- The application servicing the class object is not showing itself to the user (that is, not under user control)

When, subsequently, the reference count on the class object reaches zero, the class object can be destroyed, allowing the application to exit.

Argument	Type	Description
dwRegister	DWORD	A token previously returned from CoRegisterclassObject.

Return Value	Meaning
S_OK	Success.
E_UNEXPECTED	An unknown error occurred.

The structure of a server application with its object and class factory is illustrated in Figure 6-2. This figure also illustrates the sequence of calls and events that happen when the client executes the standard object creation sequence of CoGetClassObject and IClassFactory::CreateInstance.

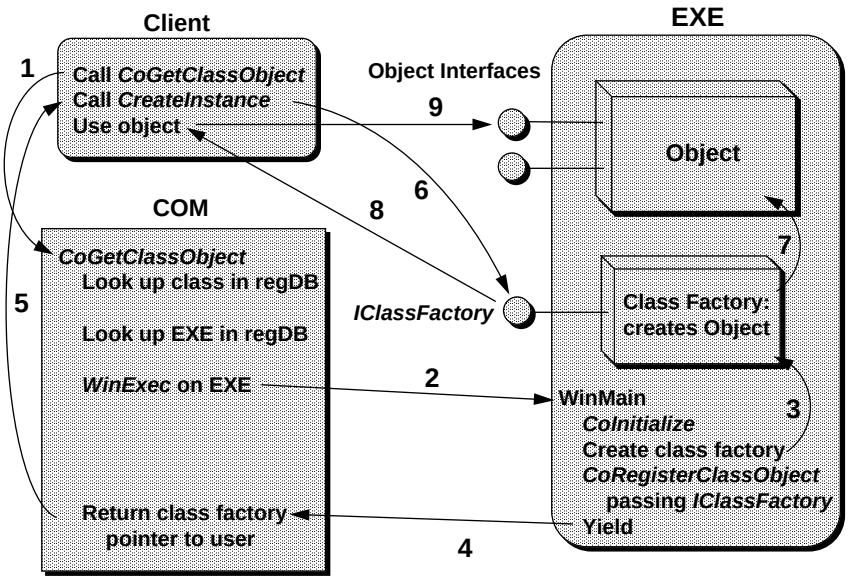


Figure 6-2: Creation sequence of an object from a server application. Function calls not in COM are from the Windows API.

Compare this figure with DLL server Figure 6-1 in the previous section. You'll notice that the structure of the server is generally the same, that is, both have their object and class factory. You'll also notice that the creation sequence from the client's point of view is identical. Again, once the client determines the CLSID of the desired object that client leaves the specifics up to CoGetClassObject. The only differences between the two figures occur inside the COM Library and the specific means of exposing the class factory from the server (along with the unloading mechanism).

Finally, CoRegisterClassObject and CoRevokeClassObject along with when a server calls them demonstrate why a reference count on the class factory is insufficient to keep a server in memory and why IClassFactory::LockServer exists. CoRegisterClassObject must, in order to be implemented properly, hold on to the IUnknown pointer passed to it (that is, the class factory). The reference counting rules state that CoRegisterClassObject must call AddRef on that pointer accordingly. This reference count can only be removed inside CoRevokeClassObject.

However, CoRevokeClassObject is only called on application shutdown and not at any other time. How does the server know when to start its shutdown sequence? Since it has to *be* in the process of shutting down to have the final reference counts on the class factory released through CoRevokeClassObject, it cannot use the reference count to determine when to start the shutdown process in the first place. Therefore there has to be another mechanism through which shutdown is controlled which is IClassFactory::LockServer.

4 Providing for Server Unloading

When a server has no objects to serve, has no locks, and is not being controlled by an end user (which applies generally to server applications with user interface), then the server has no reason to stay loaded in memory and should provide for unloading itself. This unloading provision differs between server types (DLL and EXE, but no difference for remote servers) as much as class factory registration because whereas a server application can simply terminate itself, an in-process DLL must wait for someone else to explicitly unload it. Therefore the mechanisms for unloading are different and are covered separately in the following sections.

.1 Unloading In-Process Servers

As mentioned above, a DLL must wait for someone else to explicitly unload it. The server must, however, have a mechanism through which it indicates whether or not it should be unloaded. That mechanism is a function with the name DllCanUnloadNow that is exported in the same manner as DllGetClassObject.

.1 DllCanUloadNow

HRESULT DllCanUnloadNow(void)

DllCanUnloadNow is not provided by COM. Rather, it is a function implemented by and exported from DLLs supporting the Component Object Model. DllCanUnloadNow should be exported from DLLs designed to be dynamically loaded in CoGetClassObject or CoLoadLibrary calls. A DLL is no longer in use when there are no existing instances of classes it manages; at this point, the DLL can be safely freed by calling CoFreeUnusedLibraries. If the DLL loaded by CoGetClassObject fails to export DllCanUnloadNow, the DLL will only be unloaded when CoUninitialize is called to release the COM libraries.

If this function returns S_OK, the duration within which it is in fact safe to unload the DLL depends on whether the DLL is single or multi-thread aware. For single thread DLLs, it is safe to unload the DLL up until such time as the thread on which DllCanUnloadNow was invoked causes it to be otherwise (objects created, for example).

Return Value	Meaning
S_OK	The DLL may be unloaded now.
S_FALSE	The DLL should not be unloaded at the present time.

.2 Unloading EXE Servers

A server application is responsible for unloading itself, simply by terminating and exiting its main entry function[59], when the shutdown conditions are met, including whether or not the user has control. In the ongoing example of this chapter, this would involve detecting the proper shutdown conditions whenever an object is destroyed (in the suggested ObjectDestroyed function) or whenever the last lock is removed (in IClassFactory::LockServer).

```
//User control flag
BOOL        g_fUser=FALSE;

void ObjectDestroyed(void) {
    g_cObj--;
    if (0L==g_cObj && 0L==g_cLock && !g_fUser)
        //Begin shutdown
    return;
    }

HRESULT CTextRenderFactory::LockServer(BOOL fLock) {
    if (fLock)
        g_cLock++;          // for single threaded app only, of course
    else    {
        g_cLock--;
        if (0L==g_cObj && 0L==g_cLock && !g_fUser)
            //Begin shutdown
        }
    return NOERROR;
    }
```

If desired, you can of course centralize the shutdown conditions by artificially incrementing the object count in IClassFactory::LockServer and directly calling ObjectDestroyed. That way you do not need redundant code in both functions.

During shutdown, the server is responsible for calling CoRevokeClassObject on all previously registered class factories and for calling CoUninitialize like any COM application.

A server application only needs a "user-control" flag if it becomes visible in some way and also allows the user to perform some action which would necessitate the application stays running regardless of any other

[59] Under Microsoft Windows, the application usually starts shutdown by posting a WM_CLOSE message to its main window, simulating what happens when a user closes an application. This eventually causes the application to exit the WinMain function.

conditions. For example, the server might be running to service an object for a client and the user opens another file in that same application. Since the user is the only agent who can close the file, the user control flag is set to TRUE meaning that the user must explicitly close the application: no automatic shutdown is possible.

If a server is visible and under user control, there is the possibility that clients have connections to objects within that server when the user explicitly closes the application. In that situation the server can take one of two actions:

1. Simply hide the application and reset the user control flag to FALSE such that the server will automatically shut down when all objects and locks are released.
2. Terminate the application but call CoDisconnectObject for each object in service to forcibly disconnect all clients.

The second option, though more brutal, is necessary in some situations. The CoDisconnectObject function exists to insure that all external reference counts to the server's objects are released such that the server can release its own references and destroy all objects.

.1 CoDisconnectObject

HRESULT CoDisconnectObject(pUnk, dwReserved)

This function serves any extant remote connections that are being maintained on behalf of all the interface pointers on this object. This is a very rude and privileged operation which should generally only be invoked by the process in which the object actually is managed by the object implementation itself.

The primary purpose of this operation is to give an application process certain and definite control over connections to other processes that may have been made from objects managed by the process. If the application process wishes to exit, then we do not want it to be the case that the extant reference counts from clients of the application's objects in fact keeps the process alive. The process can call this function for each of the objects that it manages without waiting for any confirmation from clients. Having thus released resources maintained by the remoting connections, the application process can exit safely and cleanly. In effect, CoDisconnectObject causes a controlled crash of the remoting connections to the object.

Argument	Type	Description
pUnk	IUnknown *	The object that we wish to disconnect. May be any interface on the object which is polymorphic with IUnknown, not necessarily the exact interface returned by QueryInterface(IID_IUnknown...).
dwReserved	DWORD	Reserved for future use; must be zero.

Return Value	Meaning
S_OK	Success.
E_UNEXPECTED	An unspecified error occurred.

5 Object Handlers

As mentioned earlier this specification, object handlers from one perspective are special cases of in-process servers that talk to their local or remote servers as well as a client. From a second perspective, an object handler is really just a fancy proxy for a local or remote server that does a little more than just forward calls through RPC. The latter view is more precise architecturally: a "handler" is simply the piece of code that runs in the client's space on behalf of a remote object; it can be used synonymously with the term "proxy object." The handler may be a trivial one, one that simply forwards all of its calls on to the remote object, or it may implement some amount of non-trivial client side processing. (In practice, the term "proxy object" is most often reserved for use with trivial handlers, leaving "handler" for the more general situation.)

The structure of an object handler is exactly the same as a full-in process server: an object handler implements an object, a class factory, and the two functions DllGetClassObject and DllCanUnloadNow exactly as described above.

The key difference between handlers and full DLL servers (and simple proxy objects, for that matter) is the extent to which they implement their respective objects. Whereas the full DLL server implements the complete object (using other objects internally, if desired), the handler only implements a partial object de-

pending on a local or remote server to complete the implementation. Again, the reasons for this is that sometimes a certain interface can only be useful when implemented on an in-process object, such as when member functions of that interface contain parameters that cannot be shared between processes. Thus the object in the handler would implement the restricted in-process interface but leave all others for implementation in the local or remote server.

6 Object Reusability

With object-oriented programming it is often true that there already exists some object that implements some of what you want to implement, and instead of rewriting all that code yourself you would like to reuse that other object for your own implementation. Hence we have the desire for object reusability and a number means to achieve it such as implementation inheritance, which is exploited in C++ and other languages. However, as discussed in the "Object Reusability" section of Chapter 2, implementation inheritance has some significant drawbacks and problems that do not make it a good object reusability mechanism for a system object model.

For that reason COM supports two notions of object reuse, containment and aggregation, that were also described in Chapter 2. In that chapter we saw that containment, the most common and simplest for of object reuse, is where the "outer object" simply uses other "inner objects" for their services. The outer object is nothing more than a client of the inner objects. We also saw in Chapter 2 the notion of aggregation, where the outer object exposes interfaces from inner objects as if the outer object implemented those interfaces itself. We brought up the catch that there has to be some mechanism through which the IUnknown behavior of inner object interfaces exposed in this manner is appropriate to the outer object. We are now in a position to see exactly how the solution manifests itself.

The following sections treat Containment and Aggregation in more detail using the TextRender object as an example. To refresh our memory of this object's purpose, the following list reiterates the specific features of the TextRender object that implements the IPersistFile and IDataObject interfaces:

- Read text from a file through IPersistFile::Load
- Write text to a file through IPersistFile::Save
- Accept a memory copy of the text through IDataObject::SetData
- Render a memory copy of the text through IDataObject::GetData
- Render metafile and bitmap images of the text also through IDataObject::GetData

.1 Reusability Through Containment

Let's say that when we decide to implement the TextRender object we find that another object exists with CLSID_TextImage that is capable of accepting text through IDataObject::SetData but can do nothing more than render a metafile or bitmap for that text through IDataObject::GetData. This "TextImage" object cannot render memory copies of the text and has no concept of reading or writing text to a file. But it does such a good job implementing the graphical rendering that we wish to use it to help implement our TextRender object.

In this case the TextRender object, when asked for a metafile or bitmap of its current text in IDataObject::GetData, would delegate the rendering to the TextImage object. TextRender would first call TextImage's IDataObject::SetData to give it the most recent text (if it has changed since the last call) and then call TextImage's IDataObject::GetData asking for the metafile or bitmap format. This delegation is illustrated in Figure 6-3.

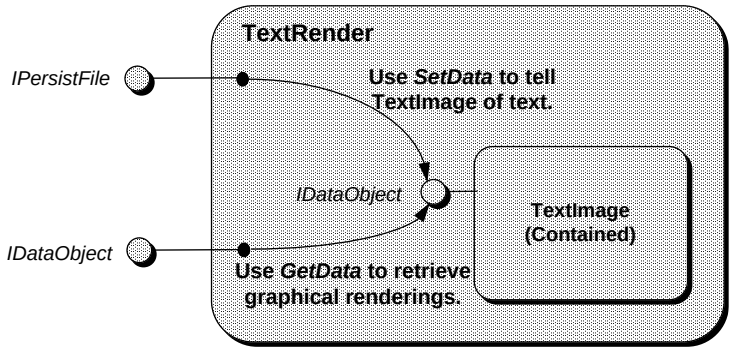


Figure 6-3: An outer object that uses inner objects through containment is a client of the inner objects.

To create this configuration, the TextRender object would, during its own creation, instantiate the TextImage object with the following code, storing the TextImage's IDataObject pointer in a TextImage field m_pIDataObjImage:

```
//TextRender initialization
HRESULT      hr;
hr=CoCreateInstance(CLSID_TextImage, CLSCTX_SERVER, NULL, IID_IDataObject, (void *)&m_pIDataObjImage);
if (FAILED(hr))
    //TextImage not available, either fail or disable graphic rendering
//Success:   can now make use of TextImage object.
```

This code is included here to show the NULL parameter in the middle of the call to CoCreateInstance. This is the "outer unknown" and is only applicable to aggregation. Containment does not make use of the outer unknown concept and so this parameter should always be NULL.

Now that the TextRender object has TextImage's IDataObject it can delegate functionality to TextImage as needed. The following pseudo-code illustrates how TextRender's IDataObject::GetData function might be implemented:

```
HRESULT CTextRender::GetData(FORMATETC *pFE, STGMEDIUM *pSTM)
    {
    switch ([format in FORMATETC])
        {
        case <text>:
            //Make copy of text and return
        case <metafile>:
        case <bitmap>:
            //Insure TextImage has current text
            m_pIDataObjImage->SetData(<copy of our current text>);
            return m_pIDataObjImage->GetData(pFE, pSTM);
        }
    return <error>;
    }
```

Note that if the TextImage object was modified at some later date to implement additional interfaces (such as IPersistFile) or was updated to also support rendering copies of text in memory just like TextRender, **the code above would still function perfectly.** This is the *key* power of COM's reusability mechanisms over traditional language-style implementation inheritance: the reused object can freely revise itself so long as it continues to provide the exact behavior it has provided in the past. Since the TextRender object never bothers to query for any other interface on TextImage, and because it never call's TextImage's GetData for any format other than metafile or bitmap, TextImage can implement any number of new interfaces and support any number of new formats in GetData. All TextImage has to insure is that the behavior of SetData for text and the behavior of GetData for metafiles and bitmaps remains the same.

Of course, this is just a simple example of containment. Real components will generally be much more complex and will generally make use of many inner objects and many more interfaces in this manner. But again, since the outer object only depends on the *behavior* of the inner object and does not care how it goes about performing its operations, the inner object can be modified without requiring any recompilation or any other changes to the outer object. That is reusability at its finest.

.2 Reusability Through Aggregation

Let's now say that we are planning to revise our TextRender object at a later time than out initial containment implementation in the previous section. At that time we find that the implementor of the TextImage object at the time the implementor of the TextRender object sat down to work (or perhaps is making a revision of his object) that the vendor of the TextImage object has improved TextImage such that it implements everything that TextRender would like to do through its IDataObject interface. That is, TextImage still accepts text through SetData but has recently added the ability to make copies of its text and provide those copies through GetData in addition to metafiles and bitmaps.

In this case, the implementor of TextRender now sees that TextImage's implementation of IDataObject is exactly the implementation that TextRender requires. What we, as the implementors of TextRender, would like to do now is simply expose TextImage's IDataObject as our own as shown in Figure 6-4.

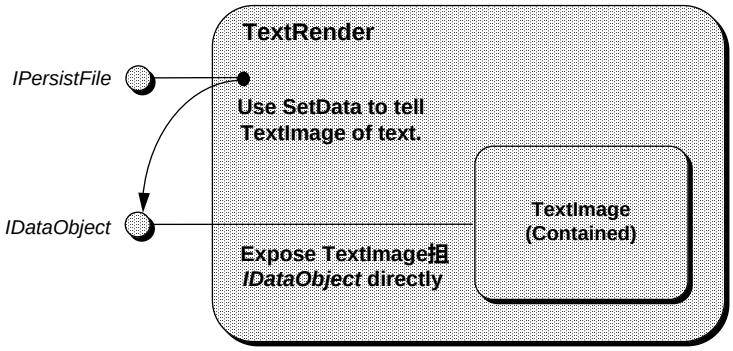


Figure 6-4: When an inner object does a complete job implementing an interface, outer objects may want to expose the interface directly.

The only catch is that we must implement the proper behavior of the IUnknown members in the inner object's (TextImage) IDataObject interface: AddRef and Release have to affect the reference count on the outer object (TextRender) and not the reference count of the inner object. Furthermore, QueryInterface has to be able to return the TextRender object's IPersistFile interface. The solution is to inform the inner object that it is being used in an aggregation such that when it sees IUnknown calls to its interfaces it can delegate those calls to the outer object.

One other catch remains: the outer object must have a means to control the lifetime of the inner object through AddRef and Release as well as have a means to query for the interfaces that only exist on the inner object. For that reason, the inner object *must implement an isolated version of IUnknown that controls the inner object exclusively and never delegates to the outer object.*[60] This requires that the inner object separates the IUnknown members of its functional interfaces from an implementation of IUnknown that strictly controls the inner object itself. In other words, the inner object, to support aggregation, must implement two sets of IUnknown functions: delegating and non-delegating.

This, then, is the mechanism for making aggregation work:

1. When creating the inner object, the outer object must pass its own IUnknown to the inner object through the pUnkOuter parameter of IClassFactory::CreateInstance. pUnkOuter in this case is called the "controlling unknown."
2. The inner object must check pUnkOuter in its implementation of CreateInstance. If this parameter is non-NULL, then the inner object knows it is being created as part of an aggregate. If the inner object does not support aggregation, then it must fail with CLASS_E_NOAGGREGATION. If aggregation is supported, the inner object saves pUnkOuter for later use, but does not call AddRef on it. The reason is that the inner object's lifetime is entirely contained within the outer object's lifetime, so there is no need for the call and to do so would create a circular reference.

[60] An interface with such an IUnknown is sometimes called an "inner" interface on the aggregated object. There may, in general, be several inner interfaces on an object. IRpcProxyBuffer, for example, is one. This is a property of the interface itself, not the implementation.

3. If the inner object detects a non-NULL pUnkOuter in CreateInstance, and the call requests the interface IUnknown itself (as is almost always the case), the inner object must be sure to return its non-delegating IUnknown.
4. If the inner object itself aggregates other objects (which is unknown to the outer object) it must pass the same pUnkOuter pointer it receives down to the next inner object.
5. When the outer object is queried for an interface it exposes from the inner object, the outer object calls QueryInterface in the non-delegating IUnknown to obtain the pointer to return to the client.
6. The inner object must delegate to the controlling unknown, that is, pUnkOuter, all IUnknown calls occurring in any interface it implements other than the non-delegating IUnknown.

Through these steps, the inner object is made aware of the outer object, obtains an IUnknown to which it can delegate calls to insure proper behavior of reference counting and QueryInterface, and provides a way for the outer object to control the inner object's lifetime separately. The mechanism is illustrated in Figure 6-5.

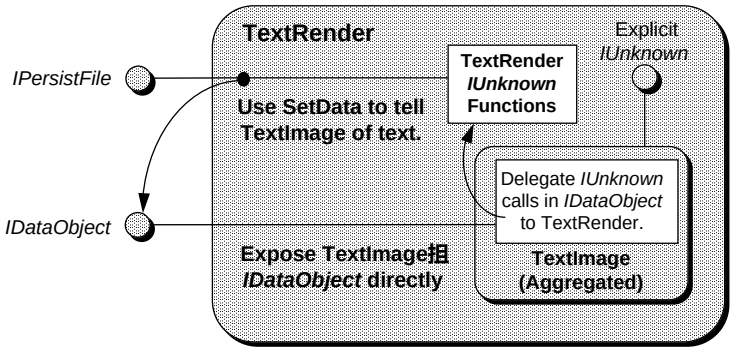


Figure 6-5: Aggregation requires an explicit implementation of *IUnknown* on the inner object and delegation of *IUnknown* function of any other interface to the outer object's *IUnknown* functions.

Now let's look at how this mechanism manifests in code. First off, the TextRender object no longer needs it's own IDataObject implementation and can thus remove it from it's class, but will need to add a member m_pUnkImage to maintain the TextImage's non-delegating *IUnknown*:

```
class CTextRender : public IPersistFile {
    private:
        ULONG       m_cRef;         //Reference Count
        char *      m_pszText;      //Pointer to allocated text
        ULONG       m_cchText;      //Number of characters in m_pszText

        IUnknown *  m_pUnkImage;        //TextImage IUnknown
        //Other internal member functions here

    public:
        [Constructor, Destructor]

        //Outer object IUnknown
        HRESULT QueryInterface(REFIID iid, void ** ppv);
        ULONG AddRef(void);
        ULONG Release(void);

        //IPersistFile Member overrides
        ...
    };
```

In the previous section we saw how the TextRender object would create a TextImage object for containment using CoCreateInstance with the pUnkOuter parameter set to NULL. In aggregation, this parameter will be TextRender's own IUnknown (obtained using a typecast). Furthermore, TextRender must request IUnknown initially from TextImage (storing the pointer in m_pUnkImage):

```
//TextRender initialization
HRESULT     hr;
hr=CoCreateInstance(CLSID_TextImage, CLSCTX_ SERVER, (IUnknown *)this, IID_IUnknown, (void *)&m_pUnkImage);
if (FAILED(hr))
    //TextImage not available, either fail or disable graphic rendering
```

```
//Success:   can now make use of TextImage object.
```

Now, since TextRender does not have it's own IDataObject any longer, its implementation of QueryInterface will use m_pUnkImage to obtain interface pointers:

```
HRESULT CTextRender::QueryInterface(REFIID iid, void ** ppv) {
    *ppv=NULL;

    //This code assumes an overloaded == operator for GUIDs exists
    if (IID_IUnknown==iid)
        *ppv=(void *)(IUnknown *)this;

    if (IID_IPersitFile==iid)
        *ppv=(void *)(IPersistFile *)this;

    if (IID_IDataObject==iid)
        return m_pUnkImage->QueryInterface(iid, ppv);

    if (NULL==*ppv)
        return E_NOINTERFACE;          //iid not supported.

    //Any call to anyone's AddRef is our own.
    AddRef();
    return NOERROR;
    }
```

Note that delegating QueryInterface to the inner object is done only for those interfaces that the outer object knows it wants to expose. The outer object should not delegate the query as a default case, for such blind forwarding without an understanding of the semantic being forwarded will almost assuredly break the outer object should the inner one be revised with new functionality.

.1 Caching interfaces on the inner object

In order to avoid reference counting cycles, special action is needed if the outer object wishes to cache pointers to the inner object's interfaces.

Specifically, if the outer object wishes to cache a to an inner object's interface, once it has obtained the interface from the inner object, the outer object should invoke Release on the punkOuter that was given to the inner object at its instantiation time.

```
// Obtaining inner object interface pointer
pUnkInner->QueryInterface(IID_IFoo, &pIFoo);
pUnkOuter->Release();

// Releasing inner object interface pointer
pUnkOuter->AddRef();
pIFoo->Release();
```

It is suggested that to allow inner objects to do better resource management that controlling objects delay the acquisition of cached pointers and release them when there is no possible use for them.

.2 Efficiency at any Depth of Aggregation

Aggregation has one interesting aspect when aggregates are used on more than one level of an object implementation. Imagine that the TextImage object in the previous example is itself an aggregate object that uses other inner objects. In such a case TextImage will be passing some controlling unknown to those other inner objects. If TextImage is not being aggregated by anyone else, then the controlling unknown is its own; otherwise it passes the pUnkOuter from IClassFactory::CreateInstance on down the line, and any other inner objects that are aggregates themselves do the same.

The net result is that any object in an aggregation, no matter how deeply it is buried in the overall structure, will almost always delegate directly to the controlling unknown if it's interface is exposed from that final outer object. Therefore performance and efficiency of multiple levels of aggregation is not an issue. At worst each delegation is a single extra function call.

7 Emulating Other Servers

The final topic related to COM Servers for this chapter is what is known as emulation: the ability for one server associated with one CLSID to emulate a server of another CLSID. A server that can emulate another is responsible for providing compatible behavior for a different class through a different implementation. This forms the basis for allowing end-users the choice in which servers are used for which objects, as long as the behavior is compatible between those servers.

As far as COM is concerned, it only has to provide some way for a server to indicate that it wishes to emulate some CLSID. To that end, the COM Library supplies the function CoTreatAsClass to establish an emulation that remains in effect (persistently) until canceled or changed. In addition it supplies CoGetTreatAsClass to allow a caller to determine if a given CLSID is marked for emulation.

.1 CoTreatAsClass

HRESULT CoTreatAsClass(clsidOld, clsidNew)

Establish or cancel an emulation relationship between two classes. When clsidNew is emulating clsidOld, calls to CoGetClassObject with clsidOld will transparently use clsidNew. Thus, for example, creating an object of clsidOld will in fact launch the server for clsidNew and have it create the object instead.

This function does no validation on whether an appropriate registration entries exist for clsidNew.

An emulation is canceled by calling this function with clsidOld equal to the original class and clsidNew set to CLSID_NULL.

Argument	Type	Description
clsidOld	REFCID	The class to be emulated.
clsidNew	REFCID	The class which should emulate clsidOld. This replaces any existing emulation for clsidOld. May be CLSID_NULL, in which case any existing emulation for clsidOld is removed.

Return Value	Meaning
S_OK	Success.
CO_E_CLASSNOTREG	*to be described.*
CO_E_READREGDB	*to be described.*
CO_E_WRITEREGDB	*to be described.*
E_UNEXPECTED	An unspecified error occurred.

.2 CoGetTreatAsClass

HRESULT CoGetTreatAsClass(clsidOld, pclsidNew)

Return the existing emulation information for a given class. If no emulation entry exists for clsidOld then clsidOld is returned in pclsidNew.

Argument	Type	Description
clsidOld	REFCID	The class for which the emulation information is to be retrieved.
pclsidNew	CLSID *	The place at which to return the class, if any, which emulates clsidOld. clsidOld is returned if there is no such class. pclsidNew may not be NULL.

Return Value	Meaning
S_OK	Success. A new, (possibly) different CLSID is returned through *pclisdNew.
S_FALSE	Success. The class is emulating itself.
CO_E_READREGDB	.
E_UNEXPECTED	An unspecified error occurred.

How the COM Library implements these functions depends upon the structure of the system registry. For example, under Microsoft Windows, COM uses an additional subkey under an object's CLSID key in the form of:

TreatAs = {*<new CLSID>*}

When the Windows' COM implementation of CoGetClassObject attempts to locate a server for a CLSID, it will always call CoGetTreatAsClass to retrieve the actual CLSID to use. Since CoGetTreatAsClass will return

the same CLSID as passed in if no emulation exists, COM doesn't have to do any special case checks for emulation.

This page intentionally left blank.

This page intentionally left blank.

7. Interface Remoting

In COM, clients communicate with objects solely through the use of vtable-based interface instances. The state of the object is manipulated by invoking functions on those interfaces. For each interface method, the object provides an implementation that does the appropriate manipulation of the object internals.

Interface remoting provides the infrastructure and mechanisms to allow a method invocation to return an interface pointer to an object that is in a different process, perhaps even on a different machine. The infrastructure that performs the remoting of interfaces is transparent to both the client and the object server. Neither the client or object server is necessarily aware that the other party is in fact in a different process.

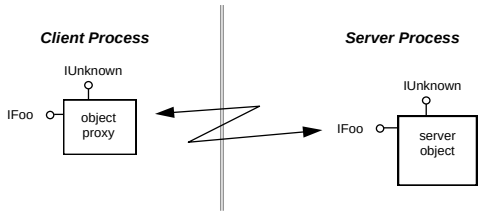
This chapter first explains how interface remoting works giving mention to the interfaces and COM API functions involved. The specifications for the interfaces and the API functions themselves are given later in this chapter. There is also a brief discussion about concurrency management at the end of the chapter that involves an interface called IMessageFilter.

1 How Interface Remoting Works

The crux of the problem to be addressed in interface remoting can be stated as follows:

> "Given an already existing remoted-interface connection between a client process and a server process, how can a method invocation through that connection return a new interface pointer so as to create a second remoted-interface connection between the two processes?"

We state the problem in this way so as to avoid for the moment the issue of how an initial connection is made between the client and the server process; we will return to that later.

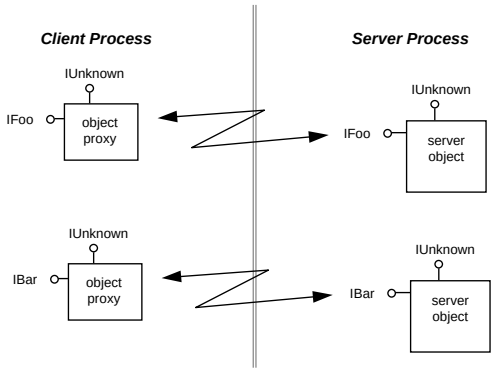


Let's look at an example. Suppose we have an object in a server process which supports an interface IFoo, and that interface of the object (and IUnknown) has sometime in the past been remoted to a client process through some means not here specified. In the client process, there is an object proxy which supports the exact same interfaces as does the original server object, but whose *implementations* of methods in those interfaces are special, in that they forward calls they receive on to calls on the real method implementations back in the server object. We say that the method implementations in the object proxy *marshal* the data, which is then conveyed to the server process, where it is *unmarshaled.* That is, "marshaling" refers to the packaging up of method arguments for transmission to a remote process; "unmarshaling" refers to the unpackaging of this data at the receiving end. Notice that in a given call, the method arguments are marshaled and unmarshaled in one direction, while the return values are marshaled and unmarshaled in the other direction.

For concreteness, let us suppose that the IFoo interface is defined as follows:

```
interface IFoo : IUnknown {
    IBar *      ReturnABar();
    };
```

If the in the client process pFoo->ReturnABar() is invoked, then the object proxy will forward this call on to the IFoo::ReturnABar() method in the server object, which will do whatever this method is supposed to do in order to come up with some appropriate IBar*. The server object is then required to return this IBar* back to the client process. The act of doing this will end up creating a second connection between the two processes:



It is the procedure by which this second connection is established which is the subject of our discussion here. This process involves two steps:

1. On the server side, the IBar* is packaged or marshaled into a data packet.

2. The data packet is conveyed by some means to the client process, where the data it contains is unmarshaled to create the new object proxy.

The term "marshaling" is a general one that is applied in the industry to the packaging of any particular data type, not just interface pointers, into a data packet for transmission through an RPC infrastructure. Each different data type has different rules for how it is to marshaled: integers are to be stored in a certain way, strings are to be stored in a certain way, etc.[61] Likewise, marshaled interface pointers are to be stored in a certain way; the Component Object Model function CoMarshalInterface() contains the knowledge of how this is to be done (note that we will in this document not mention further any kind of marshaling other than marshaling of interface pointers; that subject is well-explored in existing RPC systems).

The process begins with the code doing the marshaling of the returned IBar* interface. This code has in hand a pointer to an interface that it knows in fact to be an IBar* and that it wishes to marshal. To do so it calls CoMarshalInterface(). The first step in CoMarshalInterface() involves finding out whether the object of which this is an interface in fact supports *custom object marshaling* (often simply referred to as "custom marshaling"). Custom object marshaling is a mechanism that permits an object to be in control of creation of remote object proxies to itself. In certain situations, custom object marshaling can be used to create a more efficient object proxy than would otherwise be the case.[62] Use of custom marshaling is completely optional on the object's part; if the object chooses not to support custom marshaling, then *standard interface marshaling* is used to marshal the IBar*. Standard interface marshaling uses a system-provided object proxy implementation in the client process. This standard implementation is a generic piece of code, in that it can be used as the object proxy for any interface on any object. However, the act of marshaling (and unmarshaling) method arguments and return values is inherently interface-specific, since it is highly sensitive to the semantics and data types used in the particular methods in question. To accommodate this, the standard implementation dynamically loads in interface-specific pieces of code as needed in order to do the parameter marshaling.

We shall discuss in great detail in a moment how standard interface marshaling works. First, however, we shall review custom object marshaling, as this provides a solid framework in which standard marshaling can be better understood.

2 Architecture of Custom Object Marshaling

Imagine that we are presently in a piece of code whose job it is to marshal an interface pointer that it has in hand. For clarity, in what follows we'll refer to this piece of code as the "original marshaling stub." The general case is that the original marshaling stub does not *statically*[63] know the particular interface identifier (IID) to which the pointer conforms; the IID may be passed to this code as a second parameter. This is a common paradigm in the Component Object Model. Extant examples of this paradigm include:

```
IUnknown::QueryInterface(REFIID riid, void** ppvObject);
IOleItemContainer::GetObject(..., REFIID riid, void** ppvObject);
IClassFactory::CreateInstance(..., REFIID riid, void** ppvNewlyCreatedObject);
```

[61] In fact, there exist several standard sets of rules, each promoted by a different organization. Two common such sets of rules are known as "Network Data Representation" (NDR) and "External Data Representation" (XDR) chiefly promoted respectively by the Open Software Foundation and Sun Microsystems. ASN.1 is another standard for the same sort of technology.

[62] Notice here that we're only discussing the marshaling of pointers to interfaces, and that the term "custom object marshaling" applies only to the marshaling of this data type. In general in a given remote procedure call the many other kinds of data which appear as function parameters also needs to be marshaled: strings, integers, structures, etc. We shall not concern ourselves here with such other data types, but instead concentrate our discussion on marshaling interface pointers.

[63] i.e.: at compile time of the original marshaling stub

Let us assume the slightly less general case where the marshaling stub in fact does know a little bit about the IID: that the interface in fact derives from IUnknown. This is a requirement for remoting: it is not possible to remote interfaces which are not derived from IUnknown.

To find out whether the object to which it has an interface supports custom marshaling, the original marshaling stub simply does a QueryInterface() for the interface IMarshal. That is, an object signifies that it wishes to do custom marshaling simply by implementing the IMarshal interface. IMarshal is defined as follows:

```
[
    local,
    object,
    uuid(00000003-0000-0000-C000-000000000046)
]
interface IMarshal : IUnknown {
    HRESULT GetUnmarshalClass ( [in] REFIID riid, [in, unique] void *pv,
        [in] DWORD dwDestContext, [in, unique] void *pvDestContext,
        [in] DWORD mshlflags, [out] CLSID *pCid);
    HRESULT GetMarshalSizeMax ([in] REFIID riid, [in, unique] void *pv,
        [in] DWORD dwDestContext, [in, unique] void *pvDestContext,
         [in] DWORD mshlflags, [out] DWORD *pSize);
    HRESULT MarshalInterface ([in, unique] IStream *pStm, [in] REFIID riid, [in, unique] void *pv,
        [in] DWORD dwDestContext, [in, unique] void *pvDestContext, [in] DWORD mshlflags);
    HRESULT UnmarshalInterface ( [in, unique] IStream *pStm, [in] REFIID riid, [out] void **ppv);
    HRESULT ReleaseMarshalData ( [in, unique] IStream *pStm);
    HRESULT DisconnectObject ( [in] DWORD dwReserved);
}
```

The idea is that if the object says "Yes, I do want to do custom marshaling" that the original marshaling stub will use this interface in order to carry out the task. The sequence of steps that carry this out is:

1. Using GetUnmarshalClass, the original marshaling stub asks the object which kind of (i.e.: which class of) proxy object it would like to have created on its behalf in the client process.
2. (optional on the part of the marshaling stub) Using GetMarshalSizeMax, the stub asks the object how big of a marshaling packet it will need. When asked, the object *will* return an upper bound on the amount of space it will need.[64]
3. The marshaling stub allocates a marshaling packet of appropriate size, then creates an IStream* which points into the buffer. Unless in the previous step the marshaling stub asked the object for an upper bound on the space needed, the IStream* must be able to grow its underlying buffer dynamically as IStream::Write calls are made.
4. The original marshaling stub asks the object to marshal its data using MarshalInterface.

We will discuss the methods of this interface in detail later in this chapter.

At this point, the contents of the memory buffer pointed to by the IStream* together with the class tag returned in step (1) comprises all the information necessary in order to be able to create the proxy object in the client process. It is the nature of remoting and marshaling that "original marshaling stubs" such as we have been discussing know how to communicate with the client process; recall that we are assuming that an initial connection between the two processes had already been established. The marshaling stub now communicates to the client process, by whatever means is appropriate, the class tag and the contents of the memory that contains the marshaled interface pointer. In the client process, the proxy object is created as an instance of the indicated class using the standard COM instance creation paradigm. IMarshal is used as the initialization interface; the initialization method is IMarshal::UnmarshalInterface(). The unmarshaling process looks something like the following:

```
void ExampleUnmarshal(CLSID& clsidProxyObject, IStream* pstm, IID& iidOriginallyMarshalled, void** ppvReturn)
{
    IClassFactory* pcf;
    IMarshal* pmsh;
    CoGetClassObject(clsidProxyObject, CLSCTX_INPROC_HANDLER,   NULL, IID_IClassFactory, (void**)&pcf);
    pcf->CreateInstance(NULL, IID_IMarshal, (void**)pmsh);
    pmsh->UnmarshalInterface(pstm, iidOriginallyMarshalled, ppvReturn);
    pmsh->ReleaseMarshalData(pstm)
    pmsh->Release();
```

[64] That is, it is explicitly legal for the caller of GetMarshalSizeMax() to allocate a fixed size marshaling buffer containing no more than the indicated upper bound number of bytes.

```
        pcf->Release();
    }
```

There are several important reasons why an object may choose to do custom marshaling.

- It permits the server implementation, transparently to the client, to be in complete control of the nature of the invocations that actually transition across the network. In designing component architectures, one often runs into a design tension between the interface which for simplicity and elegance one wishes to exhibit to client programmers and the interface that is necessary to achieve efficient invocations across the network. The former, for example, might naturally wish to operate in terms of small-grained simple queries and responses, whereas the latter might wish to batch requests for efficient retrieval. The client and the network interfaces are in design tension; custom marshaling is the crucial hook that allows us to have our cake and eat it too by giving the server implementor the ability to tune the network interface without affecting the interface seen by its client.
  When the object does custom marshaling, the client loses any "COM provided" communication to the original object. If the proxy wants to "keep in touch", it has to connect through some other means (RPC, Named pipe?) to the original object. Custom Object Marshaling can not be done on a per interface basic, because object identity is lost! Custom Object Marshaling is a sophisticated way for an object to pass a copy of an existing instance of itself into another execution context.
- Some objects are of the nature that once they have been created, they are immutable: their internal state does not subsequently change. Many monikers are an example of such objects. These sorts of objects can be efficiently remoted by making independent copies of themselves in client processes. Custom marshaling is the mechanism by which they can do that, yet have no other party be the wiser for it.
- Objects which already are proxy objects can use custom marshaling to avoid creating proxies to proxies; new proxies are instead short-circuited back to the original server. This is both an important efficiency and an important robustness consideration.
- Object implementations whose whole state is kept in shared memory can often be remoted to other process on the same machine by creating an object in the client that talks directly to the shared memory rather than back to the original object. This can be a significant performance improvement, since access to the remoted object does not result in context switches. The present Microsoft Compound File implementation is an example of objects using this kind of custom marshaling.

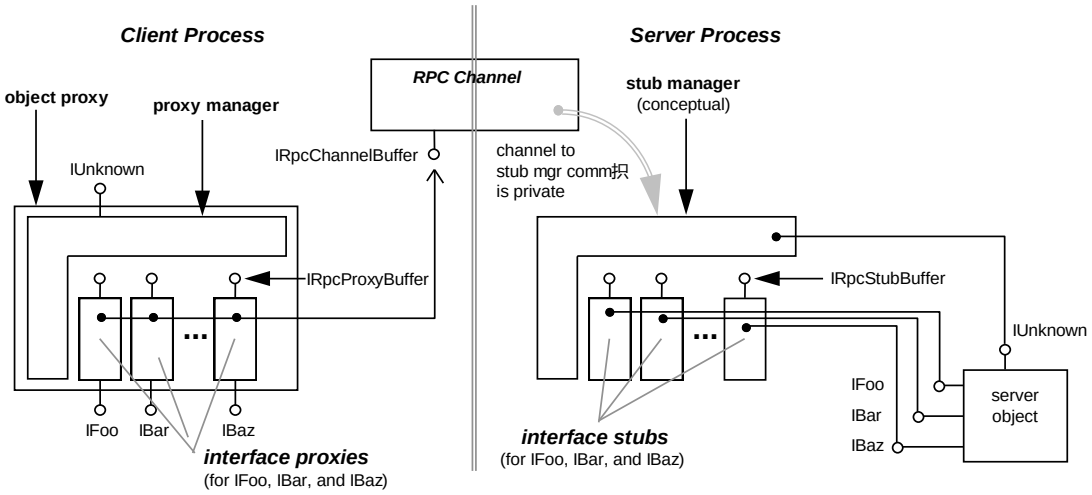
3 Architecture of Standard Interface / Object Marshaling

If the object being marshaled[65] chooses not to implement custom object marshaling, a “default” or “standard” object marshaling technique is used. An important part of this standard marshaling technique involves locating and loading the interface-specific pieces of code that are responsible for marshaling and unmarshaling remote calls to instances of that interface. We call these interface-specific pieces of code used in standard marshaling and unmarshaling “interface proxies” and “interface stubs” respectively.[66] (It is important not to confuse *interface* proxies with the *object* proxy, which relates to the *whole* representative in the client process, rather than just one interface on that representative. We apologize for the subtleties of the terminology.)

The following figure gives an slightly simplified view of how the standard client- and server-side structures cooperate.

[65] Astute readers will notice an abuse of terminology here: what is really being marshaled in hand is one particular interface on the object, not the *whole* object, though in fact in the remote process access to the whole process is indeed obtained: new interfaces on the object will be marshaled later as needed. We trust that this will not lead to too much confusion.

[66] Other RPC systems sometimes instead call these “client side stubs” and “server side stubs.” Sometimes *we* mix things up a bit and refer to “proxy interfaces” and “stub interfaces” instead of “interface proxies” and “interface stubs.”



Simplified conceptual view of client - server remoting structures

When an interface of type IFoo needs to be remoted, a system registry is consulted under a key derived from IID_IFoo to locate a class id that implements the interface proxy and interface stub for the given interface. Both the interface proxies and the interface stubs for a given interface must be implemented by the same class. Most often, this class is automatically generated by a tool whose input is a description of the function signatures and semantics of the interface, written in some "interface description language," often known as "IDL." However, while highly recommended and encouraged for accuracy's sake, the use of such a tool is by no means required; interface proxies and stubs are merely Component Object Model components which are used by the RPC infrastructure, and as such, can be written in any manner desired so long as the correct external contracts are upheld. *From a logical perspective, it is ultimately the programmer who is the designer of a new interface who is responsible for ensuring that all interface proxies and stubs that ever exist agree on the representation of their marshaled data.* The programmer has the freedom to achieve this by whatever means he sees fit, but with that freedom comes the responsibility for ensuring the compatibility.

In the figure, the "stub manager" is "conceptual" in the sense that while it useful in this documentation to have a term to refer to the pieces of code and state on in the server-side RPC infrastructure which service the remoting of a given object, there is no direct requirement that the code and state take any particular well-specified form.[67] In contrast, on the client side, there is an identifiable piece of state and associated behavior which appears to the client code to be the one, whole object. The term "proxy manager" is used to refer to the COM Library provided code that manages the client object identity, etc., and which dynamically loads in interface proxies as needed (per QueryInterface calls). The proxy manager implementation is intimate with the client-side RPC channel implementation, and the server-side RPC channel implementation is intimate with the stub manager implementation.

Interface proxies are created by the client-side COM Library infrastructure using a code sequence resembling the following:

```
clsid = LookUpInRegistry(key derived from iid)
CoGetClassObject(clsid, CLSCTX_SERVER, NULL, IID_IPSFactoryBuffer, &pPSFactory));
pPSFactory->CreateProxy(pUnkOuter, riid, &pProxy, &piid);
```

Interface stubs are created by the server-side RPC infrastructure using a code sequence resembling:

```
clsid = LookUpInRegistry(key derived from iid)
CoGetClassObject(clsid, CLSCTX_SERVER, NULL, IID_IPSFactoryBuffer, &pPSFactory));
pPSFactory->CreateStub(iid, pUnkServer, &pStub);
```

In particular, notice that the class object is talked-to with IPSFactoryBuffer interface rather than the more common IClassFactory.

The interfaces mentioned here are as follows:

```
interface IPSFactoryBuffer : IUnknown {
    HRESULT     CreateProxy(pUnkOuter, iid, ppProxy, ppv);
```

[67] There are, however, implied requirements for the existence of some piece of code / state that manages the *entire set of* external remoting connections for a given object. See CoLockObjectExternal(), for example.

```
        HRESULT     CreateStub(iid, pUnkServer, ppStub);
        };

    interface IRpcChannelBuffer : IUnknown {
        HRESULT     GetBuffer(pMessage, riid);
        HRESULT     SendReceive(pMessage, pStatus);
        HRESULT     FreeBuffer(pMessage);
        HRESULT     GetDestCtx(pdwDestCtx, ppvDestCtx);
        HRESULT     IsConnected();
        };

    interface IRpcProxyBuffer : IUnknown {
        HRESULT     Connect(pRpcChannelBuffer);
        void        Disconnect();
        };

    interface IRpcStubBuffer : IUnknown {
        HRESULT         Connect(pUnkServer);
        void                Disconnect();
        HRESULT         Invoke(pMessage, pChannel);
        IRPCStubBuffer* IsIIDSupported(iid);
        ULONG           CountRefs();
        HRESULT         DebugServerQueryInterface(ppv);
        void            DebugServerRelease(pv);
        };
```

Suppose an interface proxy receives a method invocation on one of it's interfaces (such as IFoo, IBar, or IBaz in the above figure). The interface proxy's implementation of this method first obtains a marshaling packet from its RPC channel using IRpcChannelBuffer::GetBuffer(). The process of marshaling the arguments will copy data into the buffer. When marshaling is complete, the interface proxy invokes IRpcChannelBuffer::SendReceive() to send the method invocation across the "wire" to the corresponding interface stub. When IRpcChannelBuffer::SendReceive() returns, the contents of buffer into which the arguments were marshaled will have been replaced by the return values marshaled from the interface stub. The interface proxy unmarshals the return values, invokes IRpcChannelBuffer::FreeBuffer() to free the buffer, then returns the return values to the original caller of the method.

It is the implementation of IRpcChannelBuffer::SendReceive() that actually sends the request over to the server process. It is only the channel who knows or cares how to identify the server process and object within that process to which the request should be sent; this encapsulation allows the architecture we are describing here to function for a variety of different kinds of channels: intra-machine channels, inter-machine channels (i.e.: across the network), etc. The channel implementation knows how to forward the request onto the appropriate stub manager object in the appropriate process. From the perspective of this specification, the channel and the stub manager are intimate with each other (and intimate with the proxy manager, for that matter). Through this intimacy, eventually the appropriate interface stub receives an IRpcStubBuffer::Invoke() call. The stub unmarshals the arguments from the provided buffer, invokes the indicated method on the server object, and marshals the return values back into a new buffer, allocated by a call to IRpcChannelBuffer::GetBuffer(). The stub manager and the channel then cooperate to ferry the return data packet back to the interface proxy, who is still in the middle of IRpcChannelBuffer::SendReceive(). IRpcChannelBuffer::SendReceive() returns to the proxy, and we proceed as just described above.

When created, interface proxies are always aggregated into the larger object proxy: at interface-proxy-creation time, the proxy is given the IUnknown* to which it should delegate its QueryInterface(), etc., calls, as per the usual aggregation rules. When connected, the interface proxy is also given (with IRpcProxyBuffer::Connect()) a pointer to an IRpcChannelBuffer interface instance. It is through this pointer that the interface proxy actually sends calls to the server process. Interface proxies bring a small twist to the normal everyday aggregation scenario. In aggregation, each interface supported by an aggregateable object is classified as either "external" or "internal." External interfaces are the norm. They are the ones whose instances are exposed directly to the clients of the aggregate as whole. It is *always* the case that a QueryInterface() that requests an external interface of an aggregated object should be delegated by the object to its controlling unknown (ditto for AddRef() and Release()). Internal interfaces, on the other hand, are never exposed to outside clients. Instead, they are solely for the use of the controlling unknown in manipulating the aggregated object. QueryInterface() for internal interfaces should *never* be delegated to the controlling un-

known (ditto again). In the common uses of aggregation, the IUnknown interface on the object is the only internal interface. The twist that interface proxies bring is that IRpcProxyBuffer is *also* an internal interface.

Interface stubs, by contrast with interface proxies, are not aggregated, since there is no need that they appear to some external client to be part of a larger whole. When connected, an interface stub is given (with IRpcStubBuffer::Connect()) a pointer to the server object to which they should forward invocations that they receive.

A given interface proxy instance can if it chooses to do so service more than one interface. For example, in the above figure, one interface proxy could have chosen to service *both* IFoo and IBar. To accomplish this, in addition to installing itself under the appropriate registry entries, the proxy should support QueryInterface()ing from one supported interface (and from IUnknown and IRpcProxyBuffer) to the other interfaces, as usual. When the Proxy Manager in a given object proxy finds that it needs the interface proxy for some new interface that it doesn't already have, before it goes out to the registry to load in the appropriate code using the code sequence described above, it first does a QueryInterface() for the new interface id (IID) on all of its *existing* interface proxies. If one of them supports the interface, then it is used rather than loading a new interface proxy.

Interface stub instances, too, can service more than one interface on a server object. However, the extent to which they can do so is quite restricted: a given interface stub instance may support one or more interfaces only if that set of interfaces has in fact a strict single-inheritance relationship. In short, a given interface stub needs to know how to interpret a given method number that it is asked to invoke without at that same time also being told the interface id (IID) in which that method belongs; the stub must already *know* the relevant IID. The IID which an interface stub is initially created to service is passed as parameter to IPSFactoryBuffer::CreateStub(). After creation, the interface stub may from time to time be asked using IRpcStubBuffer::IsIIDSupported() if it in fact would also like be used to service another IID. If the stub also supports the second IID, then it should return the appropriate IRpcStubBuffer* for that IID; otherwise, the stub buffer should return NULL. This permits the stub manager in certain cases to optimize the loading of interface stubs.

Both proxies and stubs will at various times have need to allocate or free memory. Interface proxies, for example, will need to allocate memory in which to return out parameters to their caller. In this respect interface proxies and interface stubs are just normal Component Object Model components, in that they should use the standard task allocator; see CoGetMalloc(). See also the earlier discussion regarding specific rules for passing in, out, and in out pointers.

On Microsoft Windows platforms, the "key derived from IID" under which the registry is consulted to learn the proxy/stub class is as follows:

Interfaces
{IID}
ProxyStubClsid32 = {CLSID}

Here {CLSID} is a shorthand for any class id; the actual value of the unique id is put between the {}'s; e.g. {DEADBEEF-DEAD-BEEF-C000-000000000046}; all digits are upper case hex and there can be no spaces. This string format for a unique id (without the {}'s) is the same as the OSF DCE™ standard and is the result of the StringFromCLSID routine. {IID} is a shorthand for an interface id; this is similar to {CLSID}; StringFromIID can be used to produce this string.

4 Architecture of Handler Marshaling

Handler marshaling is a third variation on marshaling, one closely related to standard marshaling. Colloquially, one can think of it as a middle ground between raw standard marshaling and full custom marshaling.

In handler marshaling, the object specifies that it would like to have some amount of client-side state; this is designated by the class returned by IStdMarshalInfo::GetClassForHandler. However, this handler class rather than fully taking over the remoting to the object instead aggregates in the default handler, which carries out the remoting in the standard manner as described above.

5 Standards for Marshaled Data Packets

In the architecture described here, nothing has yet to be said about representation or format standards for the data that gets placed in marshaling packets. There is a good reason for this. In the Component Object Model architecture, the only two parties that have to agree on what goes into a marshaling packet are the code that marshals the data into the packet and the code that unmarshals it out again: the interface proxies and the interface stubs. So long as we are dealing only with intra-machine procedure calls (i.e.: non-network), then we can reasonably assume that pairs of interface proxies and stubs are always installed together on the machine. In this situation, we have no need to specify a packet format standard; the packet format can safely be a private matter between the two piece of code.

However, once a network is involved, relying on the simultaneous installation of corresponding interface proxies and stubs (on different machines) is no longer a reasonable thing to do. Thus, when the a method invocation is in fact remoted over a network, it is strongly recommended that the data marshaled into the packet to conform to a published standard (NDR), though, as pointed out above, it is technically the interface-designer's responsibility to achieve this correspondence by whatever means he sees fit.

6 Creating an Initial Connection Between Processes

Earlier we said we would later discuss how an initial remoting connection is established between two processes. It is now time to have that discussion.

The real truth of the matter is that the initial connection is established by some means outside of the architecture that we have been discussing here. The minimal that is required is some primitive communication channel between the two processes. As such, we cannot hope to discuss all the possibilities. But we will point out some common ones.

One common approach is that initial connections are established just like other connections: an interface pointer is marshaled in the server process, the marshaled data packet is ferried the client process, and it is unmarshaled. The only twist is that the ferrying is done by some means *other* than the RPC mechanism which we've been describing. There are many ways this could be accomplished. The most important, by far is one where the marshaled data is passed as an out-parameter from an invocation on a well-known endpoint to a Service Control Manager.

7 Marshaling Interface and Function Descriptions

Having discussed on a high level how various remoting related interfaces work together, we now present each of them in detail.

.1 IPSFactoryBuffer Interface

IPSFactoryBuffer is the interface through which proxies and stubs are created. It is used to create proxies and stubs that support IRpcProxyBuffer and IRpcStubBuffer respectively. Each proxy / stub DLL must support IPSFactory interface on the class object accessible through its DllGetClassObject() entry point. As was described above, the registry is consulted under a key derived from the IID to be remoted in order to learn the proxy/stub class that handles the remoting of the indicated interface. The class object for this class is retrieved, asking for this interface. A proxy or a stub is then instantiated as appropriate.

```
interface IPSFactoryBuffer : IUnknown {
    HRESULT     CreateProxy(pUnkOuter, iid, ppProxy, ppv);
    HRESULT     CreateStub(iid, pUnkServer, ppStub);
    };
```

.1 IPSFactoryBuffer::CreateProxy

HRESULT IPSFactoryBuffer::CreateProxy(pUnkOuter, iid, ppProxy, ppv)

Create a new interface proxy object. This function returns both an IRpcProxy instance and an instance of the interface which the proxy is being created to service in the first place. The newly created proxy is initially in the unconnected state.

Argument	Type	Description
pUnkOuter	IUnknown *	the controlling unknown of the aggregate in which the proxy is being created.
iid	REFIID	the interface id which the proxy is being created to service, and of which an instance should be returned through ppv.
ppProxy	IRpcProxyBuffer**	on exit, contains the new IRpcProxyBuffer instance.
ppv	void **	on exit, contains an interface pointer of type indicated by iid.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_NOINTERFACE, E_UNEXPECTED, no others.

.2 IPSFactoryBuffer::CreateStub

HRESULT IPSFactoryBuffer::CreateStub(iid, pUnkServer, ppStub)

Create a new interface stub object. The stub is created in the connected state on the object indicated by pUnkServer.

If pUnkServer is non-NULL, then before this function returns the stub must verify (by using QueryInterface()) that the server object in fact supports the interface indicated by iid. If it does not, then this function should fail with the error E_NOINTERFACE.

Argument	Type	Description
iid	REFIID	the interface that the stub is being created to service
pUnkServer	IUnknown*	the server object that is being remoted. The stub should delegate in-coming calls (see IRpcStubBuffer::Invoke()) to the appropriate interface on this object. pUnkServer may legally be NULL, in which case the caller is responsible for later calling IRpcStubBuffer::Connect() before using IRpcStubBuffer::Invoke().
ppStub	IRpcStubBuffer**	the place at which the newly create stub is to be returned.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_NOINTERFACE, E_UNEXPECTED, no others.

.2 IRpcChannelBuffer interface

IRpcChannelBuffer is the interface through which interface proxies send calls through to the corresponding interface stub. This interface is implemented by the RPC infrastructure. The infrastructure provides an instance of this interface to interface proxies in IRpcProxyBuffer::Connect(). The interface proxies hold on to this instance and use it each time they receive an incoming call.

```
interface IRpcChannelBuffer : IUnknown {
    HRESULT     GetBuffer(pMessage, riid);
    HRESULT     SendReceive(pMessage, pStatus);
    HRESULT     FreeBuffer(pMessage);
    HRESULT     GetDestCtx(pdwDestCtx, ppvDestCtx);
    HRESULT     IsConnected();
    };
```

.1 RPCOLEMESSAGE and related structures

Common to several of the methods in IRpcChannelBuffer is a data structure of type RPCOLEMESSAGE. This structure is defined as is show below. The structure is to be packed so that there are no holes in its memory layout.

```
typedef struct RPCOLEMESSAGE {
    void *                  reserved1;
    RPCOLEDATAREP           dataRepresentation;   // in NDR transfer syntax: info about endianness, etc.
    void        *               pvBuffer;             // memory buffer used for marshalling
    ULONG                   cbBuffer;             // size of the marshalling buffer
    ULONG                   iMethod;              // the method number being invoked
    void *                  reserved2[5];
    ULONG                   rpcFlags;
```

} on the ultimate destination machine **MESSAGE**;[68]

The most significant member of this structure is pvBuffer. It is through the memory buffer to which pvBuffer points that marshaled method arguments are transferred. cbBuffer is used to indicate the size of the buffer. iMethod is indicates a particular method number within the interface being invoked. The IID of that interface is identified through other means: on the client side as a parameter to GetBuffer(), and on the server side as part of the internal state of each interface stub.

At all times all reserved values in this structure are to be initialized to zero by non-RPC-infrastructure parties (i.e.: parties other than the channel / RPC runtime implementor) who allocate RPCOLEMESSAGE structures. However, the RPC channel (more generally, the RPC runtime infrastructure) is free to modify these reserved fields. Therefore, once initialized, the reserved fields must be ignored by the initializing code; they cannot be relied on to remain as zero. Further, there are very carefully specified rules as to what values in these structures may or may not be modified at various times and by which parties. In almost all cases, aside from actually reading and writing data from the marshaling buffer, which is done by proxies and stubs, only the channel may change these fields. See the individual method descriptions for details.

Readers familiar with the connection-oriented DCE protocol may notice that the "transfer syntax" used for marshaling the arguments, the particular set of rules and conventions according to which data is marshaled, is not explicitly called out. Architecturally speaking, it is only the interface proxy for a given interface and its corresponding interface stub that cares at all about what set of marshaling rules is in fact used. However, in the general case these interface proxies and stubs may be installed on different machines with a network in the middle, be written by different development organizations on different operating systems, etc. Accordingly, in cases where the author of an interface proxy for a given IID cannot guarantee that all copies of the corresponding interface stub are in fact always revised and updated in synchrony with his interface proxy, a well-defined convention should be used for the transfer syntax. Indeed, formal transfer syntax standards exist for this purpose. The one most commonly used is known as "Network Data Representation" (NDR), originally developed by Apollo Corporation and subsequently enhanced and adopted by the Open Software Foundation as part of their Distributed Computing Environment (DCE). The Windows NT operating system also uses NDR in its RPC implementation. Unless very good reasons exist to do otherwise, programmers are encouraged to use the NDR transfer syntax.

When NDR transfer syntax is used (and whether it *is* in use or not is implicitly known by the proxy or stub), the member dataRepresentation provides further information about the rules by which data in the buffer is marshaled. NDR is a "multi-canonical" standard, meaning that rather than adopting one standard for things like byte-order, character set, etc., multiple standards (a fixed set of them) are accommodated. Specifically, this is accommodated by a "reader make right" policy: the writer / marshaler of the data is free to write the data in any of the supported variations and the reader / unmarshaler is expected to be able to read any of them. The particular data type in use is conveyed in an RPCOLEDATAREP structure, which is defined as follows. Note that this structure, too, is packed; the size of the entire structure is exactly four bytes. The actual layout of the structure in all cases always corresponds to the data representation value as defined in the DCE standard; the particular structure shown here is equivalent to that layout in Microsoft's and other common compilers.

```
typedef RPCOLEDATAREP {
    UINT        uCharacterRep   : 4;    // least signficant nibble of first byte
    UINT        uByteOrder      : 4;    // most signficant nibble of first byte
    BYTE        uFloatRep;
    BYTE        uReserved;
    BYTE        uReserved2;
    } RPCOLEDATAREP;
```

The values which may legally be found in these fields are as shown in Table 1. Further information on the interpretation of this field can be found in the NDR Transfer Syntax standards documentation.

[68] The layout of this structure is as odd as it is for historical reasons. Apologies are extended to those whose design aesthetics are offended.

Field Name	Meaning of Field	Value in field	Interpretation
uCharacterRep	determines interpretation of single-byte-character valued and single-byte-string valued entities	0	ASCII
		1	EBCDIC
uByteOrder	integer and floating point byte order	0	Big-endian (Motorola)
		1	Little-endian (Intel)
uFloatRep	representation of floating point numbers	0	IEEE
		1	VAX
		2	Cray
		3	IBM

Table 1. Interpretation of dataPresentation

.2 IRpcChannelBuffer::GetBuffer

HRESULT IRpcChannelBuffer::GetBuffer(pMessage, iid)

This method returns a buffer into which data can be marshaled for subsequent transmission over the wire. It is used both by interface proxies and by interface stubs, the former to marshal the incoming arguments for transmission to the server, and the latter to marshal the return values back to the client.

Upon receipt of an incoming call from the client of the proxy object, interface proxies use GetBuffer() to get a buffer into which they can marshaling the incoming arguments. A new buffer must be obtained for every call operation; old buffers cannot be reused by the interface proxy. The proxy needs to ask for and correctly manage a new buffer even if he himself does not have arguments to marshal (i.e.: a void argument list).[69] Having marshaled the arguments, the interface proxy then calls SendReceive() to actually invoke the operation. Upon return from SendReceive(), the buffer no longer contains the marshaled arguments but instead contains the marshaled return values (and out parameter values). The interface proxy unmarshals these values, calls FreeBuffer() to free the buffer, then returns to its calling client.

On the server side (in interface stubs), the sequence is somewhat different. The server side will not be explored further here; see instead the description of IRpcStubBuffer::Invoke() for details.

On the client side, the RPCOLEMESSAGE structure argument to GetBuffer() has been allocated and initialized by the caller (or by some other party on the caller's behalf). Interface proxies are to initialize the members of this structure as follows.

Member Name	Value to initalize to
reserved members	as always, reserved values must be initialized to zero / NULL.
pvBuffer	must be NULL.
cbBuffer	the size in bytes that the channel should allocate for the buffer; that is, the maximum size in bytes needed to marshal the arguments. The interface proxy will have determined this information by considering the function signature and the particular argument values passed in. It is explicitly legal to have this value be zero, indicating that that the caller does not himself require a memory buffer.
iMethod	the zero-based method number in the interface iid which is being invoked
dataRepresentation	if NDR transfer syntax is being used, then this indicates the byte order, etc., by which the caller will marshal data into the returned buffer.
rpcFlags	♦ *Exact values to be listed here.*

If the GetBuffer() function is successful, then upon function exit pvBuffer will have been changed by the channel to point to a memory buffer of (at least) cbBuffer bytes in size into which the method arguments can now be marshaled (if cbBuffer was zero, pvBuffer may or may not be NULL). The reserved fields in the RPCOLEMESSAGE structure may or may not have been changed by the channel. However, neither the cbBuffer nor iMethod fields of RPCOLEMESSAGE will have been changed; the channel treats these as

[69] This permits the channel to behind-the-scenes add additional space into the buffer. Such a capability is needed, for example, in order to support remote debugging.

read-only.[70] Furthermore, until such time as the now-allocated memory buffer is subsequently freed (see SendReceive() and FreeBuffer()), no party other than the channel may modify any of the data accessible from pMessage with the lone exceptions of the data pointed to by pvBuffer and the member cbBuffer, which may be modified only in limited ways; see below.

The arguments to GetBuffer() are as follows:

Argument	Type	Description
pMessage	RPCOLEMESSAGE *	a message structure initialized as discussed above.
iid	REFIID	the interface identifier of the interface being invoked.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_UNEXPECTED

.3 IRpcChannelBuffer::SendReceive

HRESULT IRpcChannelBuffer::SendReceive(pMessage, pStatus)

Cause an invocation to be sent across to the server process. The caller will have first obtained access to a transmission packet in which to marshal the arguments by calling IRpcChannelBuffer::GetBuffer(). The same pMessage structure passed as an argument into that function is passed here to the channel a second time.

In the intervening time period, the method arguments will have been marshaled into the buffer pointed to by pMessage->pvBuffer. However, the pvBuffer pointer parameter must on entry to SendReceive() be exactly as it was when returned from GetBuffer(). That is, it must point to the start of the memory buffer. The caller should in addition set pMessage->cbBuffer to the number of bytes actually written into the buffer (zero is explicitly a legal value). No other values accessible from pMessage may be different than they were on exit from GetBuffer().

Upon *successful* exit from SendReceive(), the incoming buffer pointed to by pvBuffer will have been freed by the channel. In its place will be found a buffer containing the marshaled return values / out parameters from the interface stub: pMessage->pvBuffer points to the new buffer, and pMessage->cbBuffer indicates the size thereof. If there are no such return values, then pMessage->cbBuffer is set to zero, while pMessage->pvBuffer may or may not be NULL.

On *error* exit from SendReceive(),[71] the incoming buffer pointed to by pvBuffer may or may not have been freed. If it has been freed, then on error exit pMessage->pvBuffer is set to NULL and pMessage->cbBuffer is set to zero. If in contrast, pMessage->pvBuffer is on error exit not NULL, then that pointer, the data to which it points, and the value pMessage->cbBuffer will contain exactly as they did on entry; that is, the marshaled arguments will not have been touched. Thus, on error exit from SendReceive(), in no case are any marshaled *return values* passed back; if a marshaling buffer is in fact returned, then it contains the marshaled *arguments* as they were on entry.

The exact cases on error exit when the incoming buffer has or has not been freed needs careful attention. There are three cases:

1) The channel implementation knows with certainty either that all of the incoming data was successfully unmarshaled or that if any errors occurred during unmarshaling that the interface stub correctly cleaned up. In practical terms, this condition is equivalent to the stub manager having actually called IRpcStubBuffer::Invoke() on the appropriate interface stub.

   In this case, on exit from SendReceive() the incoming arguments will *always* have been freed.

2) The channel implementation knows with certainty the situation in case 1) has *not* occurred.

   In this case, on exit from SendReceive(), the incoming arguments will *never* have been freed.

3) The channel implementation does not know with certainty that either of the above two cases has occurred.

   In this case, on exit from SendReceive(), the incoming arguments will *always* have been freed. This is a possible resource leakage (due to, for example, CoReleaseMarshalData() calls that never get made), but it safely avoids freeing resources that should not be freed.

[70] The fact that cbBuffer is unchanged can be of particular use to interface stubs. See IRpcStubBuffer::Invoke().

[71] That is, if SendReceive() returns an error. Note that this does NOT indicate an error returned from the function invocation on the server object, for in that case SendReceive() returns success; rather, it indicates an error that occurred somewhere in the RPC transmission.

If pMessage->pvBuffer is returned as non-NULL, then the caller is responsible for subsequently freeing it; see FreeBuffer(). A returned non-NULL pMessage->pvBuffer may in general legally be (and will commonly be, the success case) different than the (non-NULL) value on entry; i.e.: the buffer may be legally be reallocated. Further, between the return from SendReceive() and the subsequent freeing call no data accessible from pMessage may be modified, with the possible exception of the data actually in the memory buffer.

Upon successful exit from SendReceive(), the pMessage->dataRepresentation field will have been modified to contain whatever was returned by the interface stub in field of the same name value on exit to IRpcStubBuffer::Invoke(). This is particularly important when NDR transfer syntax is used, as dataRepresentation indicates critical things (such as byte order) which apply to the marshaled return / out values. Upon error exit from SendReceive(), pMessage->dataRepresentation is undefined.

Argument	Type	Description
pMessage	RPCOLEMESSAGE *	message structure containing info to transmit to server.
pStatus	ULONG *	may legally be NULL. If non-NULL, then if either 1) an RPC-infrastructure-detected server-object fault (e.g.: a server object bug caused an exception which was caught by the RPC infrastructure) or 2) an RPC communications failure occurs, then at this location a status code is written which describes what happened. In the two error cases, the errors E_RPCFAULT and E_RPCSTATUS are (respectively) returned (and are always returned when these errors occur, irrespective of the NULL-ness of pStatus).
return value	HRESULT	S_OK, E_RPCFAULT, E_RPCSTATUS

.4 IRpcChannelBuffer::FreeBuffer

HRESULT IRpcChannelBuffer::FreeBuffer(pMessage)

Free a memory buffer in pMessage->pvBuffer that was previously allocated by the channel.

At various times the RPC channel allocates a memory buffer and returns control of same to a calling client. Both GetBuffer() and SendReceive() do so, for example. FreeBuffer() is the means by which said calling client informs the channel that it is done with the buffer.

On function entry, the buffer which is to be freed is pMessage->pvBuffer, which explicitly may or may not be NULL. If pMessage->pvBuffer is non-NULL, then FreeBuffer() frees the buffer, NULLs the pointer, and returns NOERROR; if pMessage->pvBuffer is NULL, then FreeBuffer() simply returns NOERROR (i.e.: passing NULL is *not* an error). Thus, on function exit, pMessage->pvBuffer is always NULL. Notice that pMessage->cbBuffer is never looked at or changed.

There are strict rules as to what data accessible from pMessage may have been modified in the intervening time between the time the buffer was allocated and the call to FreeBuffer(). In short, very little modification is permitted; see above and below for precise details.

Argument	Type	Description
pMessage	RPCOLEMESSAGE *	pointer to structure containing pointer to buffer to free.
return value	HRESULT	S_OK, E_UNEXPECTED

.5 IRpcChannelBuffer::GetDestCtx

HRESULT IRpcChannelBuffer::GetDestCtx(pdwDestCtx, ppvDestCtx)

Return the destination context for this RPC channel. The destination context here is as specified in the description of the IMarshal interface.

Argument	Type	Description
pdwDestCtx	DWORD *	the place at which the destination context is to be returned.
ppvDestCtx	void **	May be NULL. If non-NULL, then this is the place at which auxiliary information associated with certain destination contexts will be returned. Interface proxies may not hold on to this returned pointer in

		their internal state; rather, they must assume that a subsequent call to IRpcChannel::Call() may in fact invalidate a previously returned destination context.[72]
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_UNEXPECTED, but no others.

.6 IRpcChannelBuffer::IsConnected

HRESULT IRpcChannelBuffer::IsConnected()

Answers as to whether the RPC channel is still connected to the other side. A negative reply is definitive: the connection to server end has definitely been terminated. A positive reply is tentative: the server end may or may not be still up. Interface proxies can if they wish use this method as an optimization by which they can quickly return an error condition.

Argument	Type	Description
return value	HRESULT	S_OK, S_FALSE. No error values may be returned.

.3 IRpcProxyBuffer Interface

IRpcProxyBuffer interface is the interface by which the client-side infrastructure (i.e. the proxy manager) talks to the interface proxy instances that it manages. When created, proxies are aggregated into some larger object as per the normal creation process (where pUnkOuter in IPSFactoryBuffer::CreateProxy() is non-NULL). The controlling unknown will then QueryInterface() to the interface that it wishes to expose from the interface proxy.

```
interface IRpcProxyBuffer : IUnknown {
    virtual    HRESULT    Connect(pRpcChannelBuffer) = 0;
    virtual    void       Disconnect() = 0;
    };
```

.1 IRpcProxyBuffer::Connect

HRESULT IRpcProxyBuffer::Connect(pRpcChannelBuffer)

Connect the interface proxy to the indicated RPC channel. The proxy should hold on to the channel, AddRef()ing it as per the usual rules. If the proxy is currently connected, then this call fails (with E_UNEXPECTED); call Disconnect() first if in doubt.

Argument	Type	Description
pRpcChannelBuffer	IRpcChannelBuffer*	the RPC channel that the interface proxy is to use to effect invocations to the server object. May not be NULL.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_NOINTERFACE, E_UNEXPECTED

.2 IRpcProxyBuffer::Disconnect

void IRpcProxyBuffer::Disconnect()

Informs the proxy that it should disconnect itself from any RPC channel that it may currently be holding on to. This will involve Release()ing the IRpcChannel pointer to counteract the AddRef() done in IRpcProxy::Connect().

Notice that this function does not return a value.

.4 IRpcStubBuffer interface

IRpcStubBuffer is the interface used on the server side by the RPC runtime infrastructure (herein referred to loosely as the "channel") to communicate with interface stubs that it dynamically loads into a server process.

[72] It is possible that in the future a less restrictive rule as to the duration in which the interface proxy may hold on to ppvDestCtxt may be established, such as (perhaps) guaranteeing that the pointer is valid for the lifetime of the interface proxy itself. However, as it stands today, the rule, as stated here, is in fact the law.


```
interface IRpcStubBuffer : IUnknown {
    virtual   HRESULT          Connect(pUnkServer) = 0;
    virtual   void             Disconnect() = 0;
    virtual   HRESULT          Invoke(pMessage, pChannel) = 0;
    virtual   IRpcStubBuffer*  IsIIDSupported(iid) = 0;
    virtual   ULONG            CountRefs() = 0;
    virtual   HRESULT          DebugServerQueryInterface(ppv) = 0;
    virtual   void             DebugServerRelease(pv) = 0;
    };
```

.1 IRpcStubBuffer::Connect

HRESULT IRpcStubBuffer::Connect(pUnkServer)

Informs the interface stub of server object to which it is now to be connected, and to which it should forward all subsequent Invoke() operations. The stub will have to QueryInterface() on pUnkServer to obtain access to appropriate interfaces. The stub will of course follow the normal AddRef() rules when it stores pointers to the server object in its internal state.

If the stub is currently connected, then this call fails with E_UNEXPECTED.

Argument	Type	Description
pUnkServer	IUnknown *	the new server object to which this stub is now to be connected.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_NOINTERFACE, E_UNEXPECTED

.2 IRpcStubBuffer::Disconnect

void IRpcStubBuffer::Disconnect()

Informs the stub that it should disconnect itself from any server object that it may currently be holding on to. Notice that this function does not return a value.

.3 IRpcStubBuffer::Invoke

HRESULT IRpcStubBuffer::Invoke(pMessage, pChannel)

Invoke the pMessage->iMethod'th method in the server object interface instance to which this interface stub is currently connected. The RPC runtime infrastructure (the "channel") calls this method on the appropriate interface stub upon receipt of an incoming request from some remote client. See the discussion on page 130 regarding how interface stubs implicitly know the IID which they are servicing.

On entry, the members of pMessage are set as follows:

Member Name	Value on entry to Invoke()
reserved members	indeterminate. These members are neither to be read nor to be changed by the stub.
pvBuffer	points to a buffer which contains the marshaled incoming arguments. In the case that there are no such arguments (i.e.: cbBuffer == 0), pvBuffer may be NULL, but will not necessarily be so.
cbBuffer	the size in bytes of the memory buffer to which pvBuffer points. If pvBuffer is NULL, then cbBuffer will be zero (but the converse is not necessarily true, as was mentioned in pvBuffer).
iMethod	the zero-based method number in the interface which is being invoked
dataRepresentation	if NDR transfer syntax is being used, then this indicates the byte order, etc., according to which the data in pvBuffer has been marshaled.
rpcFlags	indeterminate. Neither to be read nor to be changed by the stub.

The stub is to do the following:

- unmarshal the incoming arguments,
- invoke the designated operation in the server object,
- ask the channel to allocate a new buffer for the return values and out values,

- marshal the return values and out values into the buffer, then
- return successfully (i.e.: NOERROR) from Invoke().

Errors may of course occur at various places in this process.[73] Such errors will cause the stub to return an error from Invoke() rather than NOERROR. In cases where such an error code is returned, it is the stub's responsibility to have cleaned up any data and other resources allocated by the unmarshaling and marshaling processes or returned as out values from the server object. However, the stub is *not* responsible for invoking FreeBuffer() to free the actual marshaling buffer (i.e.: it is illegal for the stub to do so); rather, on error return from Invoke() the caller of Invoke() will ignore pvBuffer, and will also free it if non-NULL. Having made that general statement as to the exit conditions of Invoke(), let us examine its operation in greater detail.

If the stub cannot deal with the indicated dataRepresentation, it is to return RPC_E_SERVER_INVALIDDATAREP. If it understands the data representation, the stub is to then unmarshal the arguments from the buffer provided in pMessage->pvBuffer, the size of which is passed in pMessage->cbBuffer. If the argument data cannot be completely unmarshaled, the server is to free any partially unmarshaled data, then return RPC_E_SERVER_CANTUNMARSHALDATA from Invoke().

If the data is successfully completely unmarshaled, then the interface stub is to invoke the designated method in the designated interface on the server object. Notice that the incoming pvBuffer memory buffer is at this time still valid, and that therefore the stub may if it wishes and if appropriate for the argument and data representations in question pass to the server object pointers which point directly into this buffer. The memory allocation and data copying that is thus avoided can at times be a significant performance optimization.

Once the invocation of the server object returns, the stub is to marshal the return value and out parameters returned from the server back to the client. It does so irrespective of whether the server object invocation returned an error or success code; that is, the stub marshals back to the client whatever the server object returned.[74] The stub gets a reply buffer into which to do this marshaling by calling pChannel->GetBuffer(), passing in the pMessage structure that it received in Invoke(). Before calling GetBuffer(), the stub is to set the cbBuffer member to the size that it requires for the to-be-allocated reply buffer. Zero is explicitly a legal value for cbBuffer, and the stub must *always* call GetBuffer() (more precisely, to be clear about the error case: the stub must always call GetBuffer() if the server object method has actually been invoked)[75] to allocate a reply buffer, even if the stub itself does not require one (such as would be the case if for a void-returning function with no out parameters). The stub must also set dataRepresentation as appropriate for the standard by which it intends to marshal the returning values (or would marshal them if there were some).[76] Aside from cbBuffer, dataRepresentation and possibly the contents of the bytes inside the memory buffer, on entry to GetBuffer() no other data accessible from pMessage may be different than they were on entry to Invoke().

Before it allocates a reply buffer, the call to GetBuffer() has the side effect of freeing the memory buffer to which pvBuffer presently points. Thus, the act by the interface stub of allocating a reply buffer for the return values necessarily terminates access by the stub to the incoming marshaled arguments.

If GetBuffer() successfully allocates a reply buffer (see GetBuffer() for a description of how the stub determines this), then the stub is to marshal the return value and returned out parameters into the buffer according to the rules of the transfer syntax. Once this is complete, the stub is to set the cbBuffer member to the number of bytes it actually marshaled (if it marshaled nothing, then it must explicitly set this to zero (but see also GetBuffer())), and then return NOERROR from Invoke().

If an error occurs during the unmarshaling of the incoming arguments or the marshaling of the return values, then the interface stub is responsible for correctly freeing any resources consumed by the marshaled data. See in particular CoReleaseMarshalData(). See also the discussion of this topic in IRpcChannelBuffer::-SendRecieve().

---

[73] Be careful with the terminology here: we are not talking at all about what values are returned from the invocation of the server object, but rather only about errors that occur in the unmarshaling and marshaling process itself.

[74] However, debugging versions of the stub may if they wish to at this time check that certain details of the contract of the interface have been upheld. A common example of this is checking that on error return from the server allocated out-values are explicitly NULLed, a policy which is common to many interfaces. This is simply in the interest of improving the debug capabilities. It is illegal, however, to do such things in non-debug versions of stubs; they must always simply marshal back whatever the server returned.

[75] This policy exists in order to enable behind-the-scenes things such as debugging support to function in all cases.

[76] Presently, this is only significant if NDR transfer syntax is in use. In NDR, it is explicitly the case that the return values may be marshaled using a different data representation than was used for the incoming arguments.

Argument	Type	Description
pMessage	RPCOLEMESSAGE *	channel-allocated message structure.
pChannel	IRpcChannelBuffer *	the channel to use for buffer management, etc.
return value	HRESULT	S_OK, RPC_E_SERVER_INVALIDDATAREP, RPC_E_SERVER_-CANTUNMARSHALDATA, RPC_E_SERVER_CANTMARSHALDATA

.4 IRpcStubBuffer::IsIIDSupported

IRpcStubBuffer* IRpcStubBuffer::IsIIDSupported(iid)

Answer whether this stub is designed to handle the unmarshaling of the indicated interface.

If the stub buffer supports the specified IID, then it should return an appropriate IRpcStubBuffer* for that interface. Otherwise, the it should return NULL.

When presented with the need to remote a new IID on a given object, the RPC runtime typically calls this function on all the presently-connected interface stubs in an attempt to locate one that can handle the marshaling for the request before it goes to the trouble of creating a new stub.

As in IPSFactoryBuffer::CreateStub(), if this stub is presently connected to a server object, then not only must this function verify that the *stub* can handle the requested interface id, but it must also verify (using QueryInterface()) that the connected server object in fact supports the indicated interface (depending on the IID and previous interface servicing requests, it may have already done so).

A common special case is the following: interface stubs which are designed to only support one interface id (as most are designed to do) can simply check if iid designates the one interface that they handle. If not, return false. Otherwise, then if connected check that the server object supports the interface. Otherwise return true.

Argument	Type	Description
iid	REFIID	the interface that the caller wishes to know if the stub can handle. iid is never to be IID_IUnknown.
return value	IRpcStubBuffer*	see above.

.5 IRpcStubBuffer::CountRefs

ULONG IRpcStub::CountRefs()

Return the total number of references that this stub interface instance has on the server object.

Argument	Type	Description
return value	ULONG	the number of such references.

.6 IRpcStubBuffer::DebugServerQueryInterface

HRESULT IRpcStubBuffer::DebugServerQueryInterface(ppv)

This function exists in order to facilitate the support of debuggers which wish to provide transparency when single-stepping, etc., across remote invocations on objects. As such, the semantics of this function are a little strange in order to avoid the unnecessarily disturbing the state of the actual server object.

If the stub is not presently connected then set *ppv to NULL (per the usual error-case convention) and return E_UNEXPECTED. If connected but this stub does not support the indicated interface (in the sense expressed in IsIIDSupported()), then (set *ppv to NULL and) return E_NOINTERFACE instead.

Otherwise, return the interface pointer on the connected server object which would be used by an immediate subsequent invocation of Invoke() on this interface stub (see the discussion on page 130 regarding how interface stubs implicitly know the IID which they are servicing). DebugServerQueryInterface() is analogous to invoking QueryInterface() on the server itself with the important difference that the caller will later call DebugServerRelease() to indicate that he is done with the pointer instead of releasing the returned pointer himself. It is required that DebugServerRelease() be called before the interface stub itself is destroyed or, in fact, before it is disconnected.

In the vast majority of interface stub implementations, DebugServerQueryInterface() can therefore be implemented simply by returning an internal state variable inside the interface stub itself without doing an AddRef() on the server or otherwise running any code in the actual server object. In such implementations, DebugServerRelease() will be a completely empty no-op. The other rational implementation is one where DebugServerQueryInterface() does a QueryInterface() on the server object and DebugServerRelease() does a corresponding Release(), but as this actually runs server code, the former implementation is highly preferred if at all achievable.

Argument	Type	Description
ppv	void**	the place at which the interface pointer is to be returned.
return value	HRESULT	S_OK, E_NOINTERFACE, E_UNEXPECTED

.7 IRpcStubBuffer::DebugServerRelease

void IRpcStubBuffer::DebugServerRelease(pv)

Indicate that an interface pointer returned previously from DebugServerQueryInterface() is no longer needed by the caller. In most implementations, DebugServerRelease() is a completely empty no-op; see the description of DebugServerQueryInterface() for details.

8 Marshaling - Related API Functions

The following functions are related to the process of remoting interface pointers and to marshaling in general.

```
HRESULT     CoMarshalInterface(pstm, riid, pUnk, dwDestContext, pvDestContext, mshlflags);
HRESULT     CoUnmarshalInterface(pstm, iid, ppv);
HRESULT     CoDisconnectObject(pUnkInterface, dwReserved);
HRESULT     CoReleaseMarshalData(pstm);
HRESULT     CoGetStandardMarshal(iid, pUnkObject, dwDestContext, pvDestContext, mshlflags, ppmarshal);

typedef enum tagMSHLFLAGS {
    MSHLFLAGS_NORMAL        = 0,
    MSHLFLAGS_TABLESTRONG   = 1,
    MSHLFLAGS_TABLEWEAK     = 2,
    } MSHLFLAGS;
```

.1 CoMarshalInterface

HRESULT CoMarshalInterface(pstm, riid, pUnk, dwDestContext, pvDestContext, mshlflags)

Marshal the interface riid on the object on which pUnk is an IUnknown* into the given stream in such a way as it can be reconstituted in the destination using CoUnmarshalInterface().[77] This the root level function by which an interface pointer can be marshaled into a stream. It carries out the test for custom marshaling, using it if present, and carries out standard marshaling if not. This function is normally only called by code in interface proxies or interface stubs that wish to marshal an interface pointer parameter, though it will sometimes also be called by objects which support custom marshaling.

riid indicates the interface on the object which is to be marshaled. It is specifically *not* the case that pUnk need actually be of interface riid; this function will QueryInterface from pUnk to determine the actual interface pointer to be marshaled.

dwDestContext is a bit field which identifies the execution context relative to the current context in which the unmarshaling will be done. Different marshaling might be done, for example, depending on whether the unmarshal happens on the same workstation vs. on a different workstation on the network; an object could choose to do custom marshaling in one case but not the other. The legal values for dwDestContext are taken from the enumeration MSHCTX, which presently contains the following values.

[77] That is, the mechanism for unmarshaling a marshaled interface pointer is the *same* irrespective of whether the marshaling was done using custom or standard marshaling.


```
typedef enum tagMSHCTX {
    MSHCTX_NOSHAREDMEM          = 1,
    MSHCTX_DIFFERENTMACHINE     = 2,
    MSHCTX_SAMEPROCESS          = 4,
    } MSHCTX;
```

These flags have the following meanings.

Value	Description
MSHCTX_NOSHAREDMEM	The unmarshaling context does not have shared memory access with the marshaling context.
MSHCTX_DIFFERENTMACHINE	If this flag is set, then it cannot be assumed that this marshaling is being carried out to the same machine as that on which the marshaling is being done. The unmarshaling context is (very probably) on a computer with a different set of installed applications / components than the marshaling context (i.e.: is on a different computer). This is significant in that the marshaling cannot in this case assume that it knows whether a certain piece of application code is installed remotely.
MSHCTX_SAMEPROCESS	The interface is being marshaled to another apartment within the same process in which it is being unmarshaled.

In the future, more MSHCTX flags may be defined; recall that this is a bit field.

pvDestContext is a parameter that optionally supplies additional information about the destination of the marshaling. If non-NULL, then it is a pointer to a structure of the following form.

```
typedef struct MSHCTXDATA {
    ULONG               cbStruct;
    IRpcChannelBuffer*  pChannel;
    } MSHCTXDATA;
```

The members in this structure have the following meanings:

Value	Description
cbStruct	The size of the MSHCTXDATA structure in bytes.
pChannel	The channel object involved in the marshaling process.

pvDestContext may legally be NULL, in which case such data is not provided.

mslflags indicates the purpose for which the marshal is taking place, as was discussed in an earlier part of this document. Values for this parameter are taken from the enumeration MSHLFLAGS, and have the following interpretation.

Value	Description
MSHLFLAGS_NORMAL	The marshaling is occurring because of the normal case of passing an interface from one process to another. The marshaled-data-packet that results from the call will be transported to the other process, where it will be unmarshaled (see CoUnmarshalInterface). With this flag, the marshaled data packet will be unmarshaled either one or zero times. CoReleaseMarshalData is always (eventually) called to free the data packet.
MSHLFLAGS_TABLESTRONG	The marshaling is occurring because the data-packet is to be stored in a globally-accessible table from which it is to be unmarshaled zero, one, or more times. Further, the presence of the data-packet in the table is to count as a reference on the marshaled interface. When removed from the table, it is the responsibility of the table implementor to call CoReleaseMarshalData on the data-packet.
MSHLFLAGS_TABLEWEAK	The marshaling is occurring because the data-packet is to be stored in a globally-accessible table from which it is to be unmarshaled zero, one, or more times. However, the presence of the data-packet in the table is *not* to count as a reference on the marshaled interface. Destruction of the data-packet is as in the MSHLFLAGS_TABLESTRONG case.

A consequence of this design is that the marshaled data packet will want to store the value of mshlflags in the marshaled data so as to be able to do the right thing at unmarshal time.

Argument	Type	Description
pstm	IStream *	the stream onto which the object should be marshaled. The stream passed to this function must be dynamically growable.
riid	REFIID	the interface that we wish to marshal.
pUnk	IUnknown *	the object on which we wish to marshal the interface riid.
dwDestContext	DWORD	the destination context in which the unmarshaling will occur.
pvDestContext	void*	as described above.
mshlflags	DWORD	the reason that the marshaling is taking place.
return value	HRESULT	S_OK, STG_E_MEDIUMFULL, E_NOINTERFACE, E_FAIL

.2 CoUnmarshalInterface

HRESULT CoUnmarshalInterface(pstm, iid, ppv)

Unmarshal from the given stream an object previously marshaled with CoMarshalInterface.

Argument	Type	Description
pstm	IStream *	the stream from which the object should be unmarshaled.
iid	REFIID	the interface with which we wish to talk to the reconstituted object.
ppv	void **	the place in which we should return the interface pointer.
return value	HRESULT	S_OK, E_FAIL, E_NOINTERFACE

.3 CoDisconnectObject

HRESULT CoDisconnectObject(pUnkInterface, dwReserved)

This function severs any extant Remote Procedure Call connections that are being maintained on behalf of all the interface pointers on this object. This is a very rude operation, and is not to be used in the normal course of processing; clients of interfaces should use IUnknown::Release() instead. In effect, this function is a privileged operation, which should only be invoked by the process in which the object actually is managed.

The primary purpose of this operation is to give an application process certain and definite control over remoting connections to other processes that may have been made from objects managed by the process. If

the application process wishes to exit, then we do not want it to be the case that the extant reference counts from clients of the application's objects in fact keeps the process alive. When the application process wishes to exit, it should inform the extant clients of its objects[78] that the objects are going away. Having so informed its clients, the process can then call this function for each of the object that it manages, even without waiting for a confirmation from each client. Having thus released resources maintained by the remoting connections, the application process can exit safely and cleanly. In effect, CoDisconnectObject() causes a controlled crash of the remoting connections to the object. It is also (one of) the triggers by which a client's subsequent IRpcChannel::IsConnected() call may return false.

For illustration, contrast this with the situation with Microsoft's elderly Dynamic Data Exchange (DDE) desktop application integration protocol. If it has extant DDE connections, an application is required to send a DDE Terminate message before exiting, and it is *also* responsible for waiting around for an acknowledgment from each client before it can actually exit. Thus, if the client process has crashed, the application process will wait around forever. Because of this, with DDE there simply is no way for an application process to reliably and robustly terminate itself. Using CoDisconnectObject(), we avoid this sort of situation.

Argument	Type	Description
punkInterface	IUnknown *	the object that we wish to disconnect. May be any interface on the object which is polymorphic with IUnknown*, not necessarily the exact interface returned by QueryInterface(IID_IUnknown...).
dwReserved	DWORD	reserved for future use; must be zero.
return value	HRESULT	S_OK, E_FAIL

.4 CoReleaseMarshalData

HRESULT CoReleaseMarshalData(pstm)

This helper function destroys a previously marshaled data packet. This function must always be called in order to destroy data packets. Examples of when this occurs include:

1. an internal error during an RPC invocation prevented the UnmarshalInterface() operation from being attempted.
2. a marshaled-data-packet was removed from a global table.
3. following a successful, normal, unmarshal call.

This function works as should be expected: the class id is obtained from the stream; an instance is created; IMarshal is obtained from that instance; then IMarshal::ReleaseMarshalData() is invoked.

Note for clarity: CoReleaseMarshalData() is not to be called following a normal, successful CoUnmarshalInterface(), as the latter function does this automatically for MSHLFLAGS_NORMAL. However, clients that use IMarshal interface directly, rather than simply going through the functions CoMarshal/UnmarshalInterface(), etc., must of course themselves always call IMarshal::ReleaseMarshalData() after calling IMarshal::UnmarshalInterface().

Argument	Type	Description
pstm	IStream*	a pointer to a stream that contains the data packet which is to be destroyed.
return value	HRESULT	S_OK, E_FAIL

.5 CoGetStandardMarshal

HRESULT CoGetStandardMarshal(iid, pUnkObject, dwDestContext, pvDestContext, mshlflags, ppmarshal)

Return an IMarshal instance that knows how to do the standard marshaling and unmarshaling in order to create a proxy in the indicated destination context. Custom marshaling implementations should delegate to the marshaler here returned for destination contexts that they do not fully understand or for which they

[78] using a higher-level notification scheme appropriate for the semantics of the particular connection. An example of this is OLE 2.0 is broadcasting IAdviseSink::OnClose() to connected links.

choose not to take special action. The standard marshaler is also used in the case that the object being marshaled does not support custom marshaling.

Argument	Type	Description
iid	REFIID	the interface id we would like to marshal.
pUnkObject	IUnknown*	the object that we wish to marshal. It is specifically *not* the case that this interface is known to be of shape iid; rather, it can be any interface on the object which conforms to IUnknown. The standard marshaler will internally do a QueryInterface().
dwDestContext	DWORD	the destination context in which the unmarshaling will occur.
pvDestContext	void *	associated with the destination context.
mshlflags	DWORD	the marshal flags for the marshaling operation.
ppmarshal	IMarshal **	the place at which the standard marshaler should be returned.
return value	HRESULT	S_OK, E_FAIL

.6 CoGetMarshalSizeMax

HRESULT CoGetMarshalSizeMax(riid, pUnk, dwDestContext, pvDestContext, mshlfags, pulSize)

Return the number of bytes needed to marshal the given interface on the given object. On successful exit, the value pointed to by *pulSize will have been *incremented* by the number of bytes required.

This function is useful to custom marshaling implementations which themselves internally marshal interface pointers as part of their state.

Argument	Type	Description
riid	REFIID	the interface on the object which is to be marshaled.
pUnk	IUnknown*	an IUnknown (any old one) on the object.
dwDestContext	DWORD	the context into which the object is to be marshaled.
pvDestContext	void *	the context into which the object is to be marshaled.
mshlflags	DWORD	the marshal flags for the marshaling operation
pulSize	ULONG *	the place at which the required size is to be returned.
return value	HRESULT	S_OK, E_NOINTERFACE, E_OUTOFMEMORY, E_UNEXPECTED

9 IMarshal interface

IMarshal interface is the mechanism by which an object is custom-marshaled. IMarshal is defined as follows:

```
interface IMarshal : IUnknown {
    HRESULT    GetUnmarshalClass(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pclsid);
    HRESULT    GetMarshalSizeMax(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pcb);
    HRESULT    MarshalInterface(pstm, iid, pvInterface, dwDestContext, pvDestContext, mshlflags);
    HRESULT    UnmarshalInterface(pstm, iid, ppvInterface);
    HRESULT    DisconnectObject(dwReserved);
    HRESULT    ReleaseMarshalData(pstm);
    };
```

The process of custom marshaling an interface pointer involves two steps, with an optional third:

1. The code doing the marshaling calls IMarshal::GetUnmarshalClass(). This returns the class id that will be used to create an uninitialized proxy object in the unmarshaling context.
2. (optional) The marshaler calls IMarshal::GetMarshalSizeMax() to learn an upper bound on the amount of memory that will be required by the object to do the marshaling.
3. The marshaler calls IMarshal::MarshalInterface() to carry out the marshaling.

The class id and the bits that were marshaled into the stream are then conveyed by appropriate means to the destination, where they are unmarshaled. Unmarshaling involves the following essential steps:

1. Load the class object that corresponds to the class that the server said to use in GetUnmarshalClass().

```
IClassFactory * pcf;
CoGetClassObject(clsid, CLSCTX_INPROCSERVER, IID_IClassFactory, &pcf);
```

2. Instantiate the class, asking for IMarshal interface:

```
IMarshal * proxy;
pcf->CreateInstance(NULL, IID_IMarshal, &proxy);
```

3. Initialize the proxy with IMarshal::UnmarshalInterface() using a copy of the bits that were originally produced by IMarshal::MarshalInterface() and asking for the interface that was originally marshaled.

```
IOriginal * pobj;
proxy->UnmarshalInterface(pstm, IID_Original, &pboj);
proxy->Release();
pcf->Release();
```

The object proxy is now ready for use.

.1 IMarshal::GetUnmarshalClass

HRESULT IMarshal::GetUnmarshalClass(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pclsid)

Answer the class that should be used in the unmarshaling process to create an uninitialized object proxy.

dwDestContext is described in the API function CoMarshalInterface. The implementation of GetUnmarshalClass may wish for some destination contexts for which it takes no special action to delegate to the standard marshaling implementation, which is available through CoGetStandardMarshal. In addition, this delegation should *always* be done if the dwDestContext parameter contains any flags that the GetUnmarshalClass does not fully understand; it is by this means that we can extend the richness of destination contexts in the future. For example, in the future, one of these bits will likely be defined to indicate that the destination of the marshaling is across the network.

If the caller already has in hand the iid interface identified as being marshaled, he should pass the interface pointer through pvInterface. If he does not have this interface already, then he should pass NULL. This pointer can sometimes, though rarely, be used in order to determine the appropriate unmarshal class. If the IMarshal implementation really needs it, in can always QueryInterface on itself to retrieve the interface pointer; we optionally pass it here only to improve efficiency.

Argument	Type	Description
iid	REFIID	the interface on this object that we are going to marshal.
pvInterface	void *	the actual pointer that will be marshaled. May be NULL.
dwDestContext	DWORD	the destination context relative to the current context in which the unmarshaling will be done.
pvDestContext	void*	non-NULL for some dwDestContext values.
mshlflags	DWORD	as in CoMarshalInterface().
pclsid	CLSID *	the class to be used in the unmarshaling process.
return value	HRESULT	S_OK, E_FAIL, E_NOINTERFACE, E_UNEXPECTED

.2 IMarshal::MarshalInterface

HRESULT IMarshal::MarshalInterface(pstm, iid, pvInterface, dwDestContext, pvDestContext, mshlflags)

Marshal a reference to the interface iid of this object into the given stream. The interface actually marshaled is the one that would be returned by this->QueryInterface(iid, ...). Once the contents of this stream are conveyed to the destination by whatever means, the interface reference can be reconstituted by instantiating with IMarshal interface the class here retrievable with GetUnmarshalClass and then calling IMarshal::UnmarshalInterface. The implementation of IMarshal::MarshalInterface writes in the stream any data required for initialization of this proxy.

If the caller already has in hand the iid interface identified as being marshaled, he should pass the interface pointer through pvInterface. If he does not have this interface already, then he should pass NULL; the IMarshal implementation will QueryInterface on itself to retrieve the interface pointer.

On exit from this function, the seek pointer in the stream must be positioned immediately after the last byte of data written to the stream.

Argument	Type	Description
pstm	IStream *	the stream onto which the object should be marshaled.
iid	REFIID	the interface of this object that we wish to marshal.
pvInterface	void *	the actual pointer that will be marshaled. May be NULL.
dwDestContext	DWORD	as in CoMarshalInterface().
pvDestContext	void *	as in CoMarshalInterface().
mshlflags	DWORD	as in CoMarshalInterface().
return value	HRESULT	S_OK, E_FAIL, E_NOINTERFACE, STG_E_MEDIUMFULL, E_UNEXPECTED

.3 IMarshal::GetMarshalSizeMax

HRESULT IMarshal::GetMarshalSizeMax(iid, pvInterface, dwDestContext, pvDestContext, mshlflags, pcb)

Return an upper bound on the amount of data that would be written into the marshaling stream in an IMarshal::MarshalInterface() stream. The value returned must be an upper bound in the sense that it must be the case that a subsequent call to MarshalInterface() in fact require no more than the indicated number of bytes of marshaled data.

Callers can optionally use the returned upper bound to pre-allocate stream buffers used in the marshaling process. Note that when IMarshal::MarshalInterface() is ultimately called, the IMarshal implementation cannot rely on the caller actually having called GetMarshalSizeMax() beforehand; it must still be wary of STG_E_MEDIUMFULL errors returned by the stream.

The value returned by this function is guaranteed by the callee to be a conservative estimate of the amount of data needed to marshal the object; it is valid so long as the object instance is alive. Violation of this can be treated as a catastrophic error. To repeat for emphasis: an object *must* return a reasonable maximum size needed for marshaling: callers have the option of allocating a fixed-size marshaling buffer.

Argument	Type	Description
iid	REFIID	the interface of this object that we wish to marshal.
pvInterface	void *	the actual pointer that will be marshaled. May be NULL.
dwDestContext	DWORD	as in CoMarshalInterface().
pvDestContext	void *	as in CoMarshalInterface().
mshlflags	DWORD	as in CoMarshalInterface().
pcb	ULONG *	the place at which the maximum marshal size should be returned. A return of zero indicates “unknown maximum size.”
return value	HRESULT	S_OK, E_FAIL, E_NOINTERFACE, E_UNEXPECTED

.4 IMarshal::UnmarshalInterface

HRESULT IMarshal::UnmarshalInterface(pstm, iid, ppvInterface)

This is called as part of the unmarshaling process in order to initialize a newly created proxy; see the above sketch of the unmarshaling process for more details.

iid indicates the interface that the caller in fact would like to retrieve from this object; this interface instance is returned through ppvInterface. In order to support this, UnmarshalInterface will often merely do a QueryInterface(iid, ppvInterface) on itself immediately before returning, though it is free to create a different object (an object with a different identity) if it wishes.

On successful exit from this function, the seek pointer must be positioned immediately after the data read from the stream. On error exit, the seek pointer should still be in this location: even in the face of an error, the stream should be positioned as if the unmarshal were successful.

See also CoReleaseMarshalData.

Argument	Type	Description
pstm	IStream *	the stream from which the interface should be unmarshaled.
iid	REFIID	the interface that the caller ultimately wants from the object.
ppvInterface	void **	the place at which the interface the caller wants is to be returned.
return value	HRESULT	S_OK, E_FAIL, E_NOINTERFACE, E_UNEXPECTED

.5 IMarshal::Disconnect

HRESULT IMarshal::DisconnectObject(dwReserved)

This function is called by the implementation of CoDisconnectObject in the event that the object attempting to be disconnected in fact supports custom marshaling. This is completely analogous to how CoMarshalInterface defers to IMarshal::MarshalInterface in if the object supports IMarshal.

Argument	Type	Description
dwReserved	DWORD	as in CoDisconnectObject().
return value	HRESULT	as in CoDisconnectObject().

.6 IMarshal::ReleaseMarshalData

HRESULT IMarshal::ReleaseMarshalData(pstm)

This function is called by CoReleaseMarshalData() in order to actually carry out the destruction of a marshaled-data-packet. See that function for more details.

Note that whereas the IMarshal methods

GetUmarshalClass
GetMarshalSizeMax
MarshalInterface
Disconnect

are always called on the IMarshal interface instance in the originating side (server side), the method

UnmarshalInterface

is called on the receiving (client) side. (This should be no surprise.) However, the function

ReleaseMarshalData

(when needed) will be called on the receiving (client) side if the appropriate IMarshal instance can be successfully created there; otherwise, it is invoked on the originating (server) side.

Argument	Type	Description
pstm	IStream*	as in CoReleaseMarshalData().
return value	HRESULT	as in CoReleaseMarshalData().

10 IStdMarshalInfo interface

IStdMarshalInfo is implemented by objects wishing to support handler marshaling in their remote client process. This is common, for example, for OLE 2 compound document embedded objects which for example support client-side drawing related interfaces using the IViewObject interface, an interface which is not (usually) supported on the actual embedding itself.

```
interface IStdMarshalInfo : IUnknown {
    HRESULT     GetClassForHandler(DWORD dwDestContext, void* pvDestContext, CLISD* pclsid);
    };
```

.1 IStdMarshalInfo::GetClassForHandler

HRESULT IStdMarshalInfo::GetClassForHandler(dwDestContext, dwDestContext, pclsid)

Return the CLSID whose handler is to be used in the remote client process.

Argument	Type	Description
dwDestContext	DWORD	As in CoMarshalInterface.
pvDestContext	void*	As in CoMarshalInterface.
pclsid	CLSID*	The place at which the requested CLSID is returned.

Return Value	Meaning
S_OK	Success. The required CLSID is returned.
E_UNEXPECTED	An unspecified error occurred.

11 Support for Remote Debugging

The COM Library and the COM Network Protocol provide support for debugging engines on the client and the server side of a remote COM invocation to cooperate in allowing the overall application to be debugged. This section describes the runtime infrastructure provided by the Microsoft Windows implementation of the COM Library by which that is accomplished; other implementations will provide similar infrastructures, though in practice the details of such support will be highly sensitive to the mechanisms by which debugging engines are supported on the given platform. This section also specifies the standard data formats transmitted between client and server by which this cooperation is carried out.

The following a brief example of the sort of debugging session scenario which can be supported with this infrastructure.

Suppose the programmer is debugging an application with is an OLE document container, and that the application is presently stopped in the debugger at a point in the code where the container is about to invoke a method in some interface on one of its contained objects, the implementation of which happens to be in another executable. That is, the pointer that the container has in hand actually points to an occurrence of part of the remoting infrastructure known as an "interface proxy" (see above). Interface proxies and the rest of the remoting infrastructure are not (normally) part of the programmer's concern when debugging client and server applications, as the whole *raison d'[illegible]re* of the RPC infrastructure is to be *transparent*, is to make remote object invocations appear to be local ones. Unless the programmer is debugging the remoting infrastructure himself, this should apply to debugging as well.

This perspective leads to some of the following scenarios that need to be supportable by the debugger. If the programmer Single Steps into the function invocation, then the debugger should next stop just inside the real implementation of the remote server object, having transparently passed through the RPC infrastructure. (Notice that before the Step command is executed, the remote process may not presently have the debugger[79] attached to it, and so the act of doing the step may need to cause the debugger to attach itself.) The programmer will now be able to step line by line through the server's function. When he steps past the closing brace of the function, he should wind up back in the debugger of the client process immediately after the function call.

A similar scenario is one where we skip the incoming single step but instead, out of the blue, hit a breakpoint in the server, then start single stepping. This, too, should single step over the end of the server function back into the client process. The twist is that this time, the *client* debugger may not presently be running, and therefore may need to be started.

.1 Implementation

The ability for debuggers to support scenarios such as these is provided by hooks in the client and server side RPC infrastructure. If requested by the debugger, at certain important times, these hooks inform the debugger of the fact that a transmission of a remote call about to be made or that transmission of return values is about to occur. That is, when the COM Library is about to make or return from a call to an object, it notifies the debugger of what is happening, so that the debugger can take any special actions it desires.

[79] More precisely, it may not have *a* debugger attached to it: depending on the debugger's implementation and the relative location of the two processes with respect to machine boundaries, a new debugger instance may or may not need to be created. The main point is that the process wasn't being debugged.

.1 DllDebugObjectRPCHook

BOOL WINAPI DllDebugObjectRPCHook(BOOL fTrace, LPORPC_INIT_ARGS lpOrpcInitArgs)

This function is to be exported by name from one or more DLLs that wish to be informed when from the user's point of view that debugging is engaged. Debuggers will should call this function to inform each of their loaded DLLs that export this function as to whether they are presently being debugged or not. When the debugger wants to enable debugging, it calls DllDebugObjectRpcHook with fTrace=TRUE and when it wants to disable it, it calls DllDebugObjectRpcHook with fTrace=FALSE. When enabled, debugging support such as the tracing described herein should be enabled.

Certain of the COM Library DLLs, for example, implement this function. When debugging is enabled, they turn on what is here called COM remote debugging, and which is the focus of this section.

The second argument points to an ORPC_INIT_ARGS structure whose definition is given below. The pvPSN member is used only on the Macintosh, where the calling debugger is required in this field to pass the process serial number of the debuggee's process. On other systems pvPSN should be NULL.

The lpIntfOrpcDebug member is a pointer to an interface. This is used by in-process debuggers and is discussed in more detail later. Debuggers that are neither in-process debuggers nor are Macintosh debuggers should pass NULL for lpIntfOrpcDebug.

```
typedef struct ORPC_INIT_ARGS {
    IOrpcDebugNotify __RPC_FAR *  lpIntfOrpcDebug;
    void *                        pvPSN;          // contains ptr to Process Serial No. for Mac COM debugging.
    DWORD                         dwReserved1;    // For future use, must be 0.
    DWORD                         dwReserved2;    // For future use, must be 0.
    } ORPC_INIT_ARGS;

typedef ORPC_INIT_ARGS  __RPC_FAR * LPORPC_INIT_ARGS;

interface IOrpcDebugNotify : IUnknown {
    VOID    ClientGetBufferSize(LPORPC_DBG_ALL);
    VOID    ClientFillBuffer(LPORPC_DBG_ALL);
    VOID    ClientNotify(LPORPC_DBG_ALL);
    VOID    ServerNotify(LPORPC_DBG_ALL);
    VOID    ServerGetBufferSize(LPORPC_DBG_ALL);
    VOID    ServerFillBuffer(LPORPC_DBG_ALL);
    };
```

As one would expect, a debugger calls DllDebugObjectRPCHook within the context (that is, within the process) of the relevant debuggee. Thus, the implementation of this function most often will merely store the arguments in global DLL-specific state.

Further, as this function is called from the debugger, the function can be called when the DLL in which it is implemented is in pretty well any state; no synchronization with other internal DLL state can be relied upon. Thus, it is recommended that the implementation of this function indeed do nothing *more* than set internal global variables.

Argument	Type	Description
fTrace	BOOL	TRUE if debugging is enabled, FALSE otherwise
lpOrpcInitArgs	LPORPC_INIT_ARGS	typically NULL; see comments above for MAC COM debuggers or in-process debuggers.
return value	BOOL	TRUE if the function was successful (the DLL understood and executed the request), FALSE otherwise

.2 Architectural Overview

When COM remote debugging is enabled, there are a total of six notifications that occur in the round-trip of one COM RPC call: three on the client side and three on the server side. The overall sequence of events is as follows.

Suppose the client has an interface pointer pFoo of type IFoo* which happens to be a proxy for another object in a remote server process.

```
interface IFoo : IUnknown {
    HRESULT Func();
```

```
    };
IFoo *pFoo;
```

When the client invokes pFoo->Func(), it executes code in the interface proxy. This code is responsible for marshaling the arguments into a buffer, calling the server, and unmarshaling the return values. To do so, it draws on the services of an IRpcChannelBuffer instance with which it was initialized by the COM Library.

To get the buffer, the interface proxy calls IRpcChannelBuffer::GetBuffer(), passing in (among other things) the requested size for the buffer. Before actually allocating the buffer, the GetBuffer() implementation (normally[80]) checks to see if debugging is enabled per DllDebugObjectRPCHook(). If so, then the channel calls DebugORPCClientGetBufferSize() (see below for details) to inform the debugger that an COM RPC call is about to take place and to ask the debugger how many bytes of information *it* would like to transmit to the remote server debugger. The channel then, unbeknownst to the interface proxy, allocates a buffer with this many additional bytes in it.

The interface proxy marshals the incoming arguments in the usual way into the buffer that it received, then calls IRpcChannelBuffer::SendReceive(). Immediately on function entry, the channel again checks to see if debugging is enabled. If so, then it calls DebugORPCClientFillBuffer() passing in the pointer to (the debugger's part of) the marshaling buffer. The debugger will write some information into the buffer, but this need be of no concern to the channel implementation other than that it is to ferry the contents of the buffer to the server debugger. Once DebugORPCClientFillBuffer() returns, the channel implementation of SendReceive() proceeds as in the normal case.

We now switch context in our explanation here to the server-side RPC channel. Suppose that it has received an incoming call request and has done what it normally does just up to the point where it is about to call IRpcStubBuffer::Invoke(), which when will cause the arguments to be unmarshaled, etc. Just before calling Invoke(), if there was any debugger information (i.e.: it exists in the incoming request and is of non-zero size) in the incoming request *or* if debugging is presently *already* enabled per DllDebugObjectRPCHook() (irrespective of the presence or size of the debug info), then the channel is to call DebugORPCServerNotify().[81] The act of calling this function may in fact *start* a new debugger if needed and attach it to this (the server) process; however, this need not be of concern to the channel implementation. Having made the request, the channel proceeds to call Invoke() as in the normal case.

The implementation of Invoke() will unmarshal the incoming arguments, then call the appropriate method on the server object. When the server object returns, Invoke() marshals the return values for transmission back to the client. As on the client side, the marshaling process begins by calling IRpcChannelBuffer::GetBuffer() to get a marshaling buffer. As on the client side, the server side channel GetBuffer() implementation when being debugged (per the present setting of DllDebugObjectRPCHook(), *not* per the presence of the incoming debug info) asks the debugger how many bytes it wishes to transmit back to the client debugger. The channel allocates the buffer accordingly and returns it to the Invoke() implementation who marshals the return values into it, then returns to its caller.

The caller of IRpcStubBuffer::Invoke() then checks to see if he is presently being debugged. If so, then he at this time calls DebugORPCServerFillBuffer(), passing in the pointer to the debug-buffer that was allocated in the (last, should there erroneously be more than one) call to GetBuffer() made inside Invoke(); should no such call exist, and thus there is no such buffer, NULL is passed.[82] The bytes written into the buffer (if any) by the debugger are ferried to the client side.

We now switch our explanatory context back to the client side. Eventually the client channel either receives a reply from the server containing the marshaled return values (and possibly debug info), receives an error indication from the server RPC infrastructure, or decides to stop waiting. That is, eventually the client channel decides that it is about to return from IRpcChannel::SendReceive(). Immediately before doing so, it checks to see if it is either already presently being debugged *or* if in the reply it received any (non-zero

---

[80] That is, in the channel implementation approach described here, which uses only one memory buffer. Another channel implementation approach would use two separate buffers, one to give back to the interface proxy, and another independent one for the debug information. Such an implementation would only need to call DebugORPCClientGetBufferSize() in its IRpcChannelBuffer::SendReceive() implementation immediately before calling DebugORPCClientFillBuffer(). While perfectly legal, this will not be elaborated further here, though in fact this is the implementation likely to be used in practice, given how the debug data is to be transmitted in the COM Network Protocol. We trust that readers can accommodate our pedagogical style; apologies to those who cannot.

[81] Some control as to whether this is to be actually carried out is provided by the first four bytes of the incoming debug data; see later in this specification.

[82] This is important in error handling cases to allow us to ensure that breakpoints are always cleared correctly.

sized) information from the server debugger. If so, then it calls DebugORPCClientNotify(), passing in the server-debugger's info if it has any; doing so may start and attach the debugger if needed. The channel then returns from SendReceive().

.3 Calling Convention for Notifications

The preceding discussion discussed the COM RPC debugging architecture in terms of six of debugger-notification APIs (DebugORPC...()). However, rather than being actual API-entry points in a a static-linked or dynamically-linked library, these notifications use an somewhat unusual calling convention to communicate with the notification implementations, which are found inside debugger products. This somewhat strange calling convention is used for the following reasons:

- Two of the six notifications need to start and attach the debugger if it is not already attached to the relevant process.
- The convention used transitions into the debugger code with the least possible disturbance of the debuggee's state and executing the minimal amount of debuggee code. This increases robustness of debugging.
- The debugger is necessarily equipped to deal with concurrency issues of other threads executing in the same process. Therefore, it is important to transition to the debugger as fast as possible to avoid inadvertent concurrency problems.

The actual calling convention used is by its nature inherently processor and operating-system specific. On Win32 implementations, the default calling convention for notifications takes the form of a software exception, which is raised by a call to the RaiseException Win32 API:

```
VOID RaiseException(
    DWORD   dwExceptionCode,        // exception code
    DWORD   dwExceptionFlags,       // continuable exception flag
    DWORD   cArguments,             // number of arguments in array
    CONST DWORD *   lpArguments     // address of array of arguments
    );
```

As used here, the arguments to this raised exception call in order are:

- dwExceptionCode: An exception code EXCEPTION_ORPC_DEBUG (0x804F4C45) is used. The debugger should recognize this exception as a special one indicating an COM RPC debug notification.
- dwExceptionFlags: This is zero to indicate a continuable exception.
- cArguments: One
- lpArguments: The array contains one argument. This argument is a pointer to a structure which contains the notification specific information that the COM RPC system passes to the debugger. The definition of this structure ORPC_DBG_ALL is given below. The same structure is used for all the notifications. The structure is just the union of the arguments of the six debugger notification APIs. For a particular notification not all the fields in the structure are meaningful and those that are not relevant have undefined values; details on this are below:

```
typedef struct ORPC_DBG_ALL   {
    BYTE *                  pSignature;
    RPCOLEMESSAGE *         pMessage;
    const IID *             iid;
    void*                   reserved1;
    void*                   reserved2;
    void*                   pInterface;
    IUnknown *              pUnkObject;
    HRESULT                 hresult;
    void *                  pvBuffer;
    ULONG                   cbBuffer;
    ULONG *                 lpcbBuffer;
    void *                  reserved3;
    } ORPC_DBG_ALL;
```

The pSignature member of this structure points to a sequence of bytes which contains:

- a four-byte sanity-check signature of the ASCII characters "MARB" in increasing memory order.[83]
- a 16-byte GUID indicating which notification this is. Each of the six notifications defined here has a different GUID. More notifications and corresponding GUIDs can be defined in the future and be known not to collide with existing notifications.
- a four-byte value which is reserved for future use. This value is NULL currently.

The notifications specified here pass their arguments by filling in the appropriate structure members. See each notification description for details.

Using software exceptions for COM debugging notifications is inconvenient for "in-process" debugging. In-process debuggers can alternately get these notifications via direct calls into the debugger's code. The debugger which wants to be notified by a direct call passes in an IOrpcDebugNotify interface in the LPORPC_-INIT_ARGS argument to DllDebugObjectRPCHook. If this interface pointer is available, COM makes the debug notifications by calling the methods on this interface. The methods all take an LPORPC_DBG_ALL as the only argument. The information passed in this structure is identical to that passed when the notification is done by raising a software exception.

.4 Notifications

What follows is a detailed description of each of the relevant notifications.

Note that in the network case, depending on the notification in question the byte order used may be different than that of the local machine. The byte order, etc., of the incoming data is provided from the dataRep contained the passed RPCOLEMESSAGE structure.

Though each function is documented here for purely historical reasons as if it were in fact a function call, we have seen above that this is not the case. Unless otherwise specified, the name of the argument to the DebugORPC... notification call is the same as the name of the structure member in ORPC_DBG_ALL used to pass it to the debugger. So for example the pMessage argument of the DebugORPCClientGetBufferSize notification is passed to the debugger in the pMessage structure member of ORPC_DBG_ALL. We trust that readers will not be too confused by this, and apologize profusely should this prove not to be the case.

.1 DebugORPCClientGetBufferSize

ULONG DebugORPCClientGetBufferSize(pMessage, iid, reserved, pUnkProxyObject)

Called on the client side in IRpcChannel::GetBuffer().

The GUID for this notification is 9ED14F80-9673-101A-B07B-00DD01113F11

```
GUID __private_to_macro__ = { /* 9ED14F80-9673-101A-B07B-00DD01113F11 */
    0x9ED14F80,
    0x9673,
    0x101A,
    0xB0,
    0x7B,
    {0x00, 0xDD, 0x1, 0x11, 0x3F, 0x11}
  };
```

[83] "MARB" is "Mike Alex Rico Bob," arranged in an order such that it makes a goofy-sounding syllable. Call us whimsical.

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	identification of the method being invoked, etc.
iid	REFIID	contains the IID of the interface being called.
reserved	void *	reserved for future use.
pUnkProxyObject	IUnknown *	an IUnknown (no particular one) on the object involved in this invocation. May legally be NULL, though this reduces debugging functionality. Further, this and like-named parameters must consistently be either NULL or non-NULL in all notifications in a given client side COM RPC implementation.
"return value"	ULONG	the number of bytes that the client debugger wishes to transmit to the server debugger. May legitimately be zero, which indicates that no information need be transmitted. The lpcbBuffer field in the ORPC_DBG_ALL structure holds a pointer to a ULONG. The debugger writes the number of bytes it wants to transmit with the packet in that location.

.2 DebugORPCClientFillBuffer

void DebugORPCClientFillBuffer(pMessage, iid, reserved, pUnkProxyObject, pvBuffer, cbBuffer)

Called on the client side on entry to IRpcChannel::SendReceive(). See the above overview for further details.

The GUID for this notification is DA45F3E0-9673-101A-B07B-00DD01113F11:

```
GUID __private_to_macro__ = { /* DA45F3E0-9673-101A-B07B-00DD01113F11 */
    0xDA45F3E0,
    0x9673,
    0x101A,
    0xB0,
    0x7B,
    {0x00, 0xDD, 0x01, 0x11, 0x3F, 0x11}
  };
```

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	as in DebugORPCClientGetBufferSize().
iid	REFIID	as in DebugORPCClientGetBufferSize().
reserved	void *	as in DebugORPCClientGetBufferSize().
pUnkProxyObject	IUnknown *	as in DebugORPCClientGetBufferSize().
pvBuffer	void *	the debug-data buffer which is to be filled. Is undefined (may or may not be NULL) if cbBuffer is zero.
cbBuffer	ULONG	the size of the data pointed to by pvBuffer.

.3 DebugORPCServerNotify

void DebugORPCServerNotify(pMessage, iid, pChannel, pInterface, pUnkObject, pvBuffer, cbBuffer)

Called on the server side immediately before calling IRpcStubBuffer::Invoke() to inform it that there is an incoming request. Will start the debugger in this process if need be. See the above overview for further details.

The GUID for this notification is 1084FA00-9674-101A-B07B-00DD01113F11:

```
GUID __private_to_macro__ = { /* 1084FA00-9674-101A-B07B-00DD01113F11 */
    0x1084FA00,
    0x9674,
    0x101A,
    0xB0,
    0x7B,
    {0x00, 0xDD, 0x01, 0x11, 0x3F, 0x11}
  };
```

On entry, the members of pMessage are set as follows:

Member Name	Value on entry to Invoke()
reserved members	indeterminate. These members are neither to be read nor to be changed by the callee.
dataRepresentation	this indicates the byte order, etc., of the client debugger
pvBuffer	points to a buffer which contains the marshaled incoming arguments. In the case that there are no such arguments (i.e.: cbBuffer == 0), pvBuffer may be NULL, but will not necessarily be so.
cbBuffer	the size in bytes of the memory buffer to which pvBuffer points. If pvBuffer is NULL, then cbBuffer will be zero (but the converse is not necessarily true, as was mentioned in pvBuffer).
iMethod	the zero-based method number in the interface which is being invoked.
rpcFlags	indeterminate. Neither to be read nor to be changed by the callee.

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	as in IRpcStubBuffer::Invoke().
iid	REFIID	contains the iid of the interface being called.
pChannel	IRpcChannelBuffer*	as in IRpcStubBuffer::Invoke(). The COM RPC channel implementation on the server side.
pInterface	void *	This contains the pointer to the COM interface instance which contains the pointer to the method that will be invoked by this particular remote procedure call. Debuggers can use this information in conjunction with the iMethod field of the pMessage structure to get to the address of the method to be invoked. May not be NULL.
pUnkObject	IUnknown *	this pointer is currently NULL. In the future this might be used to pass the controlling IUnknown of the server object whose method is being invoked.
pvBuffer	void *	the pointer to the incoming debug information. Is undefined (may or may not be NULL) if cbBuffer is zero.
cbBuffer	ULONG	the size of the data pointed to by pvBuffer. May be zero, but as described above, a size of zero can only passed in the case that debugging is already enabled.

.4 DebugORPCServerGetBufferSize

ULONG DebugORPCServerGetBufferSize(pMessage, iid, pChannel, pInterface, pUnkObject)

Called on the server side from within IRpcChannelBuffer::GetBuffer(). See the above overview for further details.

The GUID for this notification is 22080240-9674-101A-B07B-00DD01113F11:

```
GUID __private_to_macro__ = { /* 22080240-9674-101A-B07B-00DD01113F11 */
    0x22080240,
    0x9674,
    0x101A,
    0xB0,
    0x7B,
    {0x00, 0xDD, 0x01, 0x11, 0x3F, 0x11}
  };
```


Argument	Type	Description
pMessage	RPCOLEMESSAGE*	as in DebugORPCServerNotify().
iid	REFIID	as in DebugORPCServerNotify().
pChannel	IRpcChannelBuffer*	as in DebugORPCServerNotify().
pInterface	void *	as in DebugORPCServerNotify().
pUnkObject	IUnknown *	as in DebugORPCServerNotify().
return value	ULONG	the number of bytes that the client debugger wishes to transmit to the server debugger. May legitimately be zero, which indicates that no information need be transmitted. Value is actually returned through lpcbBuffer member of an ORPC_DBG_ALL.

.5 DebugORPCServerFillBuffer

void DebugORPCServerFillBuffer(pMessage, iid, pChannel, pInterface, pUnkObject, pvBuffer, cbBuffer)

Called on the server side immediately after calling IRpcStubBuffer::Invoke(). See the above overview for further details.

The GUID for this notification is 2FC09500-9674-101A-B07B-00DD01113F11:

```
GUID __private_to_macro__ = { /* 2FC09500-9674-101A-B07B-00DD01113F11 */
    0x2FC09500,
    0x9674,
    0x101A,
    0xB0,
    0x7B,
    {0x00, 0xDD, 0x01, 0x11, 0x3F, 0x11}
  };
```

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	as in DebugORPCServerNotify().
iid	REFIID	as in DebugORPCServerNotify().
pChannel	IRpcChannelBuffer*	as in DebugORPCServerNotify().
pInterface	void *	as in DebugORPCServerNotify().
pUnkObject	IUnknown *	as in DebugORPCServerNotify().
pvBuffer	void *	the debug-data buffer which is to be filled. Is undefined (may or may not be NULL) if cbBuffer is zero.
cbBuffer	ULONG	the size of the data pointed to by pvBuffer.

.6 DebugORPCClientNotify

void DebugORPCClientNotify(pMessage, iid, reserved, pUnkProxyObject, hresult, pvBuffer, cbBuffer)

Called on the client side immediately before returning from IRpcChannelBuffer::SendReceive(). See the above overview for further details.

The GUID for this notification is 4F60E540-9674-101A-B07B-00DD01113F11:

```
GUID __private_to_macro__ = { /* 4F60E540-9674-101A-B07B-00DD01113F11 */
    0x4F60E540,
    0x9674,
    0x101A,
    0xB0,
    0x7B,
    {0x00, 0xDD, 0x01, 0x11, 0x3F, 0x11}
  };
```

Argument	Type	Description
pMessage	RPCOLEMESSAGE*	as in DebugORPCClientGetBufferSize().
iid	REFIID	as in DebugORPCClientGetBufferSize().
reserved	void *	reserved for future use.
pUnkProxyObject	IUnknown *	as in DebugORPCClientGetBufferSize().
hresult	HRESULT	the HRESULT of the RPC call that just happened.
pvBuffer	void *	the pointer to the incoming debug information. Is undefined (may or may not be NULL) if cbBuffer is zero.
cbBuffer	ULONG	the size of the data pointed to by pvBuffer.

.5 Special Segments

The COM Library system DLLs have code in specially named segments (sections in COFF terminology) to aid debuggers. The remoting code in the COM interface proxy and interface stub DLLs and other appropriate parts of the runtime are put in segments whose name begins with ".orpc"[84]. These segments are henceforth referred to as .orpc segments. A transition of the instruction pointer from a non .orpc segment to a .orpc segment indicates that the program control is entering the RPC layer. On the client side such a transition implies that a RPC call is about to happen.[85] On the server side if a function is returning back to a .orpc segment it implies that the call is going to return back to the client side. Application writers who write their own remoting code can also avail of this feature by putting their remoting specific code in a .orpc segment.

Debuggers can use this naming convention regarding which code lies in COM RPC to aid in their user interface as to what code they choose to show the user and what code they do not. When the debugger reaches the code address after handling the DebugOrpcServerNotify() exception it should check if it is still in a .orpc segment. This implies that the instruction pointer is still in code that to the programmer is part of the local-remote transparency magic provided by COM, and so should be skipped by the debugger.

Similar behavior on the client side after the DebugOrpcClientNotify() exception is also desirable.

.6 Registry specific information

Windows NT and Windows '95 provide facilities to spawn a debugger when an application faults. Familiarity with the post-mortem debugging support on these systems is assumed in this section.

COM RPC debuggers make use of this mechanism in order to start the debugging of a client or server application that is not presently being debugged. A common scenario is that of a user wanting to step into a RPC call as she is debugging. The client side debugger is notified about the RPC call and sends debugger specific information with the packet. A DebugOrpcServerNotify() notification is raised in the server process. If the server application is already being debugged, it recognizes this as a COM RPC notification and handles it. However if the server application is not being debugged, the system will launch the debugger specified in the AeDebug entry. The debugger will then get the exception notification and handle it.

To avoid having malicious clients being able to force the debugging of a remote server, additional safeguards are required. The COM RPC system checks that the registry key DebugObjectRPCEnabled exists on the system.[86] If this key does not exist, the debug notifications are disabled. Thus, debugging will only take place if explicit action has been taken on a given machine to enable it, and so a remote client cannot cause debugging (and thus denial of service) to occur on an otherwise secure machine.

The full path to this key for a Windows NT system is:

Software\Microsoft\Windows NT\CurrentVersion\DebugObjectRPCEnabled.

For Windows '95 the path to this key is:

Software\Microsoft\Windows\CurrentVersion\DebugObjectRPCEnabled.

---

[84] This is so segment names such as .orpc1, .orpc2... can be used if the remoting code needs to be split up into different segments for swap tuning, etc.

[85] It is not guaranteed that a RPC call will happen for every such transition. The debugger should deal with the case where it receives no notification about an RPC call.

[86] In Windows NT, the registry is securable.

The client side debugger should also ensure that the AeDebug\Debugger entry on its machine is set appropriately.

Before sending any notification, COM sets the AeDebug\Auto entry to 1. This is done in order that the system does not put up a dialog box to ask the user if she wants to debug the server application. Instead it directly launches the debugger.

The scenario where the user steps out of the server application into to a client application which is not being debugged currently is symmetrically identical the preceding insofar as launch of the debugger is concerned.

.7 Format of Debug Information

This section discusses the format of the debug information which the debugger puts into the buffer in the DebugORPCClientFillBuffer and DebugORPCServerFillBuffer calls. The structure of this data is as follows, here specified in an IDL-like manner.[87] For historical reasons, this structure has 1-byte alignment of its internal members. Again, for historical reasons, the data is always transmitted in little-endian byte order.

```
#pragma pack(1) // this structure defined with 1-byte packing alignment
struct {
    DWORD   alwaysOrSometimes;  // controls spawning of debugger
    BYTE    verMajor;           // major version
    BYTE    verMinor;           // minor version
    DWORD   cbRemaining;        // inclusive of byte count itself
    GUID    guidSemantic;       // semantic of this packet
    [switch_is(guidSemantic)] union { // semantic specific information

    // case "step" semantic, guid = 9CADE560-8F43-101A-B07B-00DD01113F11
        BOOL    fStopOnOtherSide;   // should single step or not?

    // case "general" semantic, guid = D62AEDFA-57EA-11ce-A964-00AA006C3706
        USHORT  wDebuggingOpCode;   // should single step or not, etc.
        USHORT  cExtent;            // offset=28
        BYTE    padding[2];         // offset=30, m.b.z.
        [size_is(cExtent)] struct {
            ULONG cb;               // offset=32
            GUID  guidExtent;       // the semantic of this extent
            [size_is(cb)] BYTE  *rgbData;
            };
        };
    }
```

The first DWORD in the debug packet has a special meaning assigned to it. The rest of the debug packet is treated as a stream of bytes by COM and is simply passed across the channel to the debugger on the other side. If the first DWORD contains the value ORPC_DEBUG_ALWAYS (this is a manifest constant defined in the header files) then COM will *always* raise the notification on the other side (use of the four bytes "MARB" is for historical reasons synonymous with use of ORPC_DEBUG_ALWAYS). If the first DWORD in the debug packet contains the value ORPC_DEBUG_IF_HOOK_ENABLED, then the notification is raised on the other side of the channel only if COM debugging has been enabled in that context; that is only if DllDebugObjectRPCHook has been called in that process with fTrace = TRUE. It is the debugger's responsibility to include enough memory for the first DWORD in its response to the DebugOrpcClientGetBufferSize or DebugOrpcServerGetBufferSize notifications.

The two bytes immediately following the initial DWORD contain the major and minor version numbers of the data format specification.

For packets in the format of the current major version, this is followed by

- A DWORD which holds the count of bytes that follow in this data, and which is inclusive of this byte count itself.
- A GUID that identifies the semantic of the packet.
- Semantic specific information. The layout of this information is dependent on the GUID that specifies the semantic. These are as follows:

[87] One can think of this as IDL with a) default packing override, and b) the ability to have a union keyed by a GUID. This will be made more precise in future drafts of this specification.

Semantic	Meaning
Step	This semantic indicates that the single stepping is to be performed or not. The GUID of this semantic is 9CADE560-8F43-101A-B07B-00DD01113F11. The data of this semantic consists of a boolean value which indicates in the "step out of a server" case whether execution should continue once the other side is reached or one should remain stopped.
General	This semantic, which has GUID D62AEDFA-57EA-11ce-A964-00AA006C3706, allows for series of tagged bags of data to be passed. Each is byte counted, and has associated with it a GUID. wDebuggingOpCode allows for one of a series of operations to be specified. Existing-defined opcodes are as follows. Future opcodes are to be allocated by a central coordinating body.[88]

Opcode	Meaning
0x0000	No operation
0x0001	Single step, stop on the other side, as in the "Step" semantic.

Extents presently defined for use in the General semantic are as follows:

Extent	Meaning
Interface pointer	This semantic has GUID 53199051-57EB-11ce-A964-00AA006C3706. The contents of rgbData for this extent is simply an OBJREF, which is the data structure which describes a marshaled interface pointer, the data that results from calling CoMarshalInterface (OBREFs are described later in this specification). Usually, this OBJREF is either the self-enclosed LONGOBJREF variation or a custom-marshaled variation, but this is not required. The LONGOBJREF usually contains a reference count of zero, allowing this information to be freely discared without a leakage of state. Remember that OBJREFs are always in little-endian byte order. An OBJREF is converted into its corresponding interface pointer using CoUnmarshalInterface.

With the Interface Pointer extent, an object can be created in the source debugger's space that relates to the call being made. It can then be marshaled, again, in the source debugger's process, not the source debuggee; this yields an OBJREF. The OBJREF is then transmitted in the course of the call as an extent in the passed debug information. On the destination side, it is conveyed to the destination debugger, who unmarshals it in its process. The result is a COM remoting connection from the source debuggers process to the destination debugger's process that is semantically tied to a particular COM call that needs to be debugged. (TBD) Interfaces on this object can be then be used to provide stack walk-backs, remote memory manipulation, or other debugging functionality.

[88] This is presently Microsoft Corporation.

8. Security

There are two distinguishable categories of security provided by COM. The first form is termed ***Activation Security***, and it dictates how new objects are started, how new and existing objects are connected to, and how certain public services, such as the *Class Table* and the *Running Object Table* are secured. The second form is ***Call Security***, which dictates how security operates at the call level between an established connection from a client to an object (server).

Aspects of the security API are necessarily platform dependent. The Windows versions are shown for reference. Complete interoperability is supported by the user of common, installable authenticators. COM on Windows will support at least Windows NT, Novell Netware, and DCE Kerberos security.

The remainder of this chapter describes these two forms of COM security in detail.

1 Activation Security

As described in previous chapters, objects are exposed to clients either statically, by configuring a persistent registry with information about the server code the *Service Control Manager* launches to retrieve an object, or dynamically, through publishing an object, such as a class object via CoRegisterClassObject or a running object via IRunningObjectTable::Register. Accordingly, there are two aspects to activation security, one static (or automatic) form, and one dynamic form.

Activation security is automatically applied by the *Service Control Manager* of a particular machine. Upon receipt of a request to retrieve an object, the *Service Control Manager* checks the request against security information stored either within its registry or gathered dynamically from objects and stored within its internal tables.

.1 Registry Configuration

All *Service Control Managers* should offer a level of simple registry-driven configurability for use administering classes of a machine and for specific user accounts on that machine. The following tables contain suggested configuration variables and a description of their Win32 implementation as elaboration.

Machine Wide Settings	Use	Win32 Implementation
Allow Activation	Boolean enables and disables activation on a machine-wide basis.	HKLM[89]\Software\Network OLE\Enabled = [0 | 1]
Per-Class Security	Establishes automatic activation security for a specific class registered for use by any users on this machine.	HKCC[90]\CLSID\{厩\ActivationSecurity is secured.[91] HKCC\CLSID\{厩\FindActivationSecurityAt = {clsid} points to a class with an \ActivationSecurity secured key.
Default Class Security	Establishes automatic activation security for any classes without per-class security registered for use by any user on this machine.	HKLM\Software\Network OLE\DefaultActivationSecurity is secured.
Default ROT Security	Defines default security on objects placed in the *Running Object Table* of this machine by any user.	HKLM\Software\Network OLE\DefaultROTSecurity is secured.

[89] Shorthand for HKEY_LOCAL_MACHINE, the section of the registry containing machine-wide software configuration information. Typically holds configuration information used by server applications that are not running as a particular user but rather on behalf of the system.

[90] Shorthand for HKEY_COMMON_CLASSES, the section of the registry containing machine-wide class information (mappings between CLSID's and DLL/EXE names).

[91] Under Win32, when a key is *secured* the act of retrieving its value performs an access check against the security descriptor that guards it. Therefore, the *SCM*'s retrieval of the value of a secure key causes an implicit access check.

Per-User Settings	Use	Win32 Implementation
Allow Activation	Boolean enables and disables all activation for a particular user on this machine.	HKCU[92]\Software\Network OLE\Enabled = [0 | 1]
Per-Class Security	Establishes automatic activation security for a specific class registered for use by a particular user on this machine.	HKCR[93]\CLSID\{厖\ActivationSecurity is secured. HKCR\CLSID\{厖\FindActivationSecurityAt = {clsid} points to a class with an \ActivationSecurity secured key.
Default Class Security	Establishes automatic activation security for any classes without per-class security registered for use by a particular user on this machine.	HKCU\Software\Network OLE\DefaultActivationSecurity is secured.
Default ROT Security	Defines default security on objects placed in the *Running Object Table* of this machine by a particular user.	HKCU\Software\Network OLE\DefaultROTSecurity is secured.

.2 IActivationSecurity Interface

The IActivationSecurity interface is exposed by objects which register themselves via CoRegisterClassObject and IRunningObjectTable::Register in order to secure access to the tables in which these objects are registered, as described above.

```
interface IActivationSecurity : IUnknown {
    HRESULT     GetSecurityDescriptor(SECURITY_DESCRIPTOR** ppSecDesc);
    };
```

.1 IActivationSecurity::GetSecurityDescriptor

HRESULT IActivationSecurity::GetSecurityDescriptor(ppSecDesc);

Retrieves the security descriptor associated with this object. This security descriptor is used to control access to this object pointer in system-maintained tables.

Argument	Type	Description
ppSecDesc	SECURITY_DESCRIPTOR**	Location in which to return a pointer to the security descriptor for activation or binding to this object.
Returns	S_OK	Success. *ppSecDesc refers to a valid SECURITY_DESCRIPTOR.
	E_INVALIDARG	One or more arguments are invalid.

[92] Shorthand for HKEY_CURRENT_USER, the section of the registry containing per-user software configuration information. Typically holds configuration information used by applications that are running on behalf of a particular user.

[93] Shorthand for HKEY_CLASSES_ROOT, the section of the registry containing per-user class information (mappings between CLSID's and DLL/EXE names).

.3 Applying Activation Security

The following table outlines how activation security is applied to requests to the *Service Control Manager*.

Request	Action
CoGetClassObject or CoCreateInstance of a non-running class *X*	• Check "HKLM\Software\Network OLE\Enabled". Fail the request if zero. • Check "HKCU\Software\Network OLE\Enabled". Fail the request if zero. • If class is registered in HKCR, follow "HKCR\CLSID\{厓\FindActivationSecurityAt = {厓" until an "HKCR\CLSID\{厓\ActivationSecurity" key is found. If these keys do not exist, use "HKCU\Software\Network OLE\Default Activation Security". Check the request against the security on this key. • Otherwise, if class is registered in HKCC, follow "HKCC\CLSID\{厓\FindActivationSecurityAt = {厓" until an "HKCC\CLSID\{厓\ActivationSecurity" key is found. If these keys do not exist, use "HKLM\Software\Network OLE\Default Activation Security". Check the request against the security on this key.
CoGetClassObject or CoCreateInstance of a running class *Y*	• Check "HKLM\Software\Network OLE\Enabled". Fail the request if zero. • Check "HKCU\Software\Network OLE\Enabled". Fail the request if zero. • Check the request against the SECURITY_DESCRIPTOR available from CoRegisterClassObject(CLSID_Y, ?). This will be either the value returned by the class object's IactivationSecurity::GetSecurityDescriptor at the time of CoRegisterClassObject or will have been taken from "HKCU\Software\Network OLE\DefaultActivationSecurity" or "HKLM\Software\Network OLE\DefaultActivationSecurity" at the time of CoRegisterClassObject if the class object did not support IActivationSecurity.
Running Object Table	• Check "HKLM\Software\Network OLE\Enabled". Fail the request if zero. • Check "HKCU\Software\Network OLE\Enabled". Fail the request if zero. • Before performing any operation against a ROT entry (i.e., IRunningObjectTable::Revoke, IRunningObjectTable::IsRunning, IRunningObjectTable::GetObject, IRunningObjectTable::NoteTimeChange, IRunningObjectTable::GetTimeOfLastChange, or when including an entry in an IEnumMoniker::Next of an IEnumMoniker returned from IRunningObjectTable::EnumRunning), check the call against the SECURITY_DESCRIPTOR available from IRunningObjectTable::Register. This will be either the value returned by the object's IActivationSecurity::GetSecurityDescriptor at the time of IRunningObjectTable::Register or will have been taken from "HKCU\Software\Network OLE\DefaultROTSecurity" or "HKLM\Software\Network OLE\DefaultROTSecurity" at the time of IRunningObjectTable::Register if the object did not support IActivationSecurity.

2 Call Security

COM provides two mechanisms to secure calls. The first mechanism is similar to DCE-RPC: COM provides APIs that applications may use do their own security checking. The second mechanism is done automatically by the COM infrastructure. If the application provides some setup information, COM will make all the necessary checks to secure the application. This automatic mechanism does security checking for the process, not for individual objects or methods. If an application wants more fine grained security, it performs its own checking. However, the two mechanisms are not exclusive: an application may ask COM to perform automatic security checking and then perform its own.

COM call security services are divided into three categories: general APIs called by both clients and servers, new interfaces on client proxies, and server-side APIs and call-context interfaces. The general APIs allow the automatic security mechanism to be initialized and automatic authentication services to be registered. The proxy interfaces allow the client to control the security on calls to individual interfaces. The server APIs and interfaces allow the server to retrieve security information about a call and to impersonate the caller.

In a typical scenario, the client queries the object for IClientSecurity, which is implemented locally by the remoting layer. The client uses IClientSecurity to control the security of individual interface proxies on the object prior to making a call on one of the interfaces. When a call arrives at the server, the server may call CoGetCallContext to retrieve an IServerSecurity interface. IServerSecurity allows the server to check the client's authentication and to impersonate the client, if needed. The IServerSecurity object is valid for the duration of the call. CoInitializeSecurity allows the client to establish default call security for the process, avoiding the use of IClientSecurity on individual proxies. CoInitializeSecurity and CoRegisterAuthenticationServices allow a server to register automatic authentication services for the process.

Implementations of QueryInterace must never check ACLs. COM requires that an object which supports a particular IID always return success when queried for that IID. Aside from the requirement, checking ACLs on QueryInterface does not provide any real security. If client A legally has access to interface IFoo, A can hand it directly to B without any calls back to the server. Additionally, OLE caches interface pointers and will not call QueryInterface on the server every time a client does a query.

Each time a proxy is created, COM sets the security information to default values, which are the same values used for automatic security.

.1 General Call Security APIs

.1 RPC_C_AUTHN Constants

Value	Description
RPC_C_AUTHN_LEVEL_NONE	Performs no authentication.
RPC_C_AUTHN_LEVEL_CONNECT	Authenticates only when the client establishes a relationship with the server. Datagram transports always use RPC_AUTHN_LEVEL_PKT instead.
RPC_C_AUTHN_LEVEL_CALL	Authenticates only at the beginning of each remote procedure call when the server receives the request. Datagram transports use RPC_C_AUTHN_LEVEL_PKT instead.
RPC_C_AUTHN_LEVEL_PKT	Authenticates that all data received is from the expected client.
RPC_C_AUTHN_LEVEL_PKT_INTEGRITY	Authenticates and verifies that none of the data transferred between client and server has been modified.
RPC_C_AUTHN_LEVEL_PKT_PRIVACY	Authenticates all previous levels and encrypts the argument value of each remote procedure call.

.2 RPC_C_IMP Constants

Value	Description
RPC_C_IMP_LEVEL_ANONYMOUS	The client is anonymous to the server. The server process cannot obtain identification information about the client and it cannot impersonate the client.
RPC_C_IMP_LEVEL_IDENTIFY	The server can obtain the client's identity. The server can impersonate the client for ACL checking but cannot access system objects as the client. This information is obtained when the connection is established, not on every call.
RPC_C_IMP_LEVEL_IMPERSONATE	The server process can impersonate the client's security context while acting on behalf of the client. This information is obtained when the connection is established, not on every call.
RPC_C_IMP_LEVEL_DELEGATE	The server process can impersonate the client's security context while acting on behalf of the client. The server process can also make outgoing calls to other servers while acting on behalf of the client. This information is obtained when the connection is established, not on every call.

.3 CoInitializeSecurity

HRESULT CoInitializeSecurity(pSecDesc, AuthnLevel, Reserved);

Initializes the security layer.

Argument	Type	Description
pSecDesc	SECURITY_DESCRIPTOR*	This parameter contains two ACLs. The discretionary ACL indicates who is allowed to call this process and who is explicitly denied. The system ACL contains audit information. COM will write an audit entry for each account listed in the system ACL if the administrator for the machine has turned on auditing of COM calls on the machine. A NULL SACL implies no auditing. A SACL with no ACEs also implies no auditing. A NULL DACL will allow calls from anyone. A DACL with no ACEs allows no access. If the application passes a NULL security descriptor, COM will construct one that allows calls from the current user and local system. All calls will be audited. COM does not actually audit every call. It only audits new connections. COM will

		hold a pointer to the security descriptor until the last call to CoUninitialize completes. The descriptor and its components may be allocated any way the application desires, but it may not be freed until after the application uninitializes COM.
AuthnLevel	ULONG	This parameters defines the security level and impersonation level used by automatic security. It may contain one of the values from each of the RPC_C_AUTHN and RPC_C_IMP constants OR'd together. Additionally, the value RPC_C_AUTHN_MUTUAL may be OR'd in. This value causes the authentication service to guarantee that the client can find out the login account of the server securely. When calls arrive, they must be at least as high as the specified security level and impersonation level. If not, COM will automatically fail the call. Outgoing calls will be made at the specified security level or higher if COM has a hint from the server. The impersonation level will always be set as specified and not negotiated. Dynamic impersonation is not supported.
Reserved	void*	This parameter is reserved for future use. It must be set to NULL.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

.4 CoQueryAuthenticationServices

HRESULT CoQueryAuthenticationServices(pcbAuthSvc, adwAuthSvc);

Returns a list of the authentication services that are installed on the machine. The list can be used as input to CoRegisterAuthenticationService. Different authentication services support different levels of security. For example, NTLMSSP does not support delegation or mutual authentication while *Kerberos* does. The application is responsible for only registering authentication services that provide the features the application needs. There is no way to query which services have been registered with CoRegisterAuthenticationService.

Argument	Type	Description
pcbAuthSvc	DWORD*	Returns a count of the authentication services supported on the machine.
adwAuthSvc	DWORD**	Returns a list of authentication services supported on the machine. The enumeration of authentication services is in rpcdce.h. Authentication services that are not currently in the enumeration may be installed on a machine without upgrading the operating system. The list is allocated by CoTaskMemAlloc. The application must free the list by calling CoTaskMemFree.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.
	E_OUTOFMEMORY	Insufficient memory to create the adwAuthSvc out-parameter.

.5 CoRegisterAuthenticationService

HRESULT CoRegisterAuthenticationServices(cbauthSvc, asAuthSvc);

This API sets the list of authentication services COM will use to authenticate incoming calls. If a call arrives with a different authentication service, the call will fail. Registering authentication services does not prevent the arrival of unsecure calls (i.e., calls with no authentication service). This API can only be called before any interfaces are marshaled. Thus servers must call this if they want security. This call is not useful for clients (unless they are also servers).

This API can only be called once.

An application cannot call both CoInitializeSecurity and CoRegisterAuthenticationService.

Argument	Type	Description
cbAuthSvc	DWORD	Specify the number of authentication services in the list asAuthSvc.
asAuthSvc	SOLE_AUTHENTICATION_SERVICE*	An array of authentication services to register. The authentication services are enumerated in rpcdce.h. COM copies the list. If the principal name is NULL, COM will assume the current user id. A NULL principal name will work for NTLMSSP and *Kerberos*. It may or may not work for other authentication services.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

.2 IClientSecurity Interface

IClientSecurity gives the client control over the call-security of individual interfaces on a remote object.

All proxies generated by the COM MIDL compiler support the IClientSecurity interface. If QueryInterface for IClientSecurity fails, either the object is implemented in-process or it is remoted by a custom marshaler which does not support security (a custom marshaler may support security by offering the IClientSecurity interface to the client). The proxies passed as parameters to an IClientSecurity method must be from the same object as the IClientSecurity interface.

```
interface IClientSecurity : IUnknown {
    HRESULT    QueryBlanket(void* pProxy, DWORD* pcbAuthnSvc, SOLE_AUTHENTICATION_SERVICE* pasAuthnSvc,
        RPC_AUTH_IDENTITY_HANDLE** ppAuthInfo, DWORD* AuthnLevel);
    HRESULT    SetBlanket(void* pProxy, DWORD AuthnSvc, WCHAR* ServerPrincName, RPC_AUTH_IDENTITY_HANDLE* pAuthInfo,
        DWORD AuthnLevel, DWORD AuthzSvc);
    HRESULT    CopyProxy(void* pProxy, REFIID riid, void** ppCopy);
    };
```

.1 IClientSecurity::QueryBlanket

HRESULT IClientSecurity::QueryBlanket(pProxy, pcbAuthnSvc, pasAuthnSvc, ppAuthInfo, AuthnLevel);

This method returns authentication information. This method is called by the client to find out what authentication information COM will use on calls made from the specified proxy.

Argument	Type	Description
pProxy	void*	This parameter indicates the proxy to query.
pcbAuthnSvc	DWORD*	This parameter indicates the number of entries in the array pasAuthSvc.
pasAuthnSvc	SOLE_AUTHENTICATION_SERVICE*	This parameter is an array of authentication service, principal name pairs. The first entry is the one that COM will use to make calls to the server. The array is allocated with CoTaskMemAlloc and the application must free it by calling CoTaskMemFree.
ppAuthInfo	RPC_AUTH_IDENTITY_HANDLE**	This parameter returns the value passed to CoSetProxyAuthenticationInfo. It may be NULL if you do not care.
AuthnLevel	DWORD*	This parameter returns the current authentication level. It may be NULL if you do not care.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.
	E_OUTOFMEMORY	Insufficient memory to create the pasAuthnSvc out-parameter.

.2 IClientSecurity::SetBlanket

HRESULT IClientSecurity::SetBlanket(pProxy, AuthnSvc, ServerPrincName, pAuthInfo, AuthnLevel, AuthzSvc);

This method sets the authentication information that will be used to make calls on the specified proxy. The values specified here override the values chosen by automatic security. Calling this method changes

the security values for all other users of the specified proxy. Use IClientSecurity::CopyProxy to make a private copy.

By default the authentication service and principal name is set to a list of authentication service and principal name pairs that were registered on the server. When this method is called COM will forget the default list. By default COM will try one principal name from the list of authentication services available on both machines. It will not retry if that principal name fails.

If pAuthInfo is not set, it defaults to the logged in id. AuthnLevel and AuthzSvc default to the values specified to CoInitializeSecurity. If CoInitializeSecurity is not called, they default to RPC_C_AUTHN_LEVEL_NONE and RPC_C_AUTHZ_NONE.

Security information will often be ignored if set on local interfaces. For example, it is legal to set security on the IClientSecurity interface. However, since that interface is supported locally, there is no need for security. IUnknown and IMultiQuery are special cases. The local implementation makes remote calls to support these interfaces. The local implementation will use the security settings for those interfaces.

Argument	Type	Description
pProxy	void*	This parameter indicates the proxy to set.
AuthnSvc	DWORD	This parameter indicates the authentication service. It may be RPC_C_AUTHN_NONE if no authentication is required. It may be RPC_C_AUTHN_DONT_CHANGE if you do not want to change the current value.
ServerPrincName	WCHAR*	This parameter indicates the server principal name. It may be NULL if you don't want to change the current value.
pAuthInfo	RPC_AUTH_IDENTITY_HANDLE*	This parameter sets the identity of the client. It is authentication service specific. Some authentication services allow the application to pass in a different user name and password. COM keeps a pointer to the memory passed in until COM is uninitialized or a new value is set. If NULL is specified COM uses the current identity (whether the logged in or impersonated id).
AuthnLevel	DWORD	This parameter specifies the authentication level. It may be RPC_C_AUTHN_LEVEL_DONT_CHANGE if you do not want to change the current value.
AuthzSvc	DWORD	This parameter specifies the authorization level. It may be RPC_C_AUTHZ_DONT_CHANGE is you do not want to change the current value.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments is invalid.

.3 IClientSecurity::CopyProxy

HRESULT IClientSecurity::CopyProxy(pProxy, riid, ppCopy)

This method makes a copy of the specified proxy. Its authentication information may be changed without affecting any users of the original proxy. The copy has the default values for the authentication information. The copy has one reference and must be released.

Local interfaces may not be copied. IUnknown, IMultiQuery, and IClientSecurity are examples of existing local interfaces.

Argument	Type	Description
pProxy	void*	This parameter indicates the proxy to copy.
riid	REFIIID	Identifies the proxy to return.
ppCopy	void**	The copy is returned to this parameter.
Returns	S_OK	Success.
	E_NOINTERFACE	The interface riid is not supported by this object.

.3 Client APIs for Call Security

.1 CoQueryProxyAuthenticationInfo

HRESULT CoQueryProxyAuthenticationInfo(pProxy, pcbAuthnSvc, pasAuthnSvc, ppAuthInfo, pAuthnLevel);

Returns the authentication information used to make calls on the specified proxy. This function encapsulates the following sequence of common calls:

```
pProxy->QueryInterface(IID_IClientSecurity, (void**)&pcs);
pcs->QueryBlanket(pProxy, AuthnSvc, ServerPrincName, pAuthInfo, AuthnLevel);
pcs->Release();
```

Argument	Type	Description

see IClientSecurity::QueryBlanket

.2 CoSetProxyAuthenticationInfo

HRESULT CoSetProxyAuthenticationInfo(pProxy, AuthnSvc, ServerPrincName, pAuthInfo, AuthnLevel, AuthzSvc);

Sets the authentication information that will be used to make calls on the specified proxy. This function encapsulates the following sequence of common calls:

```
pProxy->QueryInterface(IID_IClientSecurity, (void**)&pcs);
pcs->SetBlanket(pProxy, AuthnSvc, ServerPrincName, pAuthInfo, AuthnLevel);
pcs->Release();
```

Argument	Type	Description

see IClientSecurity::SetBlanket

.3 CoCopyProxy

HRESULT CoCopyProxy(pProxy, riid, ppCopy);

Makes a copy of the specified proxy. This function encapsulates the following sequence of common calls:

```
pProxy->QueryInterface(IID_IClientSecurity, (void**)&pcs);
pcs->CopyProxy(pProxy, riid, ppCopy);
pcs->Release();
```

Argument	Type	Description

see IClientSecurity::CopyProxy

.4 IServerSecurity Interface

IServerSecurity may be used to impersonate the client during a call, even on other threads within the server. IServerSeciruty:QueryBlanket and IServerSecurity::ImpersonateClient may only be called before the call completes. IServerSecurity::RevertToSelf may be called at any time. The interface pointer must be released when it is no longer needed. Unless the server wishes to impersonate the client on another thread, there is not reason to keep an IServerSecurity past the end of the call, since it will at that point no longer support IServerSecurity::QueryBlanket.

```
interface IServerSecurity : IUnknown {
    HRESULT     QueryBlanket(RPC_AUTHZ_HANDLE* Privs, WCHAR** ServerPrincName, DWORD* AuthnLevel, DWORD* AuthnSvc,
        DWORD* AuthzSvc );
    HRESULT     ImpersonateClient(void);
    HRESULT     RevertToSelf(void);
    };
```

.1 IServerSecurity::QueryBlanket

HRESULT IServerSecurity::QueryBlanket(Privs, ServerPrincName, AuthnLevel, AuthnSvc, AuthzSvc);

This method is used by the server to find out about the client that invoked one of its methods. CoGetCallContext with IID_ISeverSecurity returns an IServerSecurity interface for the current call on the current thread. This interface pointer may be used on any thread and calls to it may succeed until the call completes.

Argument	Type	Description
Privs	RPC_AUTHZ_HANDLE*	Returns a pointer to a handle to the privilege information for the client application. The format of the structure is authentication service specific. The application should not write or free the memory. The information is only valid for the duration of the current call. NULL may be passed if the application is not interested in this parameter.
ServerPrincName	WCHAR*	This parameter indicates the principal name the client specified. It is a copy allocated with CoTaskMemAlloc. The application must call CoTaskMemFree to release it. NULL may be passed if the application is not interested in this parameter.
AuthnLevel	DWORD*	This parameter indicates the authentication level. NULL may be passed if the application is not interested in this parameter.
AuthnSvc	DWORD*	This parameter indicate the authentication service the client specified. NULL may be passed if the application is not interested in this parameter.
AuthzSvc	DWORD*	This parameter indicates the authorization service. NULL may be passed if the application is not interested in this parameter.
Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.
	E_OUTOFMEMORY	Insufficient memory to create one or more out-parameters.

.2 IServerSecurity::ImpersonateClient

HRESULT IServerSecurity::ImpersonateClient();

This method allows a server to impersonate a client for the duration of a call. The server may impersonate the client on any secure call at identify, impersonate, or delegate level. At identify level, the server may only find out the clients name and perform ACL checks; it may not access system objects as the client. At delegate level the server may make off machine calls while impersonating the client. The impersonation information only lasts till the end of the current method call. At that time IServerSecurity::RevertToSelf will automatically be called if necessary.

Impersonation information is not normally nested. The last call to any Win32 impersonation mechanism overrides any previous impersonation. However, in the apartment model, impersonation is maintained during nested calls. Thus if the server ***A*** receives a call from ***B***, impersonates, calls ***C***, receives a call from ***D***, impersonates, reverts, and receives the reply from ***C***, the impersonation will be set back to ***B***, not ***A***.

If IServerSecurity::ImpersonateClient is called on a thread other then the one that received the call, the impersonation will not automatically be revoked. It will be valid past the end of the call. However, IServerSecurity::ImpersonateClient must be called before the original call completes.

Argument	Type	Description
Returns	S_OK	Success.
	E_FAIL	The caller can not impersonate the client identified by this ISeverSecurity object.

.3 IServerSecurity::RevertToSelf

HRESULT IServerSecurity::RevertToSelf();

This method restores the authentication information on a thread to the process's identity.

In the apartment model, IServerSecurity::RevertToSelf only affects the current method invocation. If there are nested method invocations, they each may have there own impersonation and COM will correctly restore the impersonation before returning to them (regardless of whether or not IServerSecurity::RevertToSelf was called).

IServerSecurity::RevertToSelf may be called on threads other then the one that received the call. IServerSecurity::RevertToSelf may be called after the call completes. Calls to IServerSecurity::RevertToSelf that are not matched with an IServerSecurity::ImpersonateClient call will fail.

Argument	Type	Description
Returns	S_OK	Success.
	E_FAIL	This call was not preceded by a call to IServerSecurity::ImpersonateClient on this thread of execution.

.5 *Sever APIs for Call Security*

The following APIs are provided to give the server access to any contextual information of the caller and to encapsulate common sequences of security checking and caller impersonation.

.1 CoGetCallContext

HRESULT CoGetCallContext(riid, ppv);

Retrieves the context of the current call on the current thread. riid specifies the interface on the context to retrieve. Currently only IServerSecurity is available from the default call-context (see ISeverSecurity for details).

Argument	Type	Description
riid	REFIID	Identifies the interface to return.
ppv	void**	Returns an interface for the current call.
Returns	S_OK	Success.
	E_NOINTERFACE	The call context does not support the interface identified by riid.

.2 CoSetCallContext

HRESULT CoSetCallContext(punk);

Establishes the call context for the current call, overriding the default call context object normally available via CoGetCallContext.

This function is provided primarily for objects performing custom marshaling. Before transferring control from their stub or IPC mechanism to the server-side code, a custom marshaler may establish the call context via CoSetCallContext so that subsequent objects can be written to take advantage of call-level security or other caller-specific contextual information in a transport neutral fashion, e.g. without regard to whether an object between them and the client was remoted via custom marshaling.

The call context reverts automatically at the end of each call. Furthermore, a custom marshaling layer which calls CoSetCallContext prior to entering the server need not call CoSetCallContext(NULL) after each returning call.

A second call to CoSetCallContext with a non-NULL punk will Release the first punk and AddRef the second.

Argument	Type	Description
punk	IUnknown*	When non-NULL, the IUnknown which is to be QueryInterface'd for the requested call context interface by subsequent calls to CoGetCallContext during the span of the current call. This interface is AddRef'd prior to

returning. When NULL, resets the call context to the COM-provided default for the current call.

Returns	S_OK	Success.
	E_INVALIDARG	One or more arguments are invalid.

.3 CoQueryClientAuthenticationInfo

HRESULT CoQueryClientAuthenticationInfo(Privs, ServerPrincName, AuthnLevel, AuthnSvc, AuthzSvc);

Used by the server to find out about the client that invoked the method executing on the current thread. This function encapsulates the following sequence of common calls:

```
CoGetCallContext(IID_IServerSecurity, (void**)&pss);
pss->QueryBlanket(Privs, ServerPrincName, AuthnLevel, AuthnSvc, AuthzSvc);
pss->Release();
```

Argument	Type	Description

see IServerSecurity::QueryBlanket

.4 CoImpersonateClient

HRESULT CoImpersonateClient();

Allows the server to impersonate the client of the current call for the duration of the call. This function encapsulates the following sequence of common calls:

```
CoGetCallContext(IID_IServerSecurity, (void**)&pss);
pss->ImpersonateClient();
pss->Release();
```

Argument	Type	Description

see IServerSecurity::ImpersonateClient

.5 CoRevertToSelf

HRESULT CoRevertToSelf();

Restores the authentication information on a thread of execution to its previous identity. This function encapsulates the following sequence of common calls:

```
CoGetCallContext(IID_IServerSecurity, (void**)&pss);
pss->RevertToSelf();
pss->Release();
```

Argument	Type	Description

see IServerSecurity::RevertToSelf

Part III: Component Object Model Protocols and Services

This page intentionally left blank.

9. Connectable Objects

The COM technology known as Connectable Objects (also called "connection points") supports a generic ability for any object, called in this context a "connectable" object, to express these capabilities:

- The existence of "outgoing" interfaces,[94] such as event sets
- The ability to enumerate the IIDs of the outgoing interfaces
- The ability to connect and disconnect "sinks" to the object for those outgoing IIDs
- The ability to enumerate the connections that exist to a particular outgoing interface.

Support for these capabilities involves four interfaces: IConnectionPointContainer, IEnumConnectionPoints, IConnectionPoint, and IEnumConnections. A "connectable object" implements IConnectionPointContainer to indicate existence of outgoing interfaces. Through this interface a client can enumerate connection points for each outgoing IID (via an enumerator with IEnumConnectionPoints) and can obtain an IConnectionPoint interface to a connection point for each IID. Through a connection point a client starts or terminates an advisory loop with the connectable object and the client's own sink. The connection point can also enumerate the connections it knows about through an enumerator with IEnumConnections.

1 The *IConnectionPoint* Interface

The ability to connect to a single outgoing interface (that is, for a unique IID) is provided by a "connection point" sub-object that is conceptually owned by the connectable object. The object is separate to avoid circular reference counting problems. Through this interface the connection point allows callers to connect a sink to the connectable object, to disconnect a sink, or to enumerate the existing connections.

IDL:

```
[
uuid(B196B286-BAB4-101A-B69C-00AA00341D07)
    , object, pointer_default(unique)
]
interface IConnectionPoint : IUnknown
    {
    HRESULT GetConnectionInterface([out] IID *pIID);
    HRESULT GetConnectionPointContainer([out] IConnectionPointContainer **ppCPC);
    HRESULT Advise([in] IUnknown *pUnk, [out] DWORD *pdwCookie);
    HRESULT Unadvise([in] DWORD dwCookie);
    HRESULT EnumConnections([out] IEnumConnections **ppEnum);
    }
```

A connection point is allowed to stipulate how many connections (one or more) it will allow in its implementation of Advise. A connection point that allows only one interface can return E_NOTIMPL from EnumConnections.

[94] An "outgoing" interface is one that an object defines itself but for which the object is itself a client. Another piece of code called the "sink" (generically) implements the outgoing interface such that the object can call the sink.

.1 IConnectionPoint::GetConnectionInterface

HRESULT IConnectionPoint::GetConnectionInterface([out] IID *pIID);

Returns the IID of the outgoing interface managed by this connection point. This is provided such that a client of IEnumConnectionPoints can determine the IID of each connection point thus enumerated. The IID returned from this method must enable the caller to access this same connection point through IConnectionPointContainer::FindConnectionPoint.

Argument	Type	Description
pIID	IID *	[out] A pointer to the caller's variable to receive the IID of the outgoing interface managed by this connection point.

Return Value	**Meaning**
S_OK	Success.
E_POINTER	The address in pIID is not valid (such as NULL)
E_UNEXPECTED	An unknown error occurred.

Comments:

This function must be completely implemented in any connection point; therefore E_NOTIMPL is not an acceptable return code.

.2 IConnectionPoint::GetConnectionPointContainer

HRESULT IConnectionPoint::GetConnectionPointContainer([out] IConnectionPointContainer **ppCPC);

Retrieves the IConnectionPointContainer interface pointer to the connectable object that conceptually owns this connection point. The caller becomes responsible for the pointer on a successful return.

Argument	Type	Description
ppCPC	IConnectionPointContainer *	[out] A pointer to the caller's variable in which to return a pointer to the connectable object's IConnectionPointContainer interface. The connection point will call IConnectionPointContainer::AddRef before returning and the caller must call IConnectionPoint::Release when it is done using the pointer.

Return Value	**Meaning**
S_OK	Success.
E_POINTER	The value in ppCPC is not valid (such as NULL)
E_UNEXPECTED	An unknown error occurred.

Comments:

E_NOTIMPL is not an allowable return code.

.3 IConnectionPoint::Advise

HRESULT IConnectionPoint::Advise([in] IUnknown *pUnk, [out] DWORD *pdwCookie);

Establishes an advisory connection between the connection point and the caller's sink object identified with pUnk. The connection point must call pUnk->QueryInterface(iid, ...) on this pointer in order to obtain the correct outgoing interface pointer to call when events occur, where *iid* is the inherent outgoing interface IID managed by the connection point (that is, the that when passed to

IConnectionPointContainer::FindConnectionPoint would return an interface pointer to this same connection point).

Upone successful return, the connection point provides a unique "cookie" in *pdwCookie that must be later passed to IConnectionPoint::Unadvise to terminate the connection.

Argument	Type	Description
pUnk	IUnknown *	[in] The IUnknown pointer to the client's sink that wishes to receive calls for the outgoing interface managed by this connection point. The connection point must query this pointer for the correct outgoing interface. If this query fails, this member returns CONNECT_E_CANNOTCONNECT.
pdwCookie	DWORD *	[out] A pointer to the caller's variable that is to receive the connection "cookie" when connection is successful. This cookie must be unique for each connection to any given *instance* of a connection point.

Return Value	**Meaning**
S_OK	The connection has been established and *pdwCookie has the connection key.
E_POINTER	The value of pUnk or pdwCookie is not valid (NULL cannot be passed for either argument)
E_UNEXPECTED	An unknown error occurred.
E_OUTOFMEMORY	There was not enough memory to complete the operation, such as if the connection point failed to allocate memory in which to store the sink's interface pointer.
CONNECT_E_ADVISELIMIT	The connection point has already reached its limit of connections and cannot accept any more.
CONNECT_E_CANNOTCONNECT	The sink does not support the interface required by this connection point.

.4 IConnectionPoint::Unadvise

HRESULT IConnectionPoint::Unadvise([in] DWORD dwCookie);

Terminates an advisory connection previously established through IConnectionPoint::Advise. The dwCookie argument identifies the connection to terminate.

Argument	Type	Description
dwCookie	DWORD	[in] The connection "cookie" previously returned from IConnectionPoint::Advise.

Return Value	**Meaning**
S_OK	The connection was successfully terminated.
E_UNEXPECTED	An unknown error occurred.
CONNECT_E_NOCONNECTION	dwCookie does not represent a value connection to this connection point.

.5 IConnectionPoint::EnumConnections

HRESULT IConnectionPoint::EnumConnections([out] IEnumConnections **ppEnum);

Creates an enumerator object for iteration through the connections that exist to this connection point.

Argument	Type	Description
ppEnum	IEnumConnections *	[out] A pointer to the caller's variable to receive the interface pointer of the newly created enumerator. The caller is responsible for releasing this pointer when it is no longer needed.

Return Value	**Meaning**
S_OK	Success.
E_POINTER	The address in *ppEnum* is not valid (such as NULL)
E_NOTIMPL	The connection point does not support enumeration.
E_UNEXPECTED	An unknown error occurred.
E_OUTOFMEMORY	There was not enough memory to create the enumerator.

2 The *IConnectionPointContainer* Interface

When implemented on an object, makes the object "connectable" and expresses the existence of outgoing interfaces on the object. Through this interface a client may either locate a specific "connection point" for one IID or it can enumerate the connections points that exist.

IDL:

```
[
uuid(B196B284-BAB4-101A-B69C-00AA00341D07)
    , object, pointer_default(unique)
]
interface IConnectionPointContainer : IUnknown
    {
    HRESULT EnumConnectionPoints([out] IEnumConnectionPoints **ppEnum);
    HRESULT FindConnectionPoint([in] REFIID riid
        , [out] IConnectionPoint **ppCP);
    }
```

.1 IConnectionPointContainer::EnumConnectionPoints

HRESULT IConnectionPointContainer::EnumConnectionPoints([out] IEnumConnectionPoints **ppEnum);

Creates an enumerator of all the connection points supported in the connectable object, one connection point per IID. Since IEnumConnectionPoints enumerates IConnectionPoint* types, the caller must use IConnectionPoint::GetConnectionInterface to determine the actual IID that the connection point supports.

The caller of this member must call (*ppEnum)->Release when the enumerator object is no longer needed.

Argument	Type	Description
ppEnum	IEnumConnectionPoints *	[out] A pointer to the caller's variable that is to receive the interface pointer to the enumerator. The caller is responsible for releasing this pointer after this function returns successfully.

Return Value	**Meaning**
S_OK	The enumerator was created successfully.
E_UNEXPECTED	An unknown error occurred.
E_POINTER	The value passed in ppEnum is not valid (such as NULL).
E_OUTOFMEMORY	There was not enough memory to create the enumerator object.

Comments:

E_NOTIMPL is specifically disallowed because outside of type information there would be no other means through which a caller could find the IIDs of the outgoing interfaces.

.2 IConnectionPointContainer::FindConnectionPoint

HRESULT FindConnectionPoint([in] REFIID riid , [out] IConnectionPoint **ppCP);

Asks the "connectable object" if it has a connection point for a particular IID, and if so, returns the IConnectionPoint interface pointer to that connection point. Upon successful return, the caller must call IConnectionPoint::Release when that connection point is no longer needed.

Note that this function is the QueryInterface equivalent for an object's outgoing interfaces, where the outgoing interface is specified with riid and where the interface pointer returned is always that of a connection point.

Argument	Type	Description
riid	REFIID	[in] A reference to the outgoing interface IID whose connection point is being requested.
ppCP	IConnectionPoint **	[out] The address of the caller's variable that is to receive the IConnectionPoint interface pointer to the connection point that manages the outgoing interface identified with riid. This is set to NULL on failure of the call; otherwise the caller must call IConnectionPoint::Release when the connection point is no longer needed.

Return Value	**Meaning**
S_OK	The call succeeded and *ppCP has a valid interface pointer.
E_POINTER	The address passed in ppCP is not valid (such as NULL)
E_UNEXPECTED	An unknown error occurred.
E_OUTOFMEMORY	There was not enough memory to carry out the operation, such as not being able to create a new connection point object.
CONNECT_E_NOCONNECTION	This connectable object does not support the outgoing interface specified by riid.

Comments:

E_NOTIMPL is not allowed as a return code for this member. Any implementation of *IConnectionPointContainer* must implement this method.

3 The *IEnumConnectionPoints* Interface

A connectable object can be asked to enumerate its supported connection points–in essence, it's outgoing interfaces–through IConnectionPointContainer::EnumConnectionPoints. The resulting enumerator returned from this member implements the interface IEnumConnectionPoints through which a client can access all the individual connection point sub-objects supported within the connectable object itself, where each connection point, of course, implements IConnectionPoint.

Therefore IEnumConnectionPoints is a standard enumerator interface typed for IConnectionPoint*.

IDL:

```
[
uuid(B196B285-BAB4-101A-B69C-00AA00341D07)
    , object, pointer_default(unique)
]
interface IEnumConnectionPoints : IUnknown
    {
    HRESULT Next([in] ULONG cConnections
        , [out, max_is(cConnections)] IConnectionPoint **rgpcn
        , [out] ULONG *pcFetched);
```

```
HRESULT Skip([in] ULONG cConnections);
HRESULT Reset(void);
HRESULT Clone([out] IEnumConnectionPoints **ppEnum);
}
```

.1 IEnumConnectionPoints::Next

HRESULT IEnumConnectionPoints::Next([in] ULONG cConnections , [out, max_is(cConnections)] IConnectionPoint **rgpcn, [out] ULONG *pcFetched);

Enumerates the next cConnections elements in the enumerator's list, returning them in rgpcn along with the actual number of enumerated elements in pcFetched. The caller is responsible for calling IConnectionPoint::Release through each pointer returned in rgpcn.

Argument	Type	Description
cConnections	ULONG	[in] Specfies the number of IConnectionPoint * values to return in the array pointed to by rgpcn. This argument must be 1 if pcFetched is NULL.
rgpcn	IConnectionPoint **	[out] A pointer to a caller-allocated IConnectionPoint * array of size cConnections in which to return the enumerated connection points. The caller is responsible for calling IConnectionPoint::Release through each pointer enumerated into the array once this method returns successfully. If cConnections is greater than one the caller must also pass a non-NULL pointer passed to pcFetched to know how many pointers to release.
pcFetched	ULONG	[out] A pointer to the variable to receive the actual number of connection points enumerated in *rgpcn*. This argument can be NULL in which case the *cConnections* argument must be 1.

Return Value	**Meaning**
S_OK	The requested number of elements has been returned and *pcFetched (if non-NULL) is set to cConnections if
S_FALSE	The enumerator returned fewer elements than cConnections because there were not that many elements left in the list.. In this case, unused elements in rgpcn in the enumeration are not set to NULL and *pcFetched holds the number of valid entries, even if zero is returned.
E_POINTER	The address in rgpcn is not valid (such as NULL)
E_INVALIDARG	The value of *cConnections* is not 1 when pcFetched is NULL; or the value of cConnections is zero.
E_UNEXPECTED	An unknown error occurred.
E_OUTOFMEMORY	There is not enough memory to enumerate the elements.

Comments:

E_NOTIMPL is not allowed as a return value. If an error value is returned, no entries in the rgpcn array are valid on exit and require no release.

.2 IEnumConnectionPoints::Skip

HRESULT IEnumConnectionPoints::Skip([in] ULONG cConnections); Instructs the enumerator to skip the next cConnections elements in the enumeration such that the next call to IEnumConnectionPoints::Next will not return those elements.

Argument	Type	Description
cConnections	ULONG	[in] Specifies the number of elements to skip in the enumeration.

Return Value	**Meaning**
S_OK	The number of elements skipped is cConnections.
S_FALSE	The enumerator skipped fewer than cConnections because there were not that many left in the list. The enumerator will, at this point, be positioned at the end of the list such that subsequent calls to Next (without an intervening Reset) will return zero elements.
E_INVALIDARG	The value of *cConnections* is zero, which is not valid.
E_UNEXPECTED	An unknown error occurred.

.3 IEnumConnectionPoints::Reset

HRESULT IEnumConnectionPoints::Reset(void);

Instructs the enumerator to position itself back to the beginning of the list of elements.

Argument	Type	Description
none		

Return Value	**Meaning**
S_OK	The enumerator was successfully reset to the beginning of the list.
S_FALSE	The enumerator was not reset to the beginning of the list.
E_UNEXPECTED	An unknown error occurred.

Comments:

There is no guarantee that the same set of elements will be enumerated on each pass through the list: it depends on the collection being enumerated. It is too expensive for some collections, such as files in a directory, to maintain this condition.

.4 IEnumConnectionPoints::Clone

HRESULT IEnumConnectionPoints::Clone([out] IEnumConnectionPoints **ppEnum);

Creates another connection point enumerator with the same state as the current enumerator, which iterates over the same list. This makes it possible to record a point in the enumeration sequence in order to return to that point at a later time.

Argument	Type	Description
ppEnum	IEnumConnectionPoints**	[out] The address of the variable to receive the IEnumConnectionPoints interface pointer to the newly created enumerator. The caller must release this new enumerator separately from the first enumerator.

Return Value	**Meaning**
S_OK	Clone creation succeeded.

E_NOTIMPL	Cloning is not supported for this enumerator.
E_POINTER	The address in ppEnum is not valid (such as NULL)
E_UNEXPECTED	An unknown error occurred.
E_OUTOFMEMORY	There is not enough memory to create the clone enumerator.

4 The *IEnumConnections* Interface

Any individual connection point can support enumeration of its known connections through IConnectionPoint::EnumConnections. The enumerator created by this function implements the interface IEnumConnections which deals with the type CONNECTDATA. Each CONNECTDATA structure contains the the IUnknown * of a connected sink and the dwCookie that was returned by IConnectionPoint::Advise when that sink was connected. When enumerating connections through IEnumConnections, the enumerator is responsible for calling IUnknown::AddRef through the pointer in each enumerated structure, and the caller is responsible to later call IUnknown::Release when those pointers are no longer needed.

IDL:

```
[
uuid(B196B287-BAB4-101A-B69C-00AA00341D07)
    , object, pointer_default(unique)
]
interface IEnumConnections : IUnknown
    {
    typedef struct tagCONNECTDATA
        {
        IUnknown    *pUnk;
        DWORD       dwCookie;
        } CONNECTDATA;

    typedef struct tagCONNECTDATA *PCONNECTDATA;
    typedef struct tagCONNECTDATA *LPCONNECTDATA;

    HRESULT Next([in] ULONG cConnections
        , [out, max_is(cConnections)] CONNECTDATA *rgpcd
        , [out] ULONG *pcFetched);

    HRESULT Skip([in] ULONG cConnections);
    HRESULT Reset(void);
    HRESULT Clone([out] IEnumConnections **ppEnum);
    }
```


.1 IEnumConnections::Next

HRESULT IEnumConnections::Next([in] ULONG cConnections ,
[out, max_is(cConnections)] CONNECTDATA *rgpcd,
[out] ULONG *pcFetched);

Enumerates the next cConnections elements in the enumerator's list, returning them in rgpcd along with the actual number of enumerated elements in pcFetched. The caller is responsible for calling IUnknown::Release through each pUnk pointer returned in the structure elements of rgpcd.

Argument	Type	Description
cConnections	ULONG	[in] Specfies the number of CONNECTDATA structures to return in the array pointed to by rgpcd. This argument must be 1 if pcFetched is NULL.
rgpcd	CONNECTDATA *	[out] A pointer to a caller-allocated CONNECTDATA array of size cConnections in which to return the enumerated connections. The caller is responsible for calling CONNECTDATA.pUnk->Release for each element in the array once this method returns successfully. If cConnections is greater than one the caller must also pass a non-NULL pointer passed to pcFetched to know how many pointers to release.
pcFetched	ULONG	[out] A pointer to the variable to receive the actual number of connections enumerated in rgpcd. This argument can be NULL in which case the cConnections argument must be 1.

Return Value	**Meaning**
S_OK	The requested number of elements has been returned and *pcFetched (if non-NULL) is set to *cConnections* if
S_FALSE	The enumerator returned fewer elements than cConnections because there were not that many elements left in the list.. In this case, unused elements in *rgpcd* in the enumeration are not set to NULL and *pcFetched holds the number of valid entries, even if zero is returned.
E_POINTER	The address in rgpcd is not valid (such as NULL).
E_INVALIDARG	The value of cConnections is not 1 when pcFetched is NULL; or the value of cConnections is zero.
E_UNEXPECTED	An unknown error occurred.
E_OUTOFMEMORY	There is not enough memory to enumerate the elements.

Comments:

E_NOTIMPL is not allowed as a return value. If an error value is returned, no entries in the rgpcn array are valid on exit and require no release.

.2 IEnumConnections::Skip

HRESULT IEnumConnections::Skip([in] ULONG cConnections); Instructs the enumerator to skip the next cConnections elements in the enumeration such that the next call to IEnumConnections::Next will not return those elements.

Argument	Type	Description
cConnections	ULONG	[in] Specifies the number of elements to skip in the enumeration.

Return Value	**Meaning**
S_OK	The number of elements skipped is cConnections.

S_FALSE	The enumerator skipped fewer than cConnections because there were not that many left in the list. The enumerator will, at this point, be positioned at the end of the list such that subsequent calls to Next (without an intervening Reset) will return zero elements.
E_INVALIDARG	The value in cConnections is zero which is not valid.
E_UNEXPECTED	An unknown error occurred.

.3 IEnumConnections::Reset

HRESULT IEnumConnections::Reset(void);

Instructs the enumerator to position itself back to the beginning of the list of elements.

Argument	Type	Description
none		

Return Value	**Meaning**
S_OK	The enumerator was successfully reset to the beginning of the list.
S_FALSE	The enumerator was not reset to the beginning of the list.
E_UNEXPECTED	An unknown error occurred.

Comments:

There is no guarantee that the same set of elements will be enumerated on each pass through the list: it depends on the collection being enumerated. It is too expensive for some collections, such as files in a directory, to maintain this condition.

.4 IEnumConnections::Clone

HRESULT IEnumConnections::Clone([out] IEnumConnections **ppEnum);

Creates another connections enumerator with the same state as the current enumerator, which iterates over the same list. This makes it possible to record a point in the enumeration sequence in order to return to that point at a later time.

Argument	Type	Description
ppEnum	IEnumConnections**	[out] The address of the variable to receive the IEnumConnections interface pointer to the newly created enumerator. The caller must release this new enumerator separately from the first enumerator.

Return Value	**Meaning**
S_OK	Clone creation succeeded.
E_NOTIMPL	Cloning is not supported for this enumerator.
E_POINTER	The address in ppEnum is not valid (such as NULL)
E_UNEXPECTED	An unknown error occurred.
E_OUTOFMEMORY	There is not enough memory to create the clone enumerator.

10. Persistent Storage

In order to reduce the overall size of this document, and because the topic of this chapter is fully specified in the Microsoft Win32 Software Development Kit, the text of this chapter has been omitted.

This page intentionally left blank.

11. Persistent Intelligent Names: Monikers

1 Overview

A moniker is simply an object that supports the IMoniker interface. IMoniker interface includes the IPersistStream interface; thus, monikers can be saved to and loaded from streams. The persistent form of a moniker contains the class identifier (CLSID) of its implementation which is used during the loading process, and so new kinds of monikers can be created transparently to clients.

The most basic operation in IMoniker interface is that of *binding* to the object to which it points, which is supported by IMoniker::BindToObject. This function takes as a parameter the interface identifier by which the caller wishes to talk to the object, runs whatever algorithm is necessary in order to locate the object, then returns a pointer of that interface type to the caller.[95] Each moniker class can store arbitrary data its persistent representation, and can run arbitrary code at binding time.

If there is an identifiable piece of persistent storage in which the object referenced by the moniker is stored, then IMoniker::BindToStorage can be used to gain access to it. Many objects have such identifiable storage, but some, such as the objects which are the ranges on a Microsoft Excel spreadsheet do not. (These ranges exist only as a part of Excel's data structures; they are in effect a figment of Excel's imagination and are only reified on demand for clients.)

In most cases, a particular moniker class is designed to be one step along the path to the information source in question. These pieces can be *composed* together to form a moniker which represents the complete path. For example, the moniker stored inside a chart that refers to its underlying data in a spreadsheet might be a composite moniker formed from three pieces:

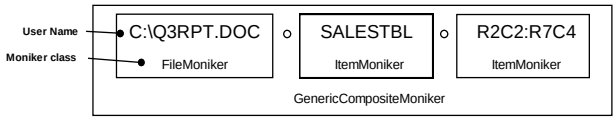


Figure 2. Moniker in a chart referring to a spreadsheet from which it extracts data.

This composite is *itself* a moniker; it just happens to be a moniker which is a sequenced collection of other monikers. The composition here is *generic* in that it has no knowledge of the pieces involved *other* than that they are monikers.

Most monikers have a textual representation which is meaningful to the user; this can be retrieved with IMoniker::GetDisplayName. The API function MkParseDisplayName goes the other direction: it can turn a textual display name into the appropriate moniker, though beware that in general this is operation is as expensive as actually binding to the object.

Monikers can compare themselves to other monikers using IMoniker::IsEqual. A hash value useful for storing monikers in lookup tables is available through IMoniker::Hash. Monikers are not a total order or even a partial order; therefore, monikers cannot be stored in tables that rely on sorting for retrieval; use hashing instead (it is inappropriate to use the display name of a moniker for sorting, since the display name may not reflect the totality of internal state of the moniker).

The earliest time after which the object to which the moniker points is known not to have changed can be obtained with IMoniker::GetTimeOfLastChange. This is *not* necessarily the time of last change of the object; rather, it is the best cheaply available approximation thereto.

A moniker can be asked to re-write itself into another equivalent moniker by calling IMoniker::Reduce. This function returns a new moniker that will bind to the same object, but does so in a more efficient way. This capability has several uses:

- It enables the construction of user-defined macros or aliases as new kinds of moniker classes. When reduced, the moniker to which the macro evaluates is returned.
- It enables the construction of a kind of moniker which tracks data as it moves about. When reduced, the moniker of the data in its current location is returned.

[95] This function also takes some parameters that provide contextual information to the binding process which we shall get to in a moment.

- On file systems such as Macintosh System 7 which support an ID-based method of accessing files which is independent of file names, a File Moniker could be reduced to a moniker which contains one of these IDs.

Figure 3 shows a (somewhat contrived) example of moniker reduction. It illustrates the reduction of a moniker which names the net income entry for this year's report in the "Projects" directory of the current user's home directory.

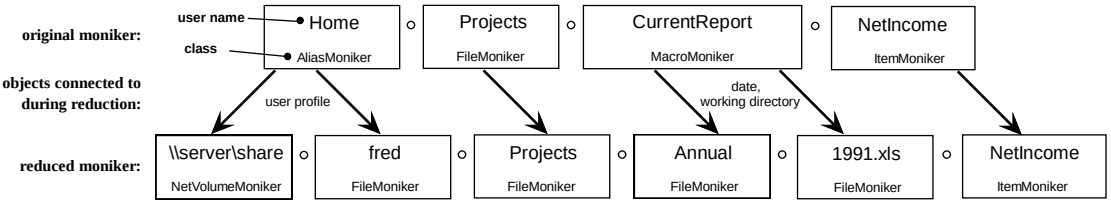


Figure 3. Reduction of a moniker showing the objects connected to during reduction.

(Note that the particular classes of monikers used here are for illustrative purposes only.) As we can see, many monikers in this example are reduced to something completely different, and some bind to something during their reduction, but some do not. For example, to reduce the alias "Home", the reduction must access the information that "Home" was an alias for "\\server\share\fred".

The process of moniker reduction may also be tied to a global table called the *Running Object Table*. The Running Object Table serves as the place where monikers in the process of binding look to see if they are already running or not.

Pointers to instances of IMoniker interface can be marshaled to other processes, just as any other interface pointer can. Many monikers are of the nature that they are immutable once created and that they maintain no object state outside themselves. Item Monikers are an example of a class of such monikers. These monikers, which can be replicated at will, will usually want to support custom marshaling (see IMarshal interface) so as to simply serialize themselves and de-serialize themselves in the destination context (see IPersistStream regarding serialization). This is referred to as marshaling an object *by value*.

2 IMoniker interface and Core Monikers

This section describes the details of IMoniker interface and related interfaces. In addition, it discusses the various kinds of monikers that are provide as part of every COM implementation.

Some moniker errors have associated with them some extended information. See IBindCtxt::RegisterObjectParam for more details.

.1 IMoniker interface

We'll now look in detail at IMoniker interface its supporting functions and structures.

```
interface IMoniker : IPersistStream {
    HRESULT     BindToObject(pbc, pmkToLeft, iidResult, ppvResult);
    HRESULT     BindToStorage(pbc, pmkToLeft, iid, ppvObj);
    HRESULT     Reduce(pbc, dwReduceHowFar, ppmkToLeft, ppmkReduced);
    HRESULT     ComposeWith(pmkRight, fOnlyIfNotGeneric, ppmkComposite)
    HRESULT     Enum(fForward, ppenmMoniker);
    HRESULT     IsEqual(pmkOtherMoniker);
    HRESULT     Hash(pdwHash);
    HRESULT     IsRunning(pbc, pmkToLeft, pmkNewlyRunning);
    HRESULT     GetTimeOfLastChange(pbc, pmkToLeft, pfiletime);
    HRESULT     Inverse(ppmk);
    HRESULT     CommonPrefixWith(pmkOther, ppmkPrefix);
    HRESULT     RelativePathTo(pmkOther, ppmkRelPath);
    HRESULT     GetDisplayName(pbc, pmkToLeft, lplpszDisplayName);
    HRESULT     ParseDisplayName(pbc, pmkToLeft, lpszDisplayName, pcchEaten, ppmkOut);
    HRESULT     IsSystemMoniker(pdwMksys);
    };

HRESULT BindMoniker(pmk, reserved, iidResult, ppvResult);
HRESULT CreateBindCtx(reserved, ppbc);
```



```
HRESULT MkParseDisplayName(pbc, lpszDisplayName, pcchEaten, ppmk);
interface IParseDisplayName : IUnknown {
    HRESULT     ParseDisplayName(pbc, lpszDisplayName, pcchEaten, ppmkOut);
    };

HRESULT CreateGenericComposite(pmkFirst, pmkRest, ppmkComposite);
HRESULT CreateFileMoniker(lpszPathName, ppmk);
HRESULT CreateItemMoniker(lpszDelim, lpszItem, ppmk);
HRESULT CreateAntiMoniker(ppmk);
HRESULT CreatePointerMoniker(punk, ppmk);
```

.1 IMoniker::BindToObject

HRESULT IMoniker::BindToObject(pbc, pmkToLeft, iidResult, ppvResult)

This is the workhorse function in IMoniker interface. Locate and load the object semantically referred to by this moniker according to the interface indicated by iidResult and return the object through ppvResult. After this call has returned, the semantics of the returned interface, whatever they are, should be fully functional.

In general, each kind of moniker is designed to be used as one piece in a composite which gives the complete path to the object in question. In this composite, any given piece has a certain prefix of the composite to its left, and a certain suffix to its right. If IMoniker::BindToObject is invoked on the given piece, then most often the implementation of IMoniker::BindToObject will require certain services of the object indicated by the prefix to its left. Item monikers, for example, require IOleItemContainer interface of the object to their left; see below. The Item Moniker implementation of IMoniker::BindToObject recursively calls pmkToLeft->BindToObject in order to obtain this interface. Other implementations of IMoniker::BindToObject might instead invoke pmkToLeft->BindToStorage if they need access not to the object itself, but to its persistent storage.

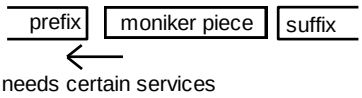


Figure 4. Interface calculus of moniker pieces

In situations where the caller of IMoniker::BindToObject does not have a moniker for the object on the left, but instead has the object itself, a Pointer Moniker can be used to wrap the object pointer so that the moniker may be bound.

In situations where the moniker in fact does *not* need services of the moniker to its left, yet one is provided by the caller nevertheless, *no* error should occur; the moniker should simply ignore the needless moniker to its left.

If the object indicated by the moniker does not exist, then the error MK_E_NOOBJECT is returned.

In general, binding a moniker can be quite a complicated process, since it may need to launch servers, open files, etc. This often may involve binding to other objects, and it is often the case that binding pieces of the composite to the right of the present piece will require the same other objects. In order to avoid loading the object, releasing it, then having it loaded again later, IMoniker::BindToObject can use the *bind context* passed through the pbc parameter in order to defer releasing the object until the binding process overall is complete. See IBindCtx::RegisterObjectBound for details.

The bind context also contains a deadline time by which the caller would like the binding process to complete, or fail with the error MK_E_EXCEEDEDDEADLINE if it cannot. This capability is not often used with IMoniker::BindToObject; it is more often used with other IMoniker functions such as IMoniker::GetTimeOfLastChange. Nevertheless, IMoniker::BindToObject implementations should (heuristically) honor the request. See IBindCtx::GetBindOptions for details.

Usually, for most monikers, binding a second time will return the same running object as binding the first time, rather than reloading it again from passive backing store. This functionality is supported with the Running Object Table, which is described in detail later in this chapter. Basically, the Running Object Table is a lookup table keyed by a moniker whose values are pointers to the corresponding now-running object. As objects become running, the register themselves in this table. Implementations of IMoniker::BindToObject can use this table to shortcut the binding process if the object to which they point is already running. More precisely, if the passed pmkToLeft parameter is NULL (and this is not an error; that is, the moniker does not *require* something to its left), then the moniker should fully reduce itself, then look itself up in the Running Object Table and simply return the pointer to the object found there. If the

pmkToLeft parameter is non-NULL, then it is the responsibility of the caller to handle this situation; the BindToObject() implementation should *not* consult the Running Object Table.[96] The Running Object Table is accessible from the bind context using IBindCtx::GetRunningObjectTable, an implementation of IMoniker::BindToObject should not use GetRunningObjectTable().

Argument	Type	Description
pbc	IBindCtx*	the bind context to be used for this binding operation.
pmkToLeft	IMoniker*	the moniker of the object to the left of this moniker.
iidResult	REFIID	the interface by which the caller wishes to connect to the object.
ppvResult	void**	on successful return, a pointer to the instantiated object is placed here, unless BINDFLAGS_JUSTTESTEXISTENCE was specified in the binding options, in which case NULL *may* be returned instead.
return value	HRESULT	S_OK, MK_E_NOOBJECT, STG_E_ACCESSDENIED, MK_E_EXCEEDED-DEADLINE, MK_E_CONNECTMANUALLY, MK_E_INTERMEDIATEINTER-FACENOTSUPPORTED, E_OUTOFMEMORY, E_NOINTERFACE

.2 BindMoniker

HRESULT BindMoniker(pmk, reserved, iidResult, ppvResult)

Bind a moniker with the specified interface and return the result. This is strictly a helper function in that it uses no functionality which is not also available publicly. It has roughly the following implementation:

```
IBindCtx pbc;
CreateBindCtx(0, &pbc);
pmk->BindToObject(pbc, NULL, iidResult, ppvResult);
pbc->Release();
```

Argument	Type	Description
pmk	IMoniker*	the moniker which is to be bound.
reserved	DWORD	reserved for future use; must be zero.
iidResult	REFIID	the interface by which the caller wishes to connect to the object.
ppvResult	void**	on successful return, a pointer to the resulting object is placed here.
return value	HRESULT	S_OK, union of IMoniker::BindToObject() & CreateBindCtx() errors

.3 IMoniker::BindToStorage

HRESULT IMoniker::BindToStorage(pbc, pmkToLeft, iid, ppvObj)

Return access to the persistent *storage* of the receiver using the given interface, rather than access to the object itself, which is what IMoniker::BindToObject returns. Consider, for example, a moniker which refers to spreadsheet embedded in a word processing document, such as:

$$[\text{c:\foo\bar.doc}]_{\text{File Moniker}} \; ? \; [\text{summaryTable}]_{\text{Item Moniker}}$$

Calling IMoniker::BindToObject on this composite will enable us to talk to the spreadsheet; calling IMoniker::BindToStorage will let us to talk to the IStorage instance in which it resides.

IMoniker::BindToStorage will most often be called during the right-to-left recursive process of IMoniker::-BindToObject invoked on a Generic Composite Moniker. Sometimes it is the case that monikers in the tail of the composite don't require access to the object on their left; they merely require access to its persistent storage. In effect, these monikers can be bound to without also binding to the objects of the monikers to their left, potentially a much more efficient operation.

Some objects do not have an independently identifiable piece of storage. These sorts of objects are really only a object-veneer on the internal state of their container. Examples include named cell ranges inside an Excel worksheet, and fragments of a Windows Word document delimited by bookmarks. Attempting to call

[96] The reason behind this rule lies in the fact that in order to look in the Running Object Table, we need the whole moniker in its fully reduced form. If the current moniker is but a piece of a generic composite, then it has to be the composite's responsibility for doing the reduction; the moniker cannot do it correctly do it by itself.

IMoniker::BindToStorage on a moniker which indicates one of these kinds of objects will fail with the error MK_E_NOSTORAGE.

Use of the bind context in IMoniker::BindToStorage is the same as in IMoniker::BindToObject.

Argument	Type	Description
pbc	IBindCtx*	the binding context for this binding operation.
iid	REFIID	the interface by which we wish to bind to this storage. Common interfaces passed here include IStorage, IStream, and ILockBytes.
ppvObj	void**	On successful return, a pointer to the instantiated storage is placed here, unless BINDFLAGS_JUSTTESTEXISTENCE was specified in the binding options, in which case NULL *may* be returned instead.
return value	HRESULT	S_OK, MK_E_NOSTORAGE, MK_E_EXCEEDEDDEADLINE, MK_E_CONNECTMANUALLY, E_NOINTERFACE, MK_E_INTERMEDIATEINTERFACENOTSUPPORTED, STG_E_ACCESSDENIED

.4 IMoniker::Reduce

HRESULT IMoniker::Reduce(pbc, dwReduceHowFar, ppmkToLeft, ppmkReduced)

The reduction of monikers was reviewed and illustrated in the synopsis above; this is the function that actually carries it out. Return a more efficient or equally efficient moniker that refers to the same object as does this moniker. Many monikers, if not most, will simply reduce to themselves, since they cannot be rewritten any further. A moniker which reduces to itself indicates this by returning itself through ppmkReduced and the returning status code MK_S_REDUCED_TO_SELF. A moniker which reduces to nothing should return NULL, and should return the status code S_OK.

If the moniker does not reduce to itself, then this function does *not* reduce this moniker in-place; instead, it returns a *new* moniker.

The reduction of a moniker which is a composite of other monikers repeatedly reduces the pieces of which it is composed until they all reduce to themselves, then returns the composite of the reduced pieces. dwReduceHowFar controls the stopping point of the reduction process. It controls to what extent the reduction should be carried out. It has the following legal values.

```
typedef enum tagMKRREDUCE {
    MKRREDUCE_ONE           = 3<<16,
    MKRREDUCE_TOUSER        = 2<<16,
    MKRREDUCE_THROUGUSER    = 1<<16,
    MKRREDUCE_ALL               = 0
    } MKRREDUCE;
```

These values have the following semantics.

Value	Description
MKRREDUCE_ONE	Perform only one step of reduction on this moniker. In general, the caller will have to have specific knowledge as to the particular kind of moniker in question in order to be able to usefully take advantage of this option.
MKRREDUCE_TOUSER	Reduce this moniker to the first point where it first is of the form where it represents something that the user conceptualizes as being the identity of a persistent object. For example, a file name would qualify, but a macro or an alias would not. If no such point exists, then this option should be treated as MKRREDUCE_ALL.
MKRREDUCE_THROUGUSER	Reduce this moniker to the point where any further reduction would reduce it to a form which the user does not conceptualize as being the identity of a persistent object. Often, this is the same stage as MKRREDUCE_TOUSER.
MKRREDUCE_ALL	Reduce the entire moniker, then, if needed reduce it again and again to the point where it reduces to simply itself.

When determining whether they have reduced themselves as far as requested, IMoniker::Reduce implementations should not compare for equality against dwReduceHowFar, as we wish to allow for the possibility that

intermediate levels of reduction will be introduced in the future. Instead, IMoniker::Reduce implementations should reduce themselves *at least* as far as is requested.

An important concept in the above is the idea of a moniker that the user thinks of as the name of a persistent object; a persistent identity. The intent is to provide the ability to programmatically reduce a moniker to canonical forms whose display names would be recognizable to the user. Paths in the file system, bookmarks in word-processing documents, and range names in spreadsheets are all examples of user-identities. In contrast, neither a macro nor an alias encapsulated in a moniker, nor an inode-like file ID moniker are such identities.

The bind context parameter is used as in IMoniker::BindToObject. In particular, implementations of IMoniker::Reduce should pay attention to the time deadline imposed by the caller and the reporting of the moniker of the object that, if it had been running, would have allowed the reduction to progress further. See IBindCtx below.

Argument	Type	Description
pbc	IBindCtx*	The bind context to use in this operation.
dwReduceHowFar	DWORD	Indicates to what degree this moniker should be reduced; see above.
ppmkToLeft	IMoniker**	On entry, the moniker which is the prefix of this one in the composite in which it is found. On exit, the pointer is either NULL or non-NULL. Non-NULL indicates that what was previously thought of as the prefix should be disregarded and the moniker returned through ppmkToLeft considered the prefix in its place (this is very rare). NULL indicates that the prefix should not be so replaced. Thus, most monikers will NULL out this parameter before returning.
ppmkReduced	IMoniker**	On exit, the reduced form of this moniker. Possibly NULL.
return value	HRESULT	S_OK, MK_S_REDUCED_TO_SELF, MK_E_EXCEEDEDDEADLINE.

.5 IMoniker::ComposeWith

HRESULT IMoniker::ComposeWith(pmkRight, fOnlyIfNotGeneric, ppmkComposite)

Return a new moniker which is a composite formed with this moniker on the left and pmkRight on the right. It is using this operation that the pieces of the path to an object are cobbled together to form the overall full path.

There are two distinct kinds of composite monikers: those that know nothing about their pieces other than that they are monikers, and those that know more. We have been terming the former a *generic* composite; we have seen several examples above. An example of the latter might be that of the result of composing a File Moniker containing a relative path on to the end of another File Moniker: the result could be a new File Moniker containing the complete path.[97] There is only a need for one implementation of a Generic Composite Moniker, and this has been provided; see CreateGenericComposite(). Non-generic composition is useful for monikers that are capable of collapsing a path within a storage domain to a more efficient representation in a subsequent IMoniker::Reduce operation. None of the core monikers on Win32 are capable of this, but an implementation of File Moniker which could collapse down to a inode-like file ID might be an example of such a behavior.

Each moniker class in general will have a (possibly empty) set of other kinds of special monikers that can be composed onto the end of it in a non-generic way; the moniker class has some sort of intimate knowledge about the semantics of these special monikers, more than simply that they are monikers. Each IMoniker::ComposeWith implementation will examine pmkRight to see if it is such a special moniker for this implementation. Often, it will ask pmkRight for its class, but other possibilities exist, such as using QueryInterface. A very common case of such special monikers are Anti Monikers.

If pmkRight is special, then the IMoniker::ComposeWith implementation does whatever is appropriate for that special case. If it is not, then fOnlyIfNotGeneric controls what should occur. If fOnlyIfNotGeneric is true, then NULL should be passed back through ppmkComposite and the status MK_E_NEEDGENERIC returned; if

[97] In fact, the Win32 implementation of File Monikers does have this behavior. An alternative to the non-generic composition implementation described here is that the elements in a path are each separate monikers which are then *generically* composed together.

fOnlyIfNotGeneric is false, then a generic composite should be returned using CreateGenericComposite. Most callers of IMoniker::ComposeWith should set fOnlyIfNotGeneric to false.[98]

In any situation that pmkRight completely annihilates the receiver (i.e.: irrespective of fOnlyIfNotGeneric), and so the resulting composite is empty, NULL should be passed back through ppmkComposite and the status S_OK returned.

The pieces of a moniker that have been composed together can be picked apart using IMoniker::Enum. On a generic composite, this enumerates the monikers contained within it. On other monikers, which particular pieces are returned is implementation-defined.

Composition of monikers is an associative operation. That is, if A, B, and C are monikers, then

(A ∘ B) ∘ C

is always equal to

A ∘ (B ∘ C)

where ∘ represents the composition operation. Each implementation of IMoniker::ComposeWith must maintain this invariant.

Argument	Type	Description
pmkRight	IMoniker*	the moniker to compose onto the end of the receiver.
fOnlyIfNotGeneric	BOOL	controls what should be done in the case that the way for form a composite is to use a generic one.
ppmkComposite	IMoniker*	on exit, the resulting composite moniker. Possibly NULL.
return value	HRESULT	S_OK, MK_E_NEEDGENERIC

.6 IMoniker::Enum

HRESULT IMoniker::Enum(fForward, ppenmMoniker)

Enumerate the monikers of which the receiver is logically a composite. On a generic composite, this enumerates the pieces of which the composite is composed. On other monikers, the semantics of the pieces of which it is a composite are implementation-defined. For example, enumerating the pieces of a File Moniker might pick apart the internally stored path name into its components, even though they are not stored internally as actual separate monikers. Many monikers have no discernible internal structure; they will simply pass back NULL instead of an enumerator.

IEnumMoniker is an enumerator that supports the enumeration of items which are monikers. It is defined as:

```
typedef Enum<IMoniker*> IEnumMoniker;
```

which is shorthand for

```
interface IEnumMoniker : IUnknown {
    virtual HRESULT Next(ULONG celt, IMoniker* rgelt[], ULONG* pceltFetched);
    virtual HRESULT Skip(ULONG celt);
    virtual HRESULT Reset();
    virtual HRESULT Clone(IEnumMoniker** ppenm);
    };
```

[98] fOnlyIfNotGeneric is set by recursive ComposeWith() calls from the implementation of Generic Composite Moniker - ComposeWith().

Argument	Type	Description
fForward	BOOL	If true, then the enumeration should be done in the normal order. If false, then the order should be the reverse of the order enumerated by the normal order.
ppenmMoniker	IEnumMoniker**	On exit, the returned enumerator. May be NULL, signifying that there is nothing to enumerate.
return value	HRESULT	S_OK.

.7 IMoniker::IsEqual

HRESULT IMoniker::IsEqual(pmkOtherMoniker)

The most important use of this function is in the implementation of the Running Object Table. As discussed in detail later, this table has two distinct but closely related roles. First, using a moniker, entries in the Running Object Table indicate those objects which are presently now logically running and to which auto-link reconnections should be made. Second, for those objects which are actually running (have an object pointer), it provides a means given their moniker to actually connect to the appropriate running object.

The moniker implementation should *not* reduce itself before carrying out the compare operation.

Two monikers which can compare as equal in either order must hash to the same value; see IMoniker::Hash.

Argument	Type	Description
pmkOtherMoniker	IMoniker*	the other moniker with whom we would like to compare the receiver.
return value	HRESULT	S_OK, S_FALSE

.8 IMoniker::Hash

HRESULT IMoniker::Hash(pdwHash)

Return a 32 bit integer associated with this moniker. This integer is useful for maintaining tables of monikers: the moniker can be hashed to determine a hash bucket in the table, then compared with IMoniker::IsEqual against all the monikers presently in that hash bucket.

It must always be the case that two monikers that compare as equal in either order hash to the same value. In effect, implementations of IMoniker::IsEqual() and IMoniker::Hash are intimate with one another; they must always be written together.

The value returned by IMoniker::Hash is invariant under marshaling: if a moniker is marshaled to a new context, then IMoniker::Hash invoked on the unmarshaled moniker in the new context must return the same value as IMoniker::Hash invoked on the original moniker. This is the only way that a global table of monikers such as the Running Object Table can be maintained in shared space, yet accessed from many processes. The obvious implementation technique this indicates is that IMoniker::Hash should not rely on the memory address of the moniker, but only its internal state.

Argument	Type	Description
pdwHash	DWORD*	the place in which to put the returned hash value.
return value	HRESULT	S_OK

.9 IMoniker::IsRunning

HRESULT IMoniker::IsRunning(pbc, pmkToLeft, pmkNewlyRunning)

Answer as to whether this moniker is in fact running. As usual, the Running Object Table in whose context this question is to be answered is obtained by this moniker from the Bind context. pmkToLeft is the moniker to the left of this object in the generic composite in which it is found, if any.

If non-NULL, pmkNewlyRunning is the moniker which has most recently been added to the Running Object Table; the implementation of IMoniker::IsRunning can assume that without this object in the Running Object Table, that IMoniker::IsRunning would have reported that it was not running; thus, the only way that it can now be running is if this newly running moniker is in fact itself! This allows for some n^2-to-n reductions in

algorithms that use monikers. (If the moniker implementation chose to ignore pmkNewlyRunning, no harm would come: this moniker is in fact in the Running Object Table)

Implementations of this method in various kinds of moniker classes are roughly as follows:

Generic Composite Moniker

```
if (pmkToLeft != NULL)
    return (pmkToLeft->ComposeWith(this)) -> IsRunning(pbc, NULL, pmkNewlyRunning);
if (pmkNewlyRunning != NULL) {
    if (pmkNewlyRunning -> IsEqual(this) == NOERROR)
        return NOERROR;
    }
else if (pRunningObjectTable -> IsRunning(this) == NOERROR)
    return NOERROR;
// otherwise, forward it on to my last element.
return this->Last()->IsRunning(pbc, this->AllButLast(), pmkNewlyRunning)
```

Any moniker whose class does not do any wildcard matching

```
if (pmkToLeft == NULL) {
    if (pmkNewlyRunning != NULL)
        return pmkNewlyRunning -> IsEqual(this);
    else
        return pRunningObjectTable -> IsRunning(this);
    }
else
    return ResultFromScode(S_FALSE);    // If I was running, then Generic Composite would have caught it.
```

A moniker class which has a wild card entry which always matches any instance of the moniker class: if the wild card is present, then all instances of the moniker class to the right of the same other moniker (that is, with the same moniker to their left) are deemed to be running. Such a moniker class might be reasonably used, for example, to match all the addressable ranges in a given spreadsheet.

```
if (pmkToLeft == NULL) {
    if (pmkNewlyRunning != NULL)
        return pmkNewlyRunning->IsEqual(this) == NOERROR
            || pmkNewlyRunning->IsEqual(my wild card moniker) == NOERROR;
    if (pRunningObjectTable -> IsRunning(this) == NOERROR)
        return NOERROR;
    return pRunningObjectTable -> IsRunning(my wild card moniker);
    }
else
    return pmkToLeft->ComposeWith(my wild card moniker) -> IsRunning(pbc, NULL, pmkNewlyRunning);
```

A moniker class which has a wild card entry which matches against some of the objects, but only the ones which are in fact actually currently running. We illustrate here specifically the behaviour of Item Monikers.

```
if (pmkToLeft == NULL) {
    if (pmkNewlyRunning != NULL) {
        if (pmkNewlyRunning->IsEqual(this) == NOERROR)
            return NOERROR;
        if (pmkNewlyRunning->IsEqual(my wild card moniker) != NOERROR)
            return ResultFromScode(S_FALSE);
        goto TestBind:
        }
    }
if (pmkToLeft->ComposeWith(my wild card moniker)->IsRunning(pbc, NULL, pmkNewlyRunning) != NOERROR)
    return ResultFromScode(S_FALSE);
TestBind:
// In general, connect to the container and ask whether the object is running. The use of
// IOleItemContainer here is Item Moniker-specific, but the theme is a general one.
IOleItemContainer *pcont;
pmkToLeft->BindToObject(pbc, NULL, IID_IOleItemContainer, &pcont);
return pcont->IsRunning(szItemString);
```

The arguments to this function are as follows:

Argument	Type	Description
pbc	IBindCtx*	the usual bind context
pmkToLeft	IMoniker*	the moniker to the left of this one in the composite in which it is found.
pmkNewlyRunning	IMoniker*	may be NULL. If non-NULL, then this is the moniker which has been most recently added to the Running Object Table. In this case, IMoniker::IsRunning implementations may assume that without this moniker in the R.O.T. that IMoniker::IsRunning would return S_FALSE.
return value	HRESULT	S_OK, S_FALSE

.10 IMoniker::GetTimeOfLastChange

HRESULT IMoniker::GetTimeOfLastChange(pbc, pmkToLeft, pfiletime)

Answer the earliest time after which the object pointed to by this moniker is known not to have changed.

The purpose of this function is to support the ability to determine whether higher-level objects based on monikers are up-to-date or not. An example of higher level objects are the link objects in OLE Compound Documents.

The returned time of change is reported using a FILETIME. A FILETIME is a 64-bit value indicating a time in units of 100 nanoseconds, with an origin in 1601.[99] A resolution of 100 nanoseconds allows us to deal with very fast-changing data; allocating this many bits gives us a range of tens of thousands of years. It is not expected that most change times in objects will be actually be internally recorded with this precision and range; they only need be reported with such.

If the time of last change is unavailable, either because the deadline was exceeded or otherwise, then it is recommended that a FILETIME of {dwLowDateTime,dwHighDateTime} = {0xFFFFFFFF,0x**7**FFFFFFF} (note the 0x7 to avoid accidental unsigned / signed confusions) should be passed back. If the deadline was exceeded, then the status MK_E_EXCEEDEDDEADLINE should be returned. If the time of change is unavailable, and would not be available no matter what deadline were used, then MK_E_UNAVAILABLE should be returned. Otherwise, S_OK should be returned.

If pmkToLeft is NULL, then this function should generally first check for a recorded change-time in the Running Object Table with IRunningObjectTable::GetTimeOfLastChange before proceeding with other strategies. Moniker classes that support wildcards will have to take into consideration exactly what does get put in the Running Object Table and look for the appropriate thing; since Generic Composite Monikers know nothing of wildcards, they may even need to do that in the non-NULL pmkToLeft case. See IMoniker::IsRunning.

Argument	Type	Description
pbc	IBindCtx*	the binding context for this operation.
pmkToLeft	IMoniker*	the moniker to the left of this one in the composite in which it is found.
pfiletime	FILETIME*	the place in which the time of last change should be reported.
return value	HRESULT	S_OK, MK_E_EXCEEDEDDEADLINE, MK_E_UNAVAILABLE, MK_E_CONNECTMANUALLY

.11 IMoniker::Inverse

HRESULT IMoniker::Inverse(ppmk)

Answer a moniker that when composed onto the end of this moniker or one of similar structure will annihilate it; that is, will compose to nothing. IMoniker::Inverse will be needed in implementations of IMoniker::RelativePathTo, which are important for supporting monikers that track information as it moves about.

This is the abstract generalization of the ".." operation in traditional file systems. For example a File Moniker which represented the path "a\b\c\d" would have as its inverse a moniker containing the path "..\..\..\..", since "a\b\c\d" composed with "..\..\..\.." yields nothing.

[99] The definition of FILETIME was taken from the Microsoft Windows–32 specification.

Notice that an the inverse of a moniker does not annihilate just that particular moniker, but all monikers with a similar structure, where structure is of course interpreted with respect to the particular moniker. Thus, the inverse of a Generic Composite Moniker is the reverse composite of the inverse of its pieces. Monikers which are non-generic composites (such as File Monikers are presently implemented) will also have non-trivial inverses, as we just saw. However, there will be many kinds of moniker whose inverse is trivial: the moniker *adds* one more piece to an existing structure; its inverse is merely a moniker that *removes* the last piece of the existing structure. A moniker that when composed onto the end of a generic moniker removes the last piece is provided; see CreateAntiMoniker. Monikers with no internal structure can return one of these as their inverse.

Not all monikers have inverses. The inverse of an anti-moniker, for example, does not exist. Neither will the inverses of most monikers which are themselves inverses. It is conceivable that other monikers do not have inverses as well; a macro moniker might be an example. Monikers which have no inverse cannot have relative paths formed from things inside the objects they denote to things outside.

Argument	Type	Description
ppmk	IMoniker**	the place to return the inverse moniker.
return value	HRESULT	S_OK, MK_E_NOINVERSE.

.12 IMoniker::CommonPrefixWith

HRESULT IMoniker::CommonPrefixWith(pmkOther, ppmkPrefix)

Answer the longest common prefix that the receiver shares with the moniker pmkOther. This functionality is useful in constructing relative paths, and for performing some of the calculus on monikers needed by the **Edit / Links** dialog in OLE Documents scenarios.

Argument	Type	Description
pmkOther	IMoniker*	the moniker with whom we are determine the common prefix.
ppmkPrefix	IMoniker*	the place to return the common prefix moniker. NULL is returned only in the case that the common prefix does not exist.
return value	HRESULT	MK_S_ME, indicating that the receiver as a whole is the common prefix. MK_S_HIM, indicating that pmkOther as a whole is the common prefix. MK_S_US, indicating that in fact the two monikers are equal. S_OK, indicating that the common prefix exists but is neither the receiver nor pmkOther. MK_S_NOPREFIX indicating that no common prefix exists.

.13 MonikerCommonPrefixWith

HRESULT MonikerCommonPrefixWith(pmkThis, pmkOther, ppmkPrefix)

This function is intended solely for the use of moniker *implementors;* clients of monikers "need not apply;" clients should instead compute the common prefix between two monikers by using

```
pmkSrc->CommonPrefixWith(pmkOther, ppmkPrefix);
```

Implementations of IMoniker::CommonPrefixWith necessarily call MonikerCommonPrefixWith as part of their internal processing. Such a method should first check to see if the other moniker is a type that it recognizes and handles specially. If not, it should call MonikerCommonPrefixWith, passing itself as pmkSrc and the other moniker as pmkDest. MonikerCommonPrefixWith will handle the generic composite cases correctly.

Argument	Type	Description
pmkThis	IMoniker*	the starting moniker for the computation of the relative path.
pmkOther	IMoniker*	the moniker to which a relative path should be taken.
ppmkPrefix	IMoniker**	May *not* be NULL. The place at which the moniker of pmkDest relative to pmkSrc is to be returned.
return value	HRESULT	S_OK, MK_S_HIM, MK_S_ME, MK_S_US, MK_S_NOPREFIX

.14 IMoniker::RelativePathTo

HRESULT IMoniker::RelativePathTo(pmkOther, ppmkRelPath)

Answer a moniker that when composed onto the end of this one or one with a similar structure will yield pmkOther. Conceptually, implementations of this function usually work as follows: the longest prefix that the receiver and pmkOther have is common is determined. This breaks the receiver and pmkOther each into two parts, say (P,T_{me}) and (P,T_{him}) respectively, where P is the maximal common prefix. The correct relative path result is then $T_{me}^{-1} \circ T_{him}$.

For any given implementation of this function, it is usually the case that the same pmkOther monikers are treated specially as would be in IMoniker::ComposeWith(). File Monikers, for example, might treat other File Monikers specially in both cases.

See also MonikerRelativePathTo().

Argument	Type	Description
pmkOther	IMoniker*	the moniker to which a relative path should be taken.
ppmkRelPath	IMoniker*	May *not* be NULL. The place at which the relative path is returned.
return value	HRESULT	MK_S_HIM, indicating that the only form of relative path is in fact just the other moniker, pmkOther. S_OK, indicating that a non-trivial relative path exists.

.15 MonikerRelativePathTo

HRESULT MonikerRelativePathTo(pmkSrc, pmkDest, ppmkRelPath, reserved)

This function is intended solely for the use of moniker *implementors;* clients of monikers "need not apply;" clients should instead compute the relative path between two monikers by using

```
pmkSrc->RelativePathTo(pmkDest, ppmkRelPath);
```

Implementations of IMoniker::RelativePathTo necessarily call MonikerRelativePathTo as part of their internal processing. Such a method should first check to see if the other moniker is a type that it recognizes and handles specially. If not, it should call MonikerRelativePathTo, passing itself as pmkSrc and the other moniker as pmkDest. MonikerRelativePathTo will handle the generic composite cases correctly.

Argument	Type	Description
pmkSrc	IMoniker*	the starting moniker for the computation of the relative path.
pmkDest	IMoniker*	the moniker to which a relative path should be taken.
ppmkRelPath	IMoniker**	May *not* be NULL. The place at which the moniker of pmkDest relative to pmkSrc is to be returned.
reserved	BOOL	must be *non-zero (**NOTE!**)*
return value	HRESULT	S_OK, MK_S_HIM

.16 IMoniker::GetDisplayName

HRESULT IMoniker::GetDisplayName(pbc, pmkToLeft, lplpszDisplayName)

Most monikers have a textual representation which is meaningful to a human being. This function returns the current display name for this moniker, or NULL if none exists.

Some display names may change over time as the object to which the moniker refers moves about in the context in which it lives. Formula references between two Microsoft Excel spreadsheets are an example of

this type of changing reference. A formula referring to cell "R1C1" in another sheet may change to the refer to "R2C1" if a new row is inserted at the top of the second sheet: the reference still refers to the same actual cell, but now the cell has a different address in its sheet. This behavior leads to the general observation that obtaining the current display name of a moniker may have to access at least the storage of the object to which it refers, if not the object itself. Thus, it has the potential to be an expensive operation. As in other IMoniker functions, a bind context parameter is passed which includes a deadline within which the operation should complete, or fail with MK_E_EXCEEDEDDEADLINE if unable to do so.

A consequence of the possible unavailability of quick access to the display name of a moniker is that callers of this function most likely will want to cache the last successful result that they obtained, and use that if the current answer is inaccessible (this caching is the Microsoft Excel between-sheet behavior).

In the general case, the display name of a moniker is *not* unambiguous: there may be more than one moniker with the same display name, though in practice this will be rare. There is also no *guarantee* that a display name obtained from a moniker will parse back into that moniker in MkParseDisplayName, though failure to do so also will be rare. Display names should therefore be thought of as a merely a note or annotation on the moniker which aid a human being in distinguishing one moniker from another, rather than a completely equivalent representation of the moniker itself.

Notice that due to how display names are constructed in composites, a moniker which is a prefix of another necessarily has a display name which is a (string) prefix of the display name of the second moniker. The converse, however, does not necessarily hold.

A moniker which is designed to be used as part of a generic composite is responsible for including any preceding delimiter as part of its display name. Many such monikers take a parameter for this delimiter in their instance creation functions.

Argument	Type	Description
pbc	IBindCtx*	the bind context for this operation.
pmkToLeft	IMoniker*	the moniker to the left of this one in the composite in which it is found. Most monikers will not require this in IMoniker::GetDisplayName.
lplpszDisplayName	LPSTR*	on exit, the current display name for this moniker. NULL if the moniker does not have a display name or the deadline was exceeded.
return value	HRESULT	S_OK, MK_E_EXCEEDEDDEADLINE.

.17 MkParseDisplayName

HRESULT MkParseDisplayName(pbc, lpszDisplayName, pcchEaten, ppmk)

Recall from IMoniker::GetDisplayName that most monikers have a textual name which is meaningful to the user. The function MkParseDisplayName does the logical inverse operation: given a string, it returns a moniker of the object that the string denotes. This operation is known as *parsing.* A display name is parsed into a moniker; it is resolved into its component moniker parts.

If a syntax error occurs, than an indication of how much of the string was successfully parsed is returned in pcchEaten and NULL is returned through ppmk. Otherwise, the value returned through pcchEaten indicates the entire size of the display name.

Argument	Type	Description
pbc	IBindCtx*	the binding context in which to accumulate bound objects.
lpszDisplayName	LPSTR	the display name to be parsed.
pcchEaten	ULONG*	on exit the number of characters of the display name that was successfully parsed. Most useful on syntax error.
ppmk	IMoniker*	the resulting moniker.
return value	HRESULT	S_OK, MK_E_SYNTAX.

Parsing a display name may in some cases be as expensive as binding to the object that it denotes, since along the way various non-trivial name space managers (such as a spreadsheet application that can parse into ranges in its sheets) need to be connected to by the parsing mechanism to succeed. As might be ex-

pected, objects are not released by the parsing operation itself, but are instead handed over to the passed-in binding context (via IBindCtx::RegisterObjectBound). Thus, if the moniker resulting from the parse is immediately bound using this same binding context, redundant loading of objects is maximally avoided.

In many other cases, however, parsing a display name may be quite inexpensive since a single name-space manager may quickly return a moniker that will perform further expensive analysis on any acceptable name during IMoniker::BindToObject or other methods. An example of such an inexpensive parser is the Win32 implementation of a File Moniker. A theoretical example would be a name URL moniker which parsed from any valid URL strings (ie, "http:…", "file:…", etc) and only during binding took time to resolve against the Internet server, a potentially expensive operation.

An important use of MkParseDisplayName worth noting lies in textual programming languages which permit remote references as syntactic elements. The expression language of a spreadsheet is a good example of such a language.

The parsing process is an inductive one, in that there is an initial step that gets the process going, followed by the repeated application of an inductive step. At any point after the beginning of the parse, a certain prefix of lpszDisplayName has been parsed into a moniker, and a suffix of the display name remains not understood. This is illustrated in Figure 5.

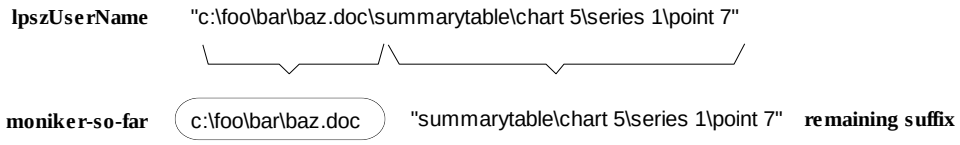


Figure 5. Intermediate stage in parsing a display name into a moniker.

The *inductive step* asks the moniker-so-far using IMoniker::ParseDisplayName to consume as much as it would like of the remaining suffix and return the corresponding moniker and the new suffix. The moniker is composed onto the end of the existing moniker-so-far, and the process repeats.

Implementations of IMoniker::ParseDisplayName vary in exactly where the knowledge of how to carry out the parsing is kept. Some monikers by their nature are only used in particular kinds of containers. It is likely that these monikers themselves have the knowledge of the legal display name syntax within the objects that they themselves denote, and so they can carry out the processes completely within IMoniker::ParseDisplayName. The common case, however, is that the moniker-so-far is generic in the sense that is not specific to one kind of container, and thus cannot know the legal syntax for elements within the container. File monikers are an example of these, as are Item Monikers. These monikers in general employ the following strategy to carry out parsing. First, the moniker connects to the *class* of object that it currently denotes, asking for IParseDisplayName interface. If that succeeds, then it uses the obtained interface pointer to attempt to carry out the parse. If the class refuses to handle the parse, then the moniker binds to the *object* it denotes, asking again for IParseDisplayName interface. If this fails, then the parse is aborted.

The effect is that ultimately an object always gets to be in control of the syntax of elements contained inside of itself. It's just that objects of a certain nature can carry out parsing more efficiently by having a moniker or their class do the parsing on their behalf.

Notice that since MkParseDisplayName knows nothing of the legal syntax of display names (with the exception of the initial parsing step; see below). It is of course beneficial to the user that display names in different contexts not have gratuitously different syntax. While there some rare situations which call for special purpose syntax, it is recommended that, unless there are compelling reasons to do otherwise, the syntax for display names should be the same as or similar to the native file system syntax; the aim is to build on user familiarity. Most important about this are the characters allowed for the delimiters used to separate the display name of one of the component monikers from the next. Unless through some special circumstances they have *very* good reason not to, all moniker implementations should use inter-moniker delimiters from the character set:

\ / : ! [

Standardization in delimiters promotes usability. But more importantly, notice that the parsing algorithm has the characteristic that a given container consumes as much as it can of the string being parsed before passing the remainder on to the designated object inside themselves. If the delimiter expected of the

next-to-be-generated moniker in fact forms (part of) a valid display name in the container, then the container's parse will consume it!

Monikers and objects which have implementations on more than one platform (such as File Monikers) should always parse according to the syntax of the platform on which they are currently running. When asked for their display name, monikers should also show delimiters appropriate to the platform on which they are currently running, even if they were originally created on a different platform. In total, users will always deal with delimiters appropriate for the host platform.

The *initial step* of the parsing process is a bit tricky, in that it needs to somehow determine the initial moniker-so-far. MkParseDisplayName is omniscient with respect to the syntax with which the display name of a moniker may legally begin, and it uses this omniscience to choose the initial moniker.

The initial moniker is determined by trying the following strategies in order, using the first to succeed.

1. All prefixes of lpszDisplayName that consist solely of valid file name characters are consulted as file monikers in the Running Object Table.
3. The file system is consulted to check if a prefix of lpszDisplayName matches an existing file. Said file name may be drive absolute, drive relative, working-directory relative, or begin with an explicit network share name. This is a common case.
4. If the initial character of lpszDisplayName is '@', then the maximal string immediately following the '@' which conforms to the legal ProgID syntax[100] is determined. This is converted to a CLSID with CLSIDFromProgID. An instance of this class is asked in turn for IParseDisplayName interface; the IParseDisplayName interface so found is then given the whole string (starting with the '@') to continue parsing.

.18 IMoniker::ParseDisplayName

HRESULT IMoniker::ParseDisplayName(pbc, pmkToLeft, lpszDisplayName, pcchEaten, ppmkOut)

Given that the composite moniker (pmkToLeft ∘ (the receiver)) is the moniker which has so far been parsed, parse as much of the remaining tail as is appropriate. In general, the maximal prefix of lpszDisplayName which is syntactically valid and which currently *represents an existing object* should be consumed.

The main loop of MkParseDisplayName finds the next piece moniker piece by calling this function on the moniker-so-far that it holds on to, passing NULL through pmkToLeft. In the case that the moniker-so-far is a generic composite, this is forwarded by that composite onto its last piece, passing the prefix of the composite to the left of the piece in pmkToLeft.

lpszDisplayName is the as-yet-to-be-parsed tail of the display name. This function is to consume as much of it as is appropriate for a name within the object identified by (pmkToLeft ∘ (the receiver)) and return the corresponding moniker.

Some moniker classes will be able to handle this parsing internally to themselves since they are designed to designate only certain kinds of objects. Others will need to bind to the object that they designate in order to accomplish the parsing process. As is usual, these objects should not be released by IMoniker::ParseDisplayName but instead should be transferred to the bind context for release at a later time.

If a syntax error occurs, then NULL should be returned through ppmkOut and MK_E_SYNTAX returned. In addition, the number of characters of the display name that were *successfully* parsed should be returned through pcchEaten.

[100] Letters, numbers, or periods; must not begin with a number. See the appendix.

Argument	Type	Description
pbc	IBindCtx*	the binding context in which to accumulate bound objects.
pmkToLeft	IMoniker*	the moniker to the left of this one in the so-far-parsed display name.
lpszDisplayName	LPSTR	the display name to be parsed.
pcchEaten	ULONG*	the number of characters of the input name that this parse consumed.
ppmkOut	IMoniker*	the resulting moniker.
return value	HRESULT	S_OK, MK_E_SYNTAX.

.19 IMoniker::IsSystemMoniker

HRESULT IMoniker::IsSystemMoniker(pdwMksys)[101]

Answer as to whether this moniker is a type of moniker whose particular implementation semantics are conceptually important to the binding process. The values returned through pdwMksys are taken from the following enumeration:

```
typedef enum tagMKSYS {
    MKSYS_NONE = 0,
    MKSYS_GENERICCOMPOSITE = 1,
    MKSYS_FILEMONIKER = 2,
    MKSYS_ANTIMONIKER = 3,
    MKSYS_ITEMMONIKER = 4,
    MKSYS_POINTERMONIKER = 5,
    } MKSYS;
```

All user implementations of this function must simply return MKSYS_NONE through pdwMksys. IMoniker::-GetClassID (see IPersist) can be used instead by non-system monikers to check for the presence of their own "special" moniker on the right in IMoniker::ComposeWith. Alternatively, use QueryInterface to test for the presence of your own private interface.

New values of this enumeration may be defined in the future; caller's of this function should be aware of this fact and should therefore explicitly test against known return values that they care about (rather than, for example, assuming that if the returned value is not one of the values listed here then it's the other).

The returned value is *not* a bitfield value; rather it is an integer.

Argument	Type	Description
pdwMksys	DWORD*	the place at which the result is to be returned. May not be NULL.
return value	HRESULT	S_OK

.2 IParseDisplayName interface

.1 IParseDisplayName::ParseDisplayName

HRESULT IParseDisplayName::ParseDisplayName(pbc, lpszDisplayName, pcchEaten, ppmkOut)

This is the single function in the IParseDisplayName interface:

```
interface IParseDisplayName : IUnknown {
    HRESULT     ParseDisplayName(pbc, lpszDisplayName, pcchEaten, ppmkOut);
    };
```

Its semantics and parameters are as described in IMoniker::ParseDisplayName.

.3 IBindCtx interface

The bind context parameter passed to many of the IMoniker operations serves a few purposes.

Its primary purpose is to accumulate the set of objects that get bound during an operation but which should be released when the operation is complete. This is particularly useful in generic composites: using the bind

[101] This function is a member of IMoniker interface rather than an independent API function in order that future revisions of this function can be correctly correlated with revisions to system moniker classes.

context in this way avoids binding an object, releasing it, only to have it bound again when the operation moves on to another piece of the composite.

Another purpose of the bind context is to pass a group of parameters which do not change as an operation moves from one piece of a generic composite to another. These are the *binding options*, and are described below. Some of these binding options have a related return value in certain error conditions; the bind context provides the means by which they can be returned.

The bind context is also the only means through which moniker operations should access contextual information about their environment. There should be no direct calls in moniker implementations to API functions that query or set state in the environment; all such calls should instead funnel through the bind context. Doing this allows for future enhancements which can dynamically modify binding behavior. The predefined piece of contextual information that moniker operations need to access is the Running Object Table; monikers should always access this table indirectly though IBindCtx::GetRunningObjectTable, rather than using the global function GetRunningObjectTable. IBindCtx interface allows for future extensions to the passed-in contextual information in the form the ability to maintain a string-keyed table of objects. See IBindCtx::RegisterObjectParam and related functions.

```
interface IBindCtx : IUnknown {
    virtual    HRESULT    RegisterObjectBound(punk);
    virtual    HRESULT    RevokeObjectBound(punk);
    virtual    HRESULT    ReleaseBoundObjects();
    virtual    HRESULT    SetBindOptions(pbindopts);
    virtual    HRESULT    GetBindOptions(pbindopts);
    virtual    HRESULT    GetRunningObjectTable(pprot);
    virtual    HRESULT    RegisterObjectParam(lpszKey, punk);
    virtual    HRESULT    GetObjectParam(lpszKey, ppunk);
    virtual    HRESULT    EnumObjectParam(ppenum);
    virtual    HRESULT    RevokeObjectParam(lpszKey);
    };

typedef struct {
    DWORD  cbStruct;                // the size in bytes of this structure. ie: sizeof(BINDOPTS).
    DWORD  grfFlags;
    DWORD  grfMode;
    DWORD  dwTickCountDeadline;
    } BINDOPTS;

HRESULT CreateBindCtx(reserved, ppbc);
```

.1 IBindCtx::RegisterObjectBound

HRESULT IBindCtx::RegisterObjectBound(punk)

Remember the passed object as one of the objects that has been bound during a moniker operation and which should be released when it is complete overall. Calling this function causes the binding context to create an additional reference to the passed-in object with AddRef; the caller is still required to Release its own copy of the pointer independently.

The effect of calling this function twice with the same object is cumulative, in that it will require two IBindCtx::RevokeObjectBound calls to completely remove the registration of the object within the binding context.

Argument	Type	Description
punk	IUnknown*	the object which is being registered as needing to be released.
return value	HRESULT	S_OK.

.2 IBindCtx::RevokeObjectBound

HRESULT IBindCtx::RevokeObjectBound(punk)

This function undoes the effect of IBindCtx::RegisterObjectBound: it removes the object from the set that will be released when the bind context in IBindCtx::ReleaseBoundObjects (actually removes one occurrence of it). This function is likely to be rarely called, but is included for completeness.

Argument	Type	Description
punk	IUnknown*	the object which no longer needs to be released.
return value	HRESULT	S_OK, MK_E_NOTBOUND, E_OUTOFMEMORY

.3 IBindCtx::ReleaseBoundObjects

HRESULT IBindCtx::ReleaseBoundObjects()

Releases all the objects currently registered with the bind context though IBindCtx::RegisterObjectBound.

This function is (conceptually) called by the implementation of IBindCtx::Release.

Argument	Type	Description
return value	HRESULT	S_OK

.4 IBindCtx::SetBindOptions

HRESULT IBindCtx::SetBindOptions(pbindopts)

Store in the bind context a block of parameters that will apply to later IMoniker operations using this bind context. Using block of parameters like this is just an alternative way to pass parameters to an operation. We distinguish the parameters we do for conveyance by this means because 1) they are common to most IMoniker operations, and 2) these parameters do not change as the operation moves from piece to piece of a generic composite.

Argument	Type	Description
pbindopts	BINDOPTS*	the block of parameters to set. These can later be retrieved with IBindCtx::GetBindOptions.
return value	HRESULT	S_OK, E_OUTOFMEMORY

BINDOPTS is defined as follows:

```
typedef struct tagBINDOPTS {
    DWORD  cbStruct;                // the size in bytes of this structure. ie: sizeof(BINDOPTS).
    DWORD  grfFlags;
    DWORD  grfMode;
    DWORD  dwTickCountDeadline;
    } BINDOPTS;
```

The members of this structure have the following meanings:

Member Description

grfFlags

A group of Boolean flags. Legal values that may be OR'd together are the taken from the enumeration BINDFLAGS; see below. Moniker implementations must ignore any set-bits in this field that they do not understand.

grfMode

A group of flags that indicates the intended use that the caller has towards the object that he ultimately receives from the associated moniker binding operation. Constants for this member are taken from the STGM enumeration.

When applied to the IMoniker::BindToObject operation, by far the most significant flag values are: STGM_READ, STGM_WRITE, and STGM_READWRITE. It is possible that some binding operations might make use of the other flags, particularly STGM_DELETEONRELEASE or STGM_CREATE, but such cases are quite esoteric.

When applied to the IMoniker::BindToStorage operation, *most* STGM values are potentially useful here.

The default value for grfMode is STGM_READWRITE | STGM_SHARE_EXCLUSIVE.

dwTickCountDeadline

This is an indication of when the caller would like the operation to complete. Having this parameter allows the caller to approximately and heuristically bound the execution time of an operation when it is more important that the operation perform quickly than it is that it perform accurately.

Most often, this capability is used with IMoniker::GetTimeOfLastChange, as was previously described, though it can be usefully applied to other operations as well.

This 32-bit unsigned value is a time in milliseconds on the local clock maintained by the GetTick-Count function. A value of zero indicates "no deadline;" callers should therefore be careful not to pass to the bind context a value of zero that was coincidentally obtained from GetTickCount. Clock wrapping is also a problem. Thus, if the value in this variable is less than the current time by more than 2^{31} milliseconds, then it should be interpreted as indicating a time in the future of its indicated value plus 2^{32} milliseconds.

Typical deadlines will allow for a few hundred milliseconds of execution. Each function should try to complete its operation by this time on the clock, or fail with the error MK_E_EXCEEDED-DEADLINE if it cannot do so in the time allotted. Functions are not *required* to be absolutely accurate in this regard, since it is almost impossible to predict how execution might take (thus, callers cannot rely on the operation completing by the deadline), but operations which exceeded their deadline excessively will usually cause intolerable user delays in the operation of their callers. Thus, in practice, the use of deadlines is a **heuristic** which callers can impose on the execution of moniker operations.

If a moniker operation exceeds its deadline because a given object or objects that it uses are not running, and if one of these had been running, then the operation would have completed more of its execution, then the monikers of these objects should be recorded in the bind context using IBindCtx::RegisterObjectParam under the parameter names "ExceededDeadline", "ExceededDeadline1", "ExceededDeadline2", etc.; use the first name in this series that is currently unused. This approach gives the caller some knowledge as to when to try the operation again

The enumeration BINDFLAGS, which contains the legal values for the bit-field BINDOPTS::grfFlags, is defined as follows:

```
typedef enum tagBINDFLAGS {
    BINDFLAGS_MAYBOTHERUSER = 1,
    BINDFLAGS_JUSTTESTEXISTENCE = 2,
    } BINDFLAGS;
```

These flags have the following interpretation.

Value	Description
BINDFLAGS_MAYBOTHERUSER	If not present, then the operation to which the bind context containing this parameter is applied should not interact with the user in any way, such to ask for a password for a network volume that needs mounting. If present, then this sort of interaction is permitted. If prohibited from interacting with the user when it otherwise would like to, an operation may elect to use a different algorithm that does not require user interaction, or it may fail with the error MK_MUSTBOTHERUSER.
BINDFLAGS_JUSTTESTEXISTENCE	If present, indicates that the caller of the moniker operation to which this flag is being applied is not actually interested in having the operation carried out, but only in learning of the operation could have been carried out had this flag not been specified. For example, this flag give the caller the ability to express that he is only interested in finding out whether an object actually exists by using this flag in a IMoniker::BindToObject call. Moniker implementations are free, however, to ignore this possible optimization and carry out the operation in full. Callers, therefore, need to be able to deal with both cases. See the individual routine descriptions for details of exactly what status is returned.

.5 IBindCtx::GetBindOptions

HRESULT IBindCtx::GetBindOptions(pbindopts)

Return the current binding options stored in this bind context. See IBindCtx::SetBindOpts for a description of the semantics of each option.

Notice that the caller provides a BINDOPTS structure, which is filled in by this routine. It is the caller's responsibility to fill in the cbStruct member correctly.

Argument	Type	Description
pbindOpts	BINDOPTS*	the structure of binding options which is to be filled in.
return value	HRESULT	S_OK, E_UNEXPECTED

.6 IBindCtx::GetRunningObjectTable

HRESULT IBindCtx::GetRunningObjectTable(pprot)

Return access to the Running Object Table relevant to this binding process. Moniker implementations should get access to the Running Object Table using this function rather than the global API GetRunningObjectTable. The appropriate Running Object Table is determined implicitly at the time the bind context is created.

Argument	Type	Description
pprot	IRunningObjectTable**	the place to return the running object table.
return value	HRESULT	S_OK, E_OUTOFMEMORY, E_UNEXPECTED

.7 IBindCtx::RegisterObjectParam

HRESULT IBindCtx::RegisterObjectParam(lpszKey, punk)

Register the given object pointer under the name lpszKey in the internally-maintained table of object pointers. The intent of this table is that it be used as an extensible means by which contextual information can be passed to the binding process. String keys are compared case-sensitive.

Like IBindCtx::RegisterObjectBound, this function creates an additional reference to the passed-in object using AddRef. The effect of calling this function a second time with the same lpszKey is to replace in the table the object passed-in the first time.

By convention, moniker implementers may freely use object parameters whose names begin with the string representation of the class id of the moniker implementation in question.

This facility is also used as means by which various errors can convey information back to the caller. Associated with certain error values are the following object parameters:

Error	Parameters
MK_E_EXCEEDEDDEADLINE	Parameters named "ExceededDeadline", "ExceededDeadline1", "ExceededDeadline2", etc., if they exist, are monikers who appearance as running would make it reasonable for the caller to attempt the binding operation again.
E_CLASSNOTFOUND	The parameter named "ClassNotFound", if present, is a moniker to the storage of the object whose class was not able to be loaded in the process of a moniker operation.

New moniker authors can freely use parameter names that begin with the string form of the CLSID of their moniker; see StringFromCLSID().

The arguments to this function are as follows:

Argument	Type	Description
lpszKey	LPSTR	the name under which the object is being registered.
punk	IUnknown*	the object being registered.
return value	HRESULT	S_OK, E_OUTOFMEMORY

.8 IBindCtx::GetObjectParam

HRESULT IBindCtx::GetObjectParam(lpszKey, ppunk)

Lookup the given key in the internally-maintained table of contextual object parameters and return the corresponding object, if one exists.

Argument	Type	Description
lpszKey	LPSTR	the key under which to look for an object.
ppunk	IUnknown**	The place to return the object interface pointer. NULL is returned on failure (along with S_FALSE).
return value	HRESULT	S_OK, S_FALSE

.9 IBindCtx::EnumObjectParam

HRESULT IBindCtx::EnumObjectParam(ppenum)

Enumerate the strings which are the keys of the internally-maintained table of contextual object parameters.

Argument	Type	Description
ppenum	IEnumString**	the place to return the string enumerator. See Chapter 4 for a description of IEnumString.
return value	HRESULT	S_OK, E_OUTOFMEMORY

.10 IBindCtx::RevokeObjectParam

HRESULT IBindCtx::RevokeObjectParam(lpszKey)

Revoke the registration of the object currently found under this key in the internally-maintained table of contextual object parameters, if any such key is currently registered.

Argument	Type	Description
lpszKey	LPSTR	the key whose registration is to be revoked.
return value	HRESULT	S_OK, S_FALSE

.11 CreateBindCtx

HRESULT CreateBindCtx(reserved, ppbc)

Allocate and initialize a new IBindCtx using a system-supplied implementation.

Argument	Type	Description
reserved	DWORD	reserved for future use; must be zero.
ppbc	IBindCtx*	the place in which to put the new BindCtx.
return value	HRESULT	S_OK, E_OUTOFMEMORY

.4 Generic Composite Moniker class

Recall that in general monikers are a composite list made up of other pieces. All monikers which are a generic composite of other monikers are instances of Generic Composite Moniker class whose implementation is provide with COM; there is no need for two Generic Composite Moniker classes.

The implementations of Generic Composite IMoniker::Reduce and Generic Composite IMoniker::BindToObject are particularly important as they manage the interactions between the various elements of the composite, and as a consequence define the semantics of binding to the moniker as a whole.

Generic composite monikers of size zero or of size one are never exposed outside of internal Generic Composite Moniker method implementations. From a client perspective, a Generic Composite Moniker always contains at least two elements.

.1 CreateGenericComposite

HRESULT CreateGenericComposite(pmkFirst, pmkRest, ppmkComposite)

Allocate and return a new composite moniker. pmkFirst and pmkRest are its first and trailing elements respectively. Either of pmkFirst and pmkRest may be a generic composite, or another kind of moniker. Generic composites get flattened into their component pieces before being put into the new composite. This func-

tion will be called by implementations of IMoniker::ComposeWith when they wish to carry out a generic compose operation.

Argument	Type	Description
pmkFirst	IMoniker*	the first element(s) in the new composite. May not be NULL.
pmkRest	IMoniker*	the trailing element(s) in the new composite. May not be NULL.
ppmkComposite	IMoniker*	through here is returned the new composite.
return value	HRESULT	S_OK, E_OUTOFMEMORY

.2 Generic Composite Moniker–IMoniker::Reduce

Reduction of a generic composite conceptually reduces each of its pieces in a left-to-right fashion and builds up a composite of the result. If any of the pieces of the composite did not reduce to themselves (and thus, the generic composite overall did not reduce to itself), then this process is repeated.

An optimized implementation of this function might use a more complicated but equivalent algorithm that avoids unnecessarily re-reducing monikers that the composite already knows reduce to themselves.

.3 Generic Composite Moniker–IMoniker::BindToObject

Binding to a generic composite works in a right-to-left manner. Conceptually, the generic composite merely forwards the bind request onto its last piece, along the way informing that piece of the moniker to its left in the composite. The last piece, if it needs to, recursively binds to the object to its left. In practice, binding to a generic composite itself has to handle the recursive call on the left-hand object, as was described in IMoniker::BindToObject.

.5 File Moniker class

A File Moniker can be thought of as a wrapper for a path name in the native file system. It's implementation of IMoniker::GetDisplayName, for example, is trivial: it just returns the path. When bound to, it determines the class of the file (using the API GetClassFile on Win32), makes sure that the appropriate class server is running, then asks it to open the file.

.1 CreateFileMoniker

HRESULT CreateFileMoniker(lpszPathName, ppmk)

Creates a moniker from the given path name. This path may be an absolute path or a relative path. In the latter case, the resulting moniker will need to be composed onto another File Moniker before it can be bound. In either case, it will often be the case that other monikers are composed onto the end of *this* moniker in order to access sub-pieces of the document stored in the file.

Argument	Type	Description
lpszPathName	LPSTR	the path name of the desired file.
ppmk	IMoniker*	the newly created moniker.
return value	HRESULT	S_OK, MK_E_SYNTAX, E_OUTOFMEMORY

.2 File Moniker–IMoniker::BindToObject

The class of an object designated by a File Moniker is determined in a platform-specific way. Having found the correct class, File Moniker IMoniker::BindToObject will instantiate it using the interface IPersistFile::Load method.

.3 File Moniker–IMoniker::BindToStorage

The Win32 implementation of File Moniker supports BindToStorage(..., IID_IStorage, ...) in the case that the designated file in fact a structured storage. Implementations may also choose to support IStream and ILockBytes.

.4 File Moniker–IMoniker::GetDisplayName

File monikers render their display names according to the syntax of the platform on which they are currently found. A File Moniker serialized on one platform then de-serialized on another will possibly have different display names on each platform.

.5 File Moniker–IMoniker::ParseDisplayName

File monikers designate objects that live in files; however, they have no knowledge of the name space contained *within* that file. A File Moniker for the path "C:\FOO.XLS", for example, knows how to connect to the spreadsheet that is the file, but it does not know anything of the syntax of range-address expression language of the sheet. Consequently, when asked to parse a display name, a File Moniker needs to delegate this operation to the class object that it designates. For this purpose it uses the IParseDisplayName interface. If the class refuses to handle the parse, the parsing is delegated to the object.

.6 Item Moniker class

Item monikers provide a bridge from the generality of IMoniker interface to the simple and common situation in which an object which is a container of other objects also provides a space of names for those objects. Examples include spreadsheet applications which contain "named ranges," various word-processing applications which support "bookmarks."

Item Moniker is a *class*, not a *interface*; that is, it is an *implementation* of IMoniker provided by COM library, not an interface that others implement. This implementation supports IMoniker interface by converting IMoniker invocations into a series of calls on part of the interface IOleItemContainer. The implication is that many object implementers do not have to deal much with monikers: they can deal simply with "items" in a string form, then wrap them in an Item Moniker as needed to support other interfaces.

A client creates an Item Moniker using CreateItemMoniker. When this new moniker is composed onto the end of a moniker that binds to an IOleItemContainer, the resulting composite moniker will bind to the appropriate contained object.

The following is the IOleItemContainer interface used by Item Monikers:

```
interface IOleItemContainer : IOleContainer {
    virtual    HRESULT    GetObject(lpszItem, dwSpeedNeeded, pbc, riid, ppvObject);
    virtual    HRESULT    GetObjectStorage(lpszItem, pbc, riid, ppvStorage);
    virtual    HRESULT    IsRunning(lpszItem);
    };
```

.1 CreateItemMoniker

HRESULT CreateItemMoniker(lpszDelim, lpszItem, ppmk)

Allocate and return a new Item Moniker. It is intended that the resulting moniker be then composed onto the end of a second moniker which binds to something that supports IOleItemContainer interface. The resulting composite moniker when bound will extract the object of the indicated name from within this container.

lpszItem is the item name which will be later passed to IOleItemContainer::GetObject. lpszDelim is simply another string that will prefix lpszItem in the display name of the Item Moniker.

See also IOleItemContainer::IsRunning.

Argument	Type	Description
lpszDelim	LPSTR	a string that will prefix lpszItem in the display name of this moniker. Often an exclamation mark: "!". See also the discussion of syntax in MkParseDisplayName.
lpszItem	LPSTR	the item name to pass to IOleItemContainer::GetObject.
ppmk	IMoniker**	the place to put the new Item Moniker.
return code	HRESULT	S_OK, E_OUTOFMEMORY

.2 Item Moniker–IMoniker::BindToObject

Item monikers merely require IOleItemContainer interface of the object to their left, which they obtain by invoking IMoniker::BindToObject on the moniker of the object to their left. Once they have the container, they merely invoke IOleItemContainer::GetObject with the internally stored item name.

.3 IOleItemContainer::GetObject

HRESULT IOleItemContainer::GetObject(lpszItem, dwSpeedNeeded, pbc, riid, ppvObject)

This method returns the object within this container specified by lpszItem.

dwSpeedNeeded is an indication of how willing the caller is to wait to get to the object. This value is set by the implementation of Item Moniker; the value it uses is derived from the dwTickCountDeadline parameter in the Bind Context that it receives. dwSpeedNeeded is one of the following values:

```
typedef enum tagBINDSPEED {
    BINDSPEED_INDEFINITE    = 1,  // the caller is willing to wait indefinitely
    BINDSPEED_MODERATE      = 2,  // the caller is willing to wait a moderate amount of time.
    BINDSPEED_IMMEDIATE     = 3,  // the caller is willing to wait only a very short time
    } BINDSPEED;
```

If BINDSPEED_IMMEDIATE is specified, then the object should be returned only if it is already running or if it is a pseudo-object (an object internal to the item container, such as a cell-range in a spreadsheet or a character-range in a word processor); otherwise, MK_E_EXCEEEDEDDEADLINE should be returned. BINDSPEED_MODERATE would include those things indicated by BINDSPEED_IMMEDIATE, plus, perhaps, those objects which are always running when the container is running and has them loaded: in this case, *load* (not load & run) the designated object, ask if it is running, and return it if so; otherwise, fail with MK_E_EXCEEEDEDDEADLINE as before. BINDSPEED_INDEFINITE indicates that time is of no concern to the caller.

The actual bind context parameter is also here passed in pbc for the use of more sophisticated containers. Less sophisticated containers can simply ignore this and look at dwSpeedNeeded instead. In effect, what the implementation of Item Moniker does is convert the deadline in the bind context into an appropriate dwSpeedNeeded, in the hope that the latter is easier to take a decision on for most containers.

Argument	Type	Description
lpszItem	LPSTR	the item in this container that should be bound to.
dwSpeedNeeded	DWORD	a value from the enumeration BINDSPEED. See above.
pbc	IBindCtx*	the actual deadline parameter involved in this binding operation. For the use of more sophisticated containers. Most can ignore this, and instead use dwSpeedNeeded.
riid	REFIID	the interface with which a connection to that object should be made.
ppvObject	void**	the bound-to object is returned here.
return value	HRESULT	S_OK, MK_E_EXCEEEDEDDEADLINE, MK_E_NOOBJECT, E_NOINTERFACE, E_OUTOFMEMORY

.4 Item Moniker–IMoniker::BindToStorage

For storage binding, Item Monikers merely require IOleItemContainer interface of the object to their left. The implementation of Item Moniker IMoniker::BindToStorage binds to the object to its left using IOleItemContainer interface, then invokes IOleItemContainer::GetObjectStorage with the internally stored item name.

.5 IOleItemContainer::GetObjectStorage

HRESULT IOleItemContainer::GetObjectStorage(lpszItem, pbc, riid, ppvStorage)

If lpszItem designates an item in this container that has an independently identifiable piece of storage (such as does an embedded object), then return access to that storage using the indicated interface.

pbc is the bind context as received by the Item Moniker IMoniker::BindToStorage call. Most container implementations can simply ignore this value; it is passed for the benefit for more sophisticated containers.

Argument	Type	Description
lpszItem	LPSTR	the item access to whose storage is being requested.
pbc	IBindCtx*	as in IOleItemContainer::GetObject. Can be ignored by most containers.
riid	REFIID	the interface by which the caller wishes to access that storage. Often IID_IStorage or IID_IStream are used.
ppvStorage	void**	the place to return the access to the storage
return value	HRESULT	S_OK, MK_E_EXCEEDEDDEADLINE, MK_E_NOOBJECT, E_OUTOFMEMORY, E_NOINTERFACE, MK_E_NOSTORAGE

.6 IOleItemContainer::IsRunning

HRESULT IOleItemContainer::IsRunning(lpszItem)

Answer whether the given item in this item container is in fact running or not. See IMoniker::IsRunning for a sketch of how this function is used in Item Monikers.

Argument	Type	Description
lpszItem	LPSTR	the item access to whose running status is being requested.
return value	HRESULT	S_OK, S_FALSE, MK_E_NOOBJECT

.7 Item Moniker–IMoniker::ParseDisplayName

Item Moniker IMoniker::ParseDisplayName uses IParseDisplayName in the same way as a File Moniker does. Note that it requests this interface from a *different object* than the one that supplies the IOleItemContainer interface used in IMoniker::BindToObject, etc.: it asks for IOleItemContainer of the object designated by the moniker to its left, whereas it asks for IParseDisplayName of the object that *it* (the Item Moniker) designates.

.7 Anti Moniker class

An Anti Moniker is a moniker that when composed onto the end of a generic composite moniker removes the last piece. Composing an Anti Moniker onto the end of another kind of moniker should, generally, annihilate the whole other moniker.

Being composed onto the end of another moniker is pretty much the only interesting thing one can do to an anti-moniker: they cannot be bound, their display name is useless, etc. They exist to support implementations of IMoniker::Inverse; see that function for usage scenarios.

Moniker implementations that use Anti Monikers as inverses should check for Anti Monikers on the right in their implementations of IMoniker::ComposeWith and collapse down to nothing if so.

.1 CreateAntiMoniker

HRESULT CreateAntiMoniker(ppmk)

Create and return a new anti-moniker.

Argument	Type	Description
ppmk	IMoniker**	the place to return the new anti-moniker
return value	HRESULT	S_OK, E_OUTOFMEMORY

.8 Pointer Moniker class

A pointer moniker is a kind of moniker that wraps an existing object pointer in a moniker so that it may participate as a piece in the moniker binding process. Think of pointers as a referencing mechanism into the "active space:" a process's memory. Most moniker implementations are by contrast references into "passive space:" the representation of an object on disk. Pointer monikers provide a means by which a a given use of a moniker can transparently reference *either* active *or* passive space.

Implementations of functions in IMoniker interface for Pointer Monikers work roughly as follows. IMoniker::BindToObject turns into a QueryInterface on the pointer; IMoniker::BindToStorage returns MK_E_NOSTORAGE; IMoniker::Reduce() reduces the moniker to itself; IMoniker::ComposeWith always does a generic composition; IMoniker::Enum returns NULL; IMoniker::IsSystemMoniker returns MKSYS_PTR; IMoniker::IsEqual() uses the identity test paradigm on pointers after first checking that the other moniker for the right class; IMoniker::Hash returns a constant; IMoniker::GetTimeOfLastChange returns MK_E_UNAVAILABLE; IMoniker::Inverse returns an anti-moniker; IMoniker::RelativePathTo returns the other moniker; IMoniker::GetDisplayName returns NULL; and IMoniker::ParseDisplayName() binds to the punk pointer using IParseDisplayName interface and works from there.

Instances of this kind of moniker refuse to be serialized; that is, IPersistStream::Save will return an error. These monikers can, however, be *marshaled* to a different process; internally, this marshals and unmarshals the pointer using the standard paradigm for marshaling interface pointers: CoMarshalInterface and CoUnmarshalInterface.

.1 CreatePointerMoniker

HRESULT CreatePointerMoniker(punk, ppmk)

Wrap a pointer in a Pointer Moniker so that it can be presented to interfaces that require monikers for generality, but specific uses of which can usefully deal with a moniker which cannot be saved to backing store.

Argument	Type	Description
punk	IUnknown*	the pointer that we are wrapping up in a moniker.
ppmk	IMoniker**	the returned Pointer Moniker.
return value	HRESULT	S_OK, E_OUTOFMEMORY

.9 Running Object Table

In general when binding to an object we want to open it if it is currently passive, but if not, then we want to connect to the running instance. To take an example from Compound Documents, a link to a Lotus 123 for Windows spreadsheet, for example, when first bound to should open the spreadsheet, but a second bind should connect to the already-open copy. The key technical piece that supports this type of functionality is the Running Object Table.

The Running Object Table is a globally accessible table on each workstation. It keeps track of the objects that are currently running on that workstation so that if an attempt is made to bind to one a connection to the currently running instance can be made instead of loading the object a second time. The table conceptually is a series of tuples, each of the form:

(pmkObjectName, pvObject)

The first element is the moniker that if bound should connect to the running object. The second element is the object that is publicized as being available, the object that is running. In the process of binding, monikers being bound with nothing to their left consult the pmkObjectName entries in the Running Object Table to see if the object that they (the moniker being bound) indicate is already running.

Access to the Running Object Table is obtained with the function GetRunningObjectTable. This returns an object with the interface IRunningObjectTable (note as described earlier, however, that moniker imple-

mentations should not use this API, but should instead access the Running Object Table from the bind context they are passed).

```
interface IRunningObjectTable : IUnknown {
    HRESULT     Register(reserved, pUnkObject, pmkObjectName, pdwRegister);
    HRESULT     Revoke(dwRegister);
    HRESULT     IsRunning(pmkObjectName);
    HRESULT     GetObject(pmkObjectName, ppunkObject);
    HRESULT     NoteChangeTime(dwRegister, pfiletime);
    HRESULT     GetTimeOfLastChange(pmkObjectName, pfiletime);
    HRESULT     EnumRunning(ppenumMoniker);
    };

SCODE   GetRunningObjectTable(reserved, pprot);
```

.1 GetRunningObjectTable

HRESULT GetRunningObjectTable(reserved, pprot)

Return a pointer to the Running Object Table for the caller's context.

Argument	Type	Description
reserved	DWORD	reserved for future use; must be zero.
pprot	IRunningObjectTable**	the place to return the running object table.
return value	HRESULT	S_OK

.2 IRunningObjectTable::Register

HRESULT IRunningObjectTable::Register(reserved, pUnkObject, pmkObjectName, pdwRegister)

Register the fact that the object pUnkObject has just entered the running state and that if the moniker pmkObjectName is bound to, then this object should be used as the result of the bind (with an appropriate QueryInterface).

The moniker pmkObjectName should be fully reduced before registration. See IMoniker::Reduce for a more complete discussion. If an object goes by more than one fully reduced moniker, then it should register itself under all such monikers. Here, "fully reduced" means reduced to the state MKRREDUCE_THROUGUSER.

Registering a second object under the same moniker sets up a second independent registration, though MK_S_MONIKERALREADYREGISTERED is returned instead of S_OK. This is done without regard to the value of pUnkObject in the second registration; thus, registering the exact same (pmkObjectName, pUnkObject) pair a second time will set up a second registration. It is not intended that multiple registration under the same moniker be a common occurrence, as which registration actually gets used in various situations is non-deterministic.

The arguments to this function are as follows:

Argument	Type	Description
reserved	DWORD	reserved for future use; must be zero.
pUnkObject	IUnknown*	the object which has just entered the running state.
pmkObjectName	IMoniker*	the moniker which would bind to the newly running object.
pdwRegister	DWORD*	a place to return a value by which this registration can later be revoked. May not be NULL. Zero will never be returned as a valid registration value; that is, on exit, *pdwRegister is never NULL.
return value	HRESULT	S_OK, MK_S_MONIKERALREADYREGISTERED

.3 IRunningObjectTable::Revoke

HRESULT IRunningObjectTable::Revoke(dwRegister)

Undo the registration done in IRunningObjectTable::Register, presumably because the object is about to cease to be running. Revoking an object that is not registered as running returns the status code E_INVALIDARG. Whenever any of the conditions that cause an object to register itself as running cease to be true, the object

should revoke its registration(s). In particular, objects should be sure to extant registrations of themselves from the Running Object Table as part of their release process.

Argument	Type	Description
dwRegister	DWORD	a value previously returned from IRunningObjectTable::Register.
return value	HRESULT	S_OK, E_INVALIDARG.

.4 IRunningObjectTable::IsRunning

HRESULT IRunningObjectTable::IsRunning(pmkObjectName)

This function should, in general, only be called by implementations of IMoniker::IsRunning; clients of monikers should invoke this on their monikers, rather than asking the Running Object Table directly.

Inquire by looking up in this Running Object Table as to whether an object with this moniker is currently registered as running. Success or failure is indicated using the return codes S_OK or S_FALSE. The Running Object Table compares monikers by sending IMoniker::IsEqual to the monikers already in the table with moniker on the right as an argument.

Argument	Type	Description
pmkObjectName	IMoniker*	the moniker that we want to see is running
return value	HRESULT	S_OK, S_FALSE.

.5 IRunningObjectTable::GetObject

HRESULT IRunningObjectTable::GetObject(pmkObjectName, ppunkObject)

If the object designated by pmkObject name is registered as actually running, then return the object so registered. The R.O.T. compares monikers by sending IMoniker::IsEqual to the monikers already in the table with moniker on the right as an argument.

This is the function moniker implementations should use to test if they are already running (and get the pointer to the object if so).

Argument	Type	Description
pmkObjectName	IMoniker*	the moniker in whom interest is being expressed.
ppunkObject	IUnknown**	the place to return the pointer to the object. A returned value of NULL indicates that the object is not registered.
return value	HRESULT	S_OK, MK_E_UNAVAILABLE

.6 IRunningObjectTable::NoteChangeTime

HRESULT IRunningObjectTable::NoteChangeTime(dwRegister, pfiletime)

Make a note of the time that a particular object has changed in order that IMoniker::GetTimeOfLastChange can report an appropriate change time. This time so registered is retrievable with IRunningObjectTable::GetTimeOfLastChange. Objects should call this as part of their data change notification process.

Argument	Type	Description
dwRegister	DWORD	the token previously returned from IRunningObjectTable::Register. The moniker whose change time is noted is the one specified in pmkObjectName in that call.
pfiletime	FILETIME*	on entry, the time at which the object has changed.
return value	HRESULT	S_OK, E_INVALIDARG

.7 IRunningObjectTable::GetTimeOfLastChange

HRESULT IRunningObjectTable::GetTimeOfLastChange(pmkObjectName, pfiletime)

As with IMoniker::IsRunning, this function should, in general, only be called by implementations of IMoniker::GetTimeOfLastChange; clients of monikers should invoke this on their monikers, rather than asking the Running Object Table directly.

Look up this moniker in the running object table and report the time of change recorded for it if same is present. The Running Object Table compares monikers by sending IMoniker::IsEqual to the monikers already in the table with moniker on the right as an argument. Implementations of IMoniker::GetTimeOfLastChange, when invoked with pmkToLeft == NULL, will want to call this function as the first thing they do.

Argument	Type	Description
pmkObjectName	IMoniker*	the moniker in which we are interested in the time of change.
pfiletime	FILETIME*	on exit, the place at which the time of change is returned.
return value	HRESULT	S_OK, MK_E_UNAVAILABLE

.8 IRunningObjectTable::EnumRunning

HRESULT IRunningObjectTable::EnumRunning(ppenumMoniker)

Enumerates the objects currently registered as running. The returned enumerator is of type IEnumMoniker, which enumerates monikers.

```
typedef Enum<IMoniker*>  IEnumMoniker;
```

The monikers which have been passed to IRunningObjectTable::Register() are enumerated.

Argument	Type	Description
ppenumMoniker	IEnumMoniker**	the place at which to return the enumerator.
return value	HRESULT	S_OK, E_OUTOFMEMORY

12. Uniform Data Transfer

In order to reduce the overall size of this document, and because the topic of this chapter is fully specified in the Microsoft Win32 Software Development Kit, the text of this chapter has been omitted.

This page intentionally left blank.

Part IV: Type Information

This page intentionally left blank.

13. Interface Definition Language

As was described previously in this specification, the COM infrastructure is completely divorced from the source-level tools used to create and use COM components. COM is completely a *binary* specification, and thus source-level specifications and standards have no role to play in the fundamental architecture of the system.

Specifically, and somewhat different than other environments, this includes any and all forms of interface definition language (IDL). Having an interoperable standard for an appropriate IDL (or any other source level tool for that matter) is still incredibly valuable and useful, it's just important to understand that this is a *tool* standard and not a fundamental *system* standard. Contrast this, for example, with the DCE RPC API specification, where, if only because the fundamental SendReceive API is not part of the public standard runtime infrastructure, one *must* use IDL to interoperate with the system.[102] People can (and have, out of necessity) built COM components with custom COM interfaces without using any interface definition language at all. This clear separation of system standards from tools standards is an important point, for without it COM tools vendors cannot innovate without centralizing their innovations through some central standards body. Innovation is stifled, and the customers suffer a loss of valuable tools in the marketplace.

That all being said, as was just mentioned, source-level standards are still useful, and DCE IDL is one such standard. The following enhancements to DCE IDL enable it to specify COM interfaces in addition to DCE RPC interfaces.[103]

1 Object RPC IDL Extensions

.1 'Object' interface attribute

COM interfaces are signified by a new interface attribute, 'object'. See [CAE RPC], page 238.

```
<interface_attributes> ::= <interface_attribute> [ , <interface_attribute> ] ...
<interface_attribute>  ::= uuid ( <Uuid_rep> )
    | version ( <Integer_literal>[.Integer_literal>])
    | endpoint ( <port_spec> [ ,<port_spec> ] ... )
    | local
    | pointer_default ( <ptr_attr> )
    | object
<port_spec> ::= <Family_string> : <[> <Port_string> <]>
```

The object interface attribute attributed may not be specified with the version attribute. However, it may be specified with any of the others, though the uuid attribute is virtually always used and the local attribute is used but rarely.

If this keyword is present, the following extensions are enabled in the interface.

.2 Interface name as a type

The interface name becomes the name of a type, which can then be used as a parameter in methods. For example:

```
[object, uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
    interface IFoo {
        };
```

causes a typed named "IFoo" to be declared, such that a method

```
[object, uuid(6A874340-57EB-11ce-A964-00AA006C3706)]
    interface IBar {
        HRESULT M1([in] short i, [in] IFoo* pfoo);
        };
```

is a legal declaration.

---

[102] By definition one cannot, for example, write a source-portable DCE IDL compiler, for the code that calls SendReceive in the proxies is implementation-specific.

[103] Microsoft has in its MIDL specification language defined additional extensions to DCE IDL; however, these are orthogonal to the subject of COM interface, and thus are not dealt with here.

.3 No handle_t required

In methods, no handle_t argument is needed, and it's absence does not indicate auto-binding. Instead, a "this" pointer is used in the C++ binding to indicate the remote object being referenced, and an implicit extra first argument is used in C. For example:

```
[object, uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
    interface IBar {
        HRESULT Bar([in] short i, [in] IFoo * pIF);
        };
```

can be invoked from C++ with:

```
IFoo * pIF;
IBar * pIB;
pIB->Bar(3, pIF);
```

or from C with the equivalent

```
pIB->lpVtbl->Bar(pIB, 3, pIF);
```

.4 Interface inheritance

Single inheritance of interfaces is supported, using the C++ notation for same. Referring again to [CAE RPC], page 238:

```
<interface_header> ::=
    <[> <interface_attributes> <]> interface <Identifier> [ <:> <Identifier> ]
```

For example:

```
[object, uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
    interface IBar : IWazoo {
        HRESULT Bar([in] short i, [in] IFoo * pIF);
        };
```

cases the first methods in IBar to be the methods of IWazoo.

*.5 iid_is and use of void**

The use of "void*" pointers are permitted, as long as such pointers are qualified with an "iid_is" pointer attribute. See [CAE RPC], page 253.

```
<ptr_attr> ::= ref
    | ptr
    | iid_is ( <attr_var_list> )
    | unique[104]
```

The iid_is construct says that the void* parameter is an interface pointer whose type is only known at run time, but whose interface ID is the parameter named in the iid_is attribute. For example:

```
[object, uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
    interface IBar : IWazoo {
        Bar([in] short i, [in, ref] uuid_t *piid, [out, iid_is(piid)] void ** ppvoid);
        };
```

This can be invoked in C++ as:

```
IFoo* pIF;
pIB->Bar(i, &IID_IFoo, (void*)&pIF);
```

where "IID_IFoo" is the name of a constant whose value is the interface ID for IFoo.

.6 All methods must return void or HRESULT

Asynchronous methods (and only asynchronous methods) must return void, all others must return HRESULT.

.7 The wire_marshal attribute

```
    typedef  [wire_marshal( transmissible_type)]   type_specifier  user_type;
```

[104] This is a non-COM-related Microsoft extension, shown here for completeness.

This attribute is a type attribute used in the IDL file and is somewhat similar in syntax and semantic to the transmit_as attribute. Each user-specific type has a corresponding transmissible type that defines the wire representation.

The user can define his specific type quite freely, (simple types, pointer types and composite types may be used) although some restrictions apply. The main one is that the type object needs to have well defined (fixed) memory size. If the changeable size needs to be accommodated, the type should have a pointer field as opposed to a conformant array; or, it can be a pointer to the interesting type. General restrictions apply as usual. Specific restrictions related to embedding affect the way types can be specified. For more information see the "User type vs. wire type" section.

The [wire_marshal] attribute cannot be used with [allocate()] attribute (directly or indirectly) as the engine doesn't control the memory allocation for the type. Also the wire type cannot be an interface pointer (these may be marshaled directly) or a full pointer (we cannot take care of the aliasing).

The following is a list of additional points regarding wire_marshal:

- The wire type cannot be an interface pointer.
- The wire type cannot be a full pointer.
- The wire type cannot have allocate attribute on it (like [allocate(all_nodes)]).
- The wire type has to have a well defined memory size (cannot be a conformant structure etc.) as we allocate the top level object for the user as usual.
- When the engine delegates responsibility for a wire_marshalable type to the user supplied routines, everything is up to the user including servicing of the possible embedded types that are defined with wire_marshal, user_marshal, transmit_as etc.
- wire_marshal is mutually exclusive with user_marshal, transmit_as or represent_as when applied to the same type.
- Two different user types cannot resolve to the same wire type and vice versa.
- The user type may or may not be rpc-able.
- The user type must be known to MIDL.

.8 The user_marshal attribute

```
typedef  [user_marshal( user_type)]   transmissible_type;
```

This attribute is a type attribute used in the ACF file and is somewhat similar in syntax and semantic to the represent_as attribute. Each user-specific type has a corresponding transmissible type that defines the wire representation. Similar to represent_as, in the generated files, each usage of the trasmissible_type name is substituted by the user_type name.

The user can define his specific type quite freely, (simple types, pointer types and composite types may be used) although some restrictions apply. The main one is that the type object needs to have well defined (fixed) memory size. If the changeable size needs to be accommodated, the type should have a pointer field as opposed to a conformant array; or, it can be a pointer to the interesting type. General restrictions apply as usual. Specific restrictions related to embedding affect the way types can be specified. For more information see the "User type vs. wire type" section.

The [user_marshal] attribute cannot be used with [allocate()] attribute (directly or indirectly) as the engine doesn't control the memory allocation for the type. Also the wire type cannot be an interface pointer (these may be marshaled directly) or a full pointer (we cannot take care of the aliasing).

Additional points regarding user_marshal:

- The wire type cannot be an interface pointer.
- The wire type cannot be a full pointer.
- The wire type cannot have allocate attribute on it (like [allocate(all_nodes)]).
- The wire type has to have a well defined memory size (cannot be a conformant structure etc.) as we allocate the top level object for the user as usual.

- When the engine delegates responsibility for a user_marshalable type to the user supplied routines, everything is up to the user including servicing of the possible embedded types that are defined with user_marshal, transmit_as etc.
- user_marshal is mutually exclusive with wire_marshal, transmit_as or represent_as when applied to the same type.
- Two different wire types cannot resolve to the same user type and vice versa.
- The user type may or may not be rpc-able.
- The user type may or may not be known to MIDL.

.9 User supplied routines

The routines required by user_marshall have the following prototypes.

<type_name> means a user specific type name. This may be non-rpcable type or even, when used with user_marshal, a type unknown to MIDL at all. The wire type name (the name of transmissible type) is not used here.

```
unsigned long __RPC_USER   <type_name>_UserSize(
      unsigned long __RPC_FAR *  pFlags,
      unsigned long              StartingSize,
      <type_name>   __RPC_FAR *       pFoo);

unsigned char __RPC_FAR * __RPC_USER   <type_name>_UserMarshal(
      unsigned long  __RPC_FAR * pFlags,
      unsigned char __RPC_FAR *       Buffer,
      <type_name>   __RPC_FAR *       pFoo);

unsigned char __RPC_FAR * __RPC_USER   <type_name>_UserUnmarshal(
      unsigned long  __RPC_FAR * pFlags,
      unsigned char __RPC_FAR *       Buffer,
      <type_name>   __RPC_FAR *pFoo);

void __RPC_USER   <type_name>_UserFree(
      unsigned long  __RPC_FAR * pFlags,
      <type_name>   __RPC_FAR *pFoo );
```

The meaning of the arguments is as follows:

pFlags - pointer to a flag ulong. Flags: local call flag, data rep flag.

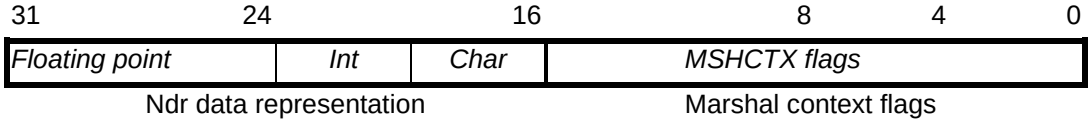
pBuffer - the current buffer pointer,

pFoo - pointer to a user type object

StartingSize - the buffer size (offset) before the object

The return value when sizing, marshaling or unmarshaling is the new offset or buffer position. See the function description below for details.

The flags pointed to by the first argument have the following layout.



- Upper word: NDR representation flags as defined by DCE: floating point, endianess and character representations.
- Lower word: marshaling context flags as defined by the COM channel. The flags are defined in the public wtypes.h file (and in wtypes.idl file). Currently the following flags are defined:

```
typedef
enum tagMSHCTX
    {MSHCTX_LOCAL          = 0,
          MSHCTX_NOSHAREDMEM  = 1,
```

```
        MSHCTX_DIFFERENTMACHINE = 2,
        MSHCTX_INPROC        = 3
    } MSHCTX;
```

The flags make it possible to differ the behavior of the routines depending on the context for the RPC call. For example when a handle is remoted in-process it could be sent as a handle (a long), while sending it remotely would mean sending the data related to the handle.

.1 _UserSize

The *_UserSize routine is called when sizing the RPC data buffer before the marshaling on the client or server side. The routine should work in terms of cumulative size. The StartingSize argument is the current buffer offset . The routine should return the cumulative size that includes the possible padding and then the data size. The starting size indicates the buffer offset for the user object and it may or may not be aligned properly. User's routine should account for all padding as necessary. In other words, the routine should return a new offset, after the user object. The sizing routine is not called if the wire size can be computed at the compile time. Note that for most unions, even if there are no pointers, the actual size of the wire representation may be determined only at the runtime.

This routine actually can return an overestimate as long as the marshaling routine does not use more than the sizing routine promised and so the marshaling buffer is not overwritten then or later (by subsequent objects).

.2 _UserMarsahal

The *_UserMarshal routine is called when marshaling the data on the client or server side. The buffer pointer may or may not be aligned upon the entry. The routine should align the buffer pointer appropriately, marshal the data and then return the new buffer pointer position which is at the first "free" byte after the marshaled object. For the complications related to pointees see the next chapter.

Please note that the wire type specification is a contract that determines the actual layout of the data in the buffer. For example, if the conversion is needed and done by the NDR engine, it follows from the wire type definitions how much data would be processed in the buffer for the type.

.3 _UserUnmarshal

The *_UserUnmarshal routine is called when unmarshaling the data on the client or server side. The flags indicate if data conversion is needed (if needed, it has been performed by the NDR engine before the call to the routine). The buffer pointer may or may not be aligned upon the entry. The routine should align the buffer as appropriate, unmarshal the data and then return the new buffer pointer position, which is at the first "free" byte after the unmarshaled object. For the complications related to pointees see the next chapter

.4 _UserFree

The *_UserFree routine is called when freeing the data on the server side. The object itself doesn't get freed as the engine takes care of it. The user shall free the pointees of the top level objects.

.10 The library keyword

```
[attributes] library libname {definitions};
```

The library keyword indicates that a type library (See Chapter 14) should be generated.[105] Below is an example library section.

```
[
    uuid(3C591B22-1F13-101B-B826-00DD01103DE1),  // IID_ISome
    object
]
```

[105] Historically the library statement was supported only in a variant of IDL called ODL that was central to OLE Automation.


```
interface ISome : IUnknown
{
    HRESULT DoSomething(void);
}

[
    uuid(3C591B20-1F13-101B-B826-00DD01103DE1),      // LIBID_Lines
    helpstring("Lines 1.0 Type Library"),
    lcid(0x0409),
    version(1.0)
]
library Lines
{
    importlib("stdole.tlb");
    [
        uuid(3C591B21-1F13-101B-B826-00DD01103DE1),  // CLSID_Lines
        helpstring("Lines Class"),
        appobject
    ]
    coclass Lines
    {
        [default] interface ISome;
        interface IDispatch;
    }
}
```

2 Mapping from ORPC IDL to DCE RPC IDL.

From the above extensions, and the wire representation definitions, one can conclude the following rules for converting ORPC IDL files to DCE IDL files:

1. Remove the [object] attribute from the interface definition.
2. Insert "[in] handle_t h" as the first argument of each method, "[in] ORPCTHIS *_orpcthis" as the second, and "[out] ORPCTHAT *_orpcthat" as the third.
3. Manually insert declarations for the operations that were inherited, if any. You may want to make the method names unique, unless the EPV invocation style is always going to be used. One way to do this is to prefix each method with the name of the interface. (Note that the IUnknown methods will never be called, as the IRemUnknown interface is used instead.)
4. Replace each occurrence of a type name derived from an interface name, or an [iid_is] qualified void* with OBJREF. Remove [iid_is] attributes.

.1 An Example

.1 Object RPC Style

```
[object, uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
interface IFoo: IUnknown
    {
    HRESULT Bar([in] short i, [in] IBozo* pIB, [out] IWaz** ppIW);
    HRESULT Zork([in, ref] UUID* iid,   [out, iid_is(iid)] void** ppvoid);
    };
```

.2 DCE style

```
[uuid(b5483f00-4f6c-101b-a1c7-00aa00389acb)]
interface IFoo
    {
    HRESULT IFoo_QueryInterface([in] handle_t h, [in] ORPCTHIS* _orpcthis, [out] ORPCTHAT* _orpcthat, [in, ref] UUID* iid,
        [out] OBJREF** ppOR);
    ULONG IFoo_AddRef([in] handle_t, [in] ORPCTHIS* _orpcthis, [out] ORPCTHAT* _orpcthat);
    ULONG IFoo_Release([in] handle_t, [in] ORPCTHIS* _orpcthis, [out] ORPCTHAT* _orpcthat);
    HRESULT IFoo_Bar([in] handle_t h, [in] ORPCTHIS* _orpcthis, [out] ORPCTHAT* _orpcthat, [in, ref] OBJREF* porIB, [out,
        ref] OBJREF** pporIW);
```

```
    HRESULT IFoo_Zork([in]handle_t h, [in] ORPCTHIS* _orpcthis, [out] ORPCTHAT* _orpcthat, [in, ref] UUID* iid, [out]
        OBJREF** ppvoid);
};
```

See Chapter 15 Network Protocol for information on the ORPCTHIS and ORPCTHAT structures and the IRemUnknown interface.

This page intentionally left blank.

14. Type Libraries

In order to reduce the overall size of this document, and because the topic of this chapter is fully specified in the Microsoft Win32 Software Development Kit, the text of this chapter has been omitted.

This page intentionally left blank.

Part V: The COM Library

It should be clear by this time that COM itself involves some systems-level code, that is, some implementation of its own. However, at the core the Component Object Model by itself is a specification (hence "Model") for how objects and their clients interact through the binary standard of interfaces. As a specification it defines a number of other standards for interoperability:

- The fundamental process of interface negotiation through QueryInterface.
- A *reference counting* mechanism through objects (and their resources) are managed even when connected to multiple clients.
- Rules for memory allocation and responsibility for those allocations when exchanged between independently developed components.
- Consistent and rich error reporting facilities.

In addition to being a specification, COM is also an implementation contained what is called the "COM Library." The implementation is provided through a library (such as a DLL on Microsoft Windows) that includes:

- A small number of fundamental API functions that facilitate the creation of COM applications, both clients and servers. For clients, COM supplies basic object creation functions; for servers the facilities to expose their objects.
- Implementation locator services through which COM determines from a class identifier which server implements that class and where that server is located. This includes support for a level of indirection, usually a system registry, between the identity of an object class and the packaging of the implementation such that clients are independent of the packaging which can change in the future.
- Transparent remote procedure calls when an object is running in a local or remote server. This includes the implementation of a standard network wire-protocol.
- A standard mechanism to allow an application to control how memory is allocated within its process.

In general, only one vendor needs to, or should, implement a COM Library for any particular operating system. For example, Microsoft has implemented COM on Microsoft Windows 3.1, Microsoft Windows 95, Microsoft Windows NT, and the Apple Macintosh.

The following chapter describes elements of the COM Library that a vendor implementing COM on a previously unsupported platform would require.

This page intentionally left blank.

15. Component Object Model Network Protocol

The COM network protocol is a protocol for object-oriented remote procedure calls and is thus also called Object RPC or ORPC. The Object RPC protocol consists of a set of extensions, layered on the distributed computing environment (DCE) RPC specification. The Object RPC protocol specifies:

- How calls are made on an object
- How object references are represented, communicated, and maintained

1 Overview

The Object RPC protocol highly leverages the OSF DCE RPC network protocol (see the reference [CAE RPC]). This leverage occurs at both the specification level and the implementation level: the bulk of the implementation effort involved in implementing the COM network protocol is in fact that of implementing the DCE RPC network protocol on which it is built.

.1 Object Calls

An actual COM network remote procedure call (hereinafter referred to as "an ORPC") is in fact a true DCE remote procedure call (herein termed "a DCE RPC"), a "Request PDU" conforming to the specification for such calls per [CAE RPC].

In an ORPC, the object ID field of the invocation header as specified in [CAE RPC] contains an "IPID". An IPID is a 128-bit identifier known as an *interface pointer identifier* which represents a particular interface on a particular object in a particular server. As it is passed in the object ID fields of a DCE RPC, the static type of an IPID is in fact a UUID. However, IPIDs are scoped not globally but rather only relative to the machine on which the server is located (and which thus originally allocated them); IPIDs do not necessarily use the standard UUID allocation algorithm, but rather may use a machine-specific algorithm which can assist with dispatching.[106]

In an ORPC, the interface ID field of the RPC header specifies the IID, and arguments are found in the body, as usual.[107] However, when viewed from the DCE RPC perspective an additional first argument is always present that is absent in the corresponding COM interface specification. This argument is of type ORPCTHIS, which is described below. It is placed first in the body of the Request PDU, before the actual arguments of the ORPC.

It is specifically legal for an ORPC to attempt a call a method number on a given interface which is beyond the number of methods believed by the server to be in that interface. Such calls should cause a fault.

Similarly, in a reply to an ORPC (a DCE RPC "Response PDU"), when viewed from the DCE RPC perspective, an additional first return value is always present that is absent in the corresponding COM interface specification. This argument is of type ORPCTHAT, which is also described below. It is placed first in the body of the Response PDU, before the actual return values of the ORPC.

An ORPCTHAT may also be present in a "Fault PDU." In the Connectionless (CL) Fault PDU,[108] it is placed four bytes after the 32-bit fault code which normally comprises the entire body of the PDU, thus achieving eight byte alignment for the ORPCTHAT; the intervening padding bytes are presently reserved and must be zero. The PDU body length[109] is of course set to encompass the entire body of the Fault PDU, including the ORPCTHAT. In the Connection-oriented (CO) Fault PDU, the ORPCTHAT is placed in the standard location

[106] As in DCE RPC object IDs are indeed only ever interpreted relative to a given machine, this relaxing of the DCE specification is not problematic.

[107] The IID in the interface ID field is from a logical perspective actually redundant because the IPID uniquely specifies an interface pointer (though the IID is not recoverable from just the IPID). However, an additional (optional) check to verify that the caller and callee agree on the type of the interface pointer would make the system more robust. Also, the specification of the actual IID in question eases the integration of the COM network protocol with the DCE RPC network protocol. Note that it is not expensive for callers to provide the IID since the space for the IID is allocated in the DCE RPC header, which is always transmitted anyway.

[108] For the specification of the Connectionless Fault PDU, see [CAE RPC], page 520. Page 535 of the same work describes the Connection-oriented Fault PDU.

[109] Ibid, page 516.

allocated for the "stub data."[110] In a Fault PDU of either form that results from an ORPC, if an ORPCTHAT is not present then no other data may be substituted in its here-specified location in the PDU.

.2 OXIDs, Object Exporters, & Machines

Although an IPID from a logical perspective semantically determines the server, object and interface to which a particular call should be directed, it does not by itself indicate the binding information necessary to actually carry out an invocation.

The protocol represents this "how-to" communication information in a UUID called an object exporter identifier, otherwise known as an OXID. Conceptually, an OXID can be thought of as an implementation scope for an COM object, which may be a whole machine, a given process, a thread within that process, or other more esoteric implementation scope, but the exact definition of such scopes has no bearing on the COM network protocol.

A given machine at any moment may support several OXIDs; however there is always a unique *Object Exporter service* per machine which coordinates the management of all the OXIDs on the machine. Data structures in each Object Exporter keep track of the IPIDs exported and imported by that Object Exporter. The Object Exporter resides at well-known endpoints (one per protocol, of course) on the machine. It supports a DCE RPC interface known as IObjectExporter, which is described below.

An OXID is used to determine the RPC string bindings that allow calls to reach their target IPID. Before making a call, the calling process must translate an OXID into a set of bindings that the underlying RPC implementation understands. It accomplishes by maintaining a cache of these mappings. When the destination application receives an object reference, it checks to see if it recognizes the OXID. If it does not, then it asks the source of the object reference (the server machine from which the object reference was acquired, which is not necessarily the home machine for the interface pointer) for the translation, and saves the resulting set of string bindings in a local table that maps OXIDs to string bindings.

Associated with each OXID (not each Object Exporter) is COM object termed an "OXID object." OXID objects implement (at least) the IRemUnknown interface, through which remote management of reference counts and requests for interfaces are returned.

Each machine is represented by a MID. MIDs are UUIDs and thus universally unique. The MID for a machine may (should) change when the machine reboots. However, when the MID for a machine changes, all OXIDs, OIDs, and IPIDs on that machine become invalid. MIDs are an optimization to simplify the task of determining which OXIDs are exported and pinged by which object exporters.

.3 Marshaled Interface References

The COM network protocol extends the Network Data Representation standard specified in [CAE RPC] by defining what can be thought of as a new primitive data type that can be marshaled: that of an interface reference to a COM object.[111] This is the only extension to NDR made by the COM network protocol.

A marshaled interface references is described by a type known as an OBJREF, which is described in detail below. An OBJREF in actuality has several variations:

- Null.
- Standard. A standard remote reference. Known as a STDOBJREF. A STDOBJREF contains:
    - An IPID, which uniquely specifies the interface and object.
    - An object ID (OID), which uniquely specifies the identity of the object on which the IPID is found. OIDs are UUIDs; they are universally unique.
    - An OXID, which identifies the scope where the implementation of the object is active, and can be used to reach the interface pointer.[112]

---

[110] That is, in the Connection-oriented case, the ORPCTHAT also follows four bytes of padding after the fault code; however, the fault code in the Connection-oriented Fault PDU is preceded other data not found in the Connectionless Fault PDU. Consult [CAE RPC] for further details.

[111] Whether one actually thinks of this as a new primitive data type or new compositional operator over existing data types depends on one's point of view. Both positions have some merit.

    - A reference count, indicating the number of references to this IPID that are conveyed by this marshaling. This count, though typically a value of one, may in fact be zero, one, or more (see the next section).
    - Some flags, explained later.
- Long. A standard reference, along with a set of protocol sequences and network addresses that can be useful when marshaling a proxy to give to another machine (a.k.a. the middle-man case).
- Custom. Contains a class ID (CLSID) and class-specific information.

  The Custom format gives an object control over the representation of references to itself. For example, an immutable object might be passed by value, in which case the class-specific information would contain the object's immutable data.
- Handler. A sub-case of the custom reference in which the class-specific information is standardized.

  For example, an object wishes to be represented in client address spaces by a proxy object that caches state. In this case, the class-specific information is just a standard reference to an interface pointer that the handler (proxy object) will use to communicate with the original object.
- Long Handler: Contains the same information as the handler case as well as the object resolver address. This form is needed for the same reason the long form is needed.

Interface references are ***always*** marshaled in little-endian byte order, irrespective of the byte order prevailing in the remainder of the data being marshaled.

.4 Reference Counting

In the COM network protocol, remote reference counting is conducted on per interface (per IPID), just as local reference counting is carried out on a per interface basis.

The actual increment and decrement calls are carried out using (respectively) the RemAddRef and RemRelease methods in a COM interface known as IRemUnknown found on an object associated with the each OXID, the IPID of which is returned from the function IObjectExporter::GetStringBindings (see below). In contrast to their analogues in IUnknown, RemAddRef and RemRelease can in one call increment or decrement the reference count of many different IPIDs by an arbitrary amount; this allows for greater network efficiency.

In the interests of performance, client COM implementations typically do *not* immediately translate each local AddRef and Release into a remote RemAddRef and RemRelease. Rather, the actual remote release of all interfaces on an object is typically deferred until all local references to all interfaces on that object have been released. Further, one actual remote reference count may be used to service many local reference counts; that is, the client infrastructure may multiplex zero or more local references to an interface into zero or one remote references on the actual IPID.[113]

.5 Pinging

The above reference counting scheme would be entirely adequate on its own if clients never crashed, but in fact they do, and the system needs to be robust in the face of clients crashing when they hold remote references. In a DCE RPC, one typically addresses this issue through the use of context handles. Context handles are *not* used, however, by the COM network protocol, for reasons of expense. The basic underlying technology used in virtually all protocols for detecting remote crashes is that of periodic pings. Naive use of RPC context handles would result in per object per client process pings being sent to the server. The COM network protocol architects its pinging infrastructure to reduce network traffic by relying on the cli-

---

[112] Note that the object reference does not include the interface identifier (IID), although you can't do much with the object reference without knowing the IID. The IPID does uniquely specify an IID; however, you can't algorithmically derive the IID from the IPID. This is not a problem because the IID does not have to be explicitly specified; it is either implicitly specified (by the type of the argument in a MIDL declaration) or available explicitly as another argument in the call that is carrying the polymorphic object reference (for example, IUnknown::QueryInterface).

[113] A consequence of allowing for this optimizations is that the reference count on each IPID as seen by the server may be in fact less than the total number of references on that interface as seen by all the extant clients of that interface.

ent Object Exporter implementation to do local management of client liveness detection, and having the actual pings be sent only on a machine to machine basis.

Pinging is carried out on a per-object (per OID), not a per-interface (per IPID) basis. Architecturally, at its server machine, each exported object (each exported OID) has associated with it a pingPeriod time value and a numPingsToTimeOut count which together (through their product) determine the overall amount of time known as the "ping period" that must elapse without receiving a ping on that OID or an invocation on one of its IPIDs before all the remote references to IPIDs associated with that OID can be considered to have expired. Once expiration has occurred, the interfaces behind the IPIDs can as would be expected be reclaimed solely on the basis local knowledge, though the timeliness with which this is carried out, if at all, is an implementation specific. If the server COM infrastructure defers such garbage collection in this situation (perhaps because it has local references keeping the interface pointer alive) and it later hears a ping or receives a remote call, then it knows a network partition healed. It can consider the extant remote references to be reactivated and can continue remote operations.

When interface pointers are conveyed from one client to another, such as being passed as either [in] or [out] parameters to a call, the interface pointer is marshaled in one client and unmarshaled in the other. In order to successfully unmarshal the interface, the destination client must obtain at least one reference count on the interface. This is usually accomplished by passing in the marshaled interface STDOBJREF a cRefs of (at least) one; the destination client then takes ownership of that many (more) reference counts to the indicated IPID, and the source client then owns that many fewer reference counts on the IPID. It is legal, however, for zero reference counts to be passed in the STDOBJREF; here, the destination client must (if it does not already have access to that IPID and thus have a non-zero reference count for it) before it successfully unmarshals the interface reference (concretely, e.g., before CoUnmarshalInterface returns) call to the object's exporter using IRemUnknown::RemAddRef to obtain a reference count for it.

If the destination client is in fact the object's server, then special processing is required by the destination client. The remote reference counts being passed to it should, in effect, be "taken out of circulation," as what where heretofore remote references are being converted into local references. Thus, the reference counts present in the STDOBJREF are in fact decremented from the remote reference count for the IPID in question.

Some objects have a usage model such that they do not need to be pinged at all; such objects are indicated by the presence of a flag in a STDOBJREF to an interface on the object. Objects which are not pinged in fact need not be reference counted either, though it is legal (but pointless) for a client to reference count the IPIDs of such objects.

For all other objects, assuming a non-zero ping period, it is the responsibility of the holder of an interface reference on some object to ensure that pings reach the server frequently enough to prevent expiration of the object. The frequency used by a client depends on the ping period, the reliability of the channel between the client and the server, and the probability of failure (no pings getting through and possible premature garbage-collection) that the client is willing to tolerate. The ping packet and / or its reply may both request changes to the ping period. Through this mechanism, network traffic may be reduced in the face of slow links busy servers.

.1 Delta Pinging

Without any further refinements, ping messages could be quite hefty. If machine A held 1024 remote objects (OIDs) on machine B, then it would send 16K byte ping messages. This would be annoying if the set of remote objects was relatively stable and the ping messages were the same from ping to ping.

The delta mechanism reduces the size of ping messages. It uses a ping-set interface that allows the pinging of a single set to replace the pinging of multiple OIDs.

Instead of pinging each OID, the client defines a set. Each ping contains only the set id and the list of additions and subtractions to the set. Objects that come and go within one ping period are removed from the set without ever having been added.

The pinging protocol is carried out using two methods in the (DCE) interface IObjectExporter on the Object Exporter: ComplexPing, and SimplePing. ComplexPing is used to by clients to group the set of OIDs that they must ping into UUID-tagged sets known to the server. These entire sets of OIDs can then be subsequently pinged with a single, short, call to SimplePing.

.6 QueryInterface

The IRemUnknown interface on the object-exporter specified object, in addition to servicing reference counting as described above also services QueryInterface calls for remote clients for IPIDs managed by that exporter. IRemUnknown::RemQueryInterface differs from IUnknown::QueryInterface in much the same way as RemAddRef and RemRelease differ from AddRef and Release, in that it is optimized for network access by being able to retrieve many interfaces at once.

.7 Causality ID

Each ORPC carries with it a UUID known as the causality id that connects together the chain of ORPC calls that are causally related. If an outgoing ORPC is made while servicing an incoming ORPC, the outgoing call is to have the same causality id as the incoming call. If an outgoing ORPC is made while not servicing an incoming ORPC, then a new causality id is allocated for it.

Causality ids may in theory be reused as soon as it is certain that no transitively outstanding call is still in progress which uses that call. In practice, however, in the face of transitive calls and the possibility of network failures in the middle of such call chains, it is difficult to know for certain when this occurs. Thus, pragmatically, causality ids are not reusable.

The causality id can be used by servers to understand when blocking or deferring an incoming call (supported in some COM server programming models) is very highly probable[114] to cause a deadlock, and thus should be avoided.

The causality id for maybe, idempotent, and broadcast calls must be set to null.[115] If a server makes a ORPC call while processing such a call, a new causality id must be generated as if it were a top level call.

2 Data types and structures

This following several sections present the technical details of the COM network protocol. The notation used herein is that of the Microsoft Interface Definition Language, version 2 (MIDL). MIDL is a upwardly compatible extension to the DCE IDL language specified in [CAE RPC]. Details of the MIDL language specification are available from Microsoft Corporation.

.1 DCE Packet Headers

Object RPC sits entirely on top of DCE RPC. The following list describes the elements of ORPC that are specified above and beyond DCE RPC.

- The object id field of the header must contain the IPID.
- The interface id of the RPC header must contain the IID, even though it is not needed given the IPID. This allows ORPC to sit on top of DCE RPC. An unmodified DCE RPC implementation will correctly dispatch based on IID and IPID. An optimized RPC need only dispatch based on IPID.
- An IPID uniquely identifies a particular interface on a particular object on a machine. The converse is not true; a particular interface on a particular object may only be represented by multiple IPIDs. IPIDs are unique on their OXID. IPIDs may be reused, however reuse of IPIDs should be avoided.
- Datagram, maybe, and idempotent calls are all allowed in ORPC. Interface pointers may not be passed on maybe or idempotent calls.
- Datagram broadcasts are not allowed in ORPC.
- Remote COM input synchronous calls are not allowed in ORPC.
- COM asynchronous calls are synchronous RPC calls.
- COM faults are returned in the stub fault field of the DCE RPC fault packet. Any 32 bit value may be returned. Only the following value is pre-specified:

[114] This has high probability and not certainty due to some pathological cases involving network failures. Suppose A calls B calls C calls D which will call back to A, and while D is processing its call, the link from B to C goes down, causing B to try to obtain C's services through another party E, which, as D would, calls back to A. In such situations, A may receive incoming calls from both D and E. Only the call from E is actually a potential for deadlock.

[115] Though this decision is subject to review.

RPC_E_VERSION_MISMATCH

- COM will allow DCE cancel.

All interface version numbers must be 0.0.

.2 Object RPC Base Definitions

There are several fundamental data types and structures on which the COM network protocol is built. These are defined in the MIDL file OBASE.IDL, which is included below:

```
[
    uuid(99fcfe60-5260-101b-bbcb-00aa0021347a),
    pointer_default(unique)
]

interface ObjectRpcBaseTypes
{
#ifndef DO_NO_IMPORTS
    import "wtypes.idl";
#endif

    // the object id specifed does not exist.
    const unsigned long RPC_E_INVALID_OBJECT = 0x80010150;
    // the objects exporter specifed does not exist.
    const unsigned long RPC_E_INVALID_OXID = 0x80010151;
    // the set id specifed does not exist.
    const unsigned long RPC_E_INVALID_SET = 0x80010152;
    //
    // Marshalling constants.
    const unsigned int MSHLFLAGS_NOPING     = 4;
    const unsigned int MSHLFLAGS_SIMPLEIPID = 8;
    const unsigned int MSHLFLAGS_KEEPALIVE  = 16;
    ////////////////////////////////////////////////////////

    typedef GUID MID;      // Machine Identifier
    typedef GUID OXID;     // Object Exporter Identifier
    typedef GUID OID;      // Object Identifer
    typedef GUID IPID;     // Interface Pointer Identifier
    typedef GUID SETID;    // Ping Set Identifier
    typedef GUID CID;      // Causality Identifier

    typedef REFGUID REFIPID;
    typedef REFGUID REFOXID;
    typedef REFGUID REFOID;

    const unsigned short COM_MAJOR_VERSION = 1;
    const unsigned short COM_MINOR_VERSION = 1;

    // Component Object Model version number
    typedef struct tagCOMVERSION
    {
        unsigned short MajorVersion;  // Major version number
        unsigned short MinorVersion;  // Minor version number
    } COMVERSION;


    // STRINGARRAYS are the return type for arrays of network addresses,
    // arrays of endpoints and arrays of both used in many ORPC interfaces

    const unsigned short NCADG_IP_UDP   = 0x08;
    const unsigned short NCACN_IP_TCP   = 0x07;
    const unsigned short NCADG_IPX      = 0x0E;
    const unsigned short NCACN_SPX      = 0x0C;
    const unsigned short NCACN_NB_NB    = 0x12;
    const unsigned short NCACN_DNET_NSP = 0x04;
    const unsigned short NCALRPC        = 0x10;
    // const unsigned short MSWMSG         = 0x01; // note: not a real tower id.

    // this is the return type for arrays of string bindings or protseqs
    // used by many ORPC interfaces
```

```
typedef struct tagSTRINGARRAY
{
    unsigned long size;    // total size of array

     // array of NULL terminated wchar_t strings with two NULLs at the end.
     // The first word of each string is the protocol ID (above) and the
     // rest of the string is the network address[endpoint].

    [size_is(size)] unsigned short awszStringArray[];
} STRINGARRAY;

// flag values for OBJREF
const unsigned long OBJREF_STANDARD  = 1;    // standard marshalled objref
const unsigned long OBJREF_HANDLER   = 2;    // handler marshalled objref
const unsigned long OBJREF_LONGSTD   = 4;    // long form objref
const unsigned long OBJREF_LONGHDLR  = 8;    // long form handler objref
const unsigned long OBJREF_CUSTOM = 16;   // custom marshalled objref

// flag values for a STDOBJREF.
// Should be an enum but DCE IDL does not support sparse enumerators.
// OXRES1 - OXRES4 are reserved for the object exporters use only,
// object importers should ignore them and not enforce MBZ.
const unsigned long SORF_NOPING   = 1;    // Pinging is not required
const unsigned long SORF_OXRES1   = 8;    // reserved for exporter
const unsigned long SORF_OXRES2   = 16;   // reserved for exporter
const unsigned long SORF_OXRES3   = 32;   // reserved for exporter
const unsigned long SORF_OXRES4   = 64;   // reserved for exporter

// Reserved flag values for a STDOBJREF.
const unsigned long SORF_FREETHREADED = 2;  // Proxy may be used on any thread

// standard object reference
typedef struct tagSTDOBJREF
{
    unsigned long  flags;      // STDOBJREF flags
    unsigned long  cRefs;      // count of references passed
    IPID           ipid;       // ipid of Interface
    OID            oid;        // oid of object with this ipid
    OXID           oxid;       // oxid of server with this oid
} STDOBJREF;

// format of a marshalled interface pointer
typedef struct tagOBJREF
{
unsigned long flags; // OBJREF flags (see above)

[switch_is(flags), switch_type(unsigned long)] union
{
    [case(OBJREF_STANDARD)]
    STDOBJREF std;     // standard objref

    [case(OBJREF_LONGSTD)] struct
    {
    STDOBJREF   std;   // standard objref
    STRINGARRAY saResAddr; // resolver address
    } longstd;

    [case(OBJREF_HANDLER)] struct
    {
    STDOBJREF std;     // standard objref
    CLSID    clsid;    // Clsid of handler
    } handler;

    [case(OBJREF_LONGHDLR)] struct
    {
    STDOBJREF   std;   // standard objref
    CLSID      clsid;  // Clsid of handler (or GUID_NULL)
    STRINGARRAY saResAddr; // resolver address
    } longhdlr;
```



```
    [case(OBJREF_CUSTOM)] struct
    {

    CLSID        clsid;   // Clsid of unmarshaling code
    unsigned long size;    // size of data that follows
    [size_is(size), ref] byte *pData;
    } custom;

} u_objref;
} OBJREF;


// enumeration of additional information present in the call packet.
// Should be an enum but DCE IDL does not support sparse enumerators.

const unsigned long INFO_NULL      = 0;  // no additional info in packet
const unsigned long INFO_LOCAL     = 1;  // call is local to this machine
const unsigned long INFO_RESERVED1 = 2;  // reserved for local use
const unsigned long INFO_RESERVED2 = 4;  // reserved for local use
const unsigned long INFO_RESERVED3 = 8;  // reserved for local use
const unsigned long INFO_RESERVED4 = 16; // reserved for local use


// Extension to implicit parameters.
typedef struct tagORPC_EXTENT
{
    GUID                   id;     // Extension identifier.
unsigned long              size;   // Extension size.
[size_is((size+7)&~7)] byte data[]; // Extension data.
} ORPC_EXTENT;


// Array of extensions.
typedef struct tagORPC_EXTENT_ARRAY
{
unsigned long size;   // num extents
[size_is((size+1)&~1,), unique] ORPC_EXTENT **extent; // extents
} ORPC_EXTENT_ARRAY;


// implicit 'this' pointer which is the first [in] parameter on
// every ORPC call.
typedef struct tagORPCTHIS
{
COMVERSION version;   // COM version number
unsigned long  flags;  // INFO flags for presence of other data
unsigned long  reserved1; // set to zero
CID    cid;       // causality id of caller

    // Extensions.
[unique] ORPC_EXTENT_ARRAY *extensions;
} ORPCTHIS;


// implicit 'that' pointer which is the first [out] parameter on
// every ORPC call.
typedef struct tagORPCTHAT
{
unsigned long  flags; // INFO flags for presence of other data

    // Extensions.
[unique] ORPC_EXTENT_ARRAY *extensions;
} ORPCTHAT;


// OR information associated with each OXID.
typedef struct tagOXID_INFO
{
DWORD      dwTid;     // thread id of object exporter
DWORD      dwPid;     // process id of object exporter
IPID       ipidRemUnknown;  // IRemUnknown IPID for object exporter
```

```
        [unique] STRINGARRAY *psa;   // protocol id's and partial string bindings
        } OXID_INFO;
    }
```

.3 OBJREF

An OBJREF is the data type used to represent an actual marshaled object reference. An OBJREF can either be empty or assume one of five variations, depending on the degree to which the object being marshaled uses the hook architecture (IMarshal, etc.) in the marshaling infrastructure. The OBJREF structure is a union consisting of a switch flag followed by the appropriate data.

.1 OBJREF_STANDARD

Contains one interface of an object marshaled in standard form. The data that follows the switch flag is a STDOBJREF structure (described below).

.2 OBJREF_HANDLER

A marshaling of an object that wishes to use handler marshaling. For example, an object wishes to be represented in client address spaces by a proxy object that caches state. In this case, the class-specific information is just a standard reference to an interface pointer that the handler (proxy object) will use to communicate with the original object. See the IStdMarshalInfo interface.

Member	Type	Semantic
std	STDOBJREF	A standard object reference used to connect to the source object.
Clsid	CLSID	The CLSID of handler to create in the destination client.

.3 OBJREF_LONGSTD

An interface marshaled on an object in long form. Contains a standard reference, along with a set of protocol sequences and network addresses that can be used to bind to an OXID resolver that is able to resolve the OXID in the STDOBJREF. This is useful when marshaling a proxy to give to another machine (a.k.a. the "middleman" case). The marshaling machine can specify the saResAddr for the resolver on the server machine so that the unmarshaler does not need to call the marshaler (middleman) back to get this information. Further, the marshaler does not need to keep the OXID in its cache beyond the lifetime of its own references in order to satisfy requests from parties that it just gave the OBJREF to.

Member	Type	Semantic
std	STDOBJREF	A standard object reference used to connect to the source object.
SaResAddr	STRINGARRAY	The resolver address.

.4 OBJREF_LONGHDLR

Contains the same information as the handler case as well as the object resolver address. This form is needed for the same reason OBJREF_LONGSTD is needed.

Member	Type	Semantic
std	STDOBJREF	A standard object reference used to connect to the source object.
Clsid	CLSID	The class ID of the handler.
SaResAddr	STRINGARRAY	The resolver address.

.5 OBJREF_CUSTOM

A marshaling of an object which supports custom marshaling. The Custom format gives an object control over the representation of references to itself. For example, an immutable object might be passed by value, in which case the class-specific information would contain the object's immutable data. See the IMarshal interface.

Member	Type	Semantic
clsid	CLSID	The CLSID of the object to create in the destination client.
size	unsigned long	The size of the marshaled data provided by the source object and passed here in pData.
pData	byte*	The data bytes that should be passed to IMarshal::UnmarshalInterface on a new in-

stance of class clsid in order to initialize it and complete the unmarshal process.

.4 STDOBJREF

An instance of a STDOBJREF represents a COM interface pointer that has been marshaled using the standard COM network protocol. A STDOBJREF in general can only be interpreted in the context of an outstanding ORPC, for it may contain an OXID unknown to the machine on which it is unmarshaled, and it is the only the machine which is making outstanding call which is guaranteed to be able to provide the binding information for the OXID.

The members and semantics of the STDOBJREF structure are as follows:

Member	Semantic
flags	Flag values taken from the enumeration SORFFLAGS. These are described below.
Crefs	The number of reference counts on ipid that being transferred in this marshaling.
Ipid	The IPID of the interface being marshaled.
Oid	The OID of the object to which ipid corresponds.
Oxid	The OXID of the server that owns this OID.

The various SORFLAGS values have the following meanings. The SORF_OXRESx bit flags are reserved for the object exporter's use only, and must be ignored by object importers. They need not be passed through when marshaling an interface proxy.

Flag	Value	Meaning
SORF_NOPING	1	This OID does not require pinging. Further, all interfaces on this OID, including this IPID, need not be reference counted. Pinging and reference counting on this object and its interfaces are still permitted, however, though such action is pointless.
SORF_OXRES1	8	Reserved for exporter.
SORF_OXRES2	16	Reserved for exporter.
SORF_OXRES3	32	Reserved for exporter.
SORF_OXRES4	64	Reserved for exporter.

.5 ORPCTHIS

In every Request PDU that is an ORPC, the body (CL case) or the stub data (CO case) which normally contains the marshaled arguments in fact begins with an instance of the ORPCTHIS structure. The marshaled arguments of the COM interface invocation follow the ORPCTHIS; thus, viewed at the DCE RPC perspective, the call has an additional first argument. The ORPCTHIS is padded with zero-bytes if necessary to achieve an overall size that is a multiple of eight bytes; thus, the remaining arguments are as a whole eight byte aligned.

As in regular calls, the causality id must be propagated. If A calls ComputePi on B, B calls Release on C (which gets converted to RemRelease), and C calls Add on A, A will see the same causality id that it called B with.

Member	Type	Semantic
version	COMVERSION	The version number of the COM protocol used to make this particular ORPC. The initial value will be 1.1. Each packet contains the sender's major and minor ORPC version numbers. The client's and server's major versions must be equal. Backward compatible changes in the protocol are indicated by higher minor version numbers. Therefore, a server's minor version must be greater than or equal to the client's. However, if the server's minor version exceeds the client's minor version, it must return the client's minor version and restrict its use of the protocol to the minor version specified by the client. A protocol version mismatch causes the RPC_E_VERSION_MISMATCH ORPC fault to be returned.
flags	unsigned long	Flag values taken from the enumeration ORPCINFOFLAGS. These are elaborated below.
reserved	unsigned long	Must be set to zero.
cid	CID	The causality id of this ORPC. See comments below.
extensions	ORPC_EXTENT_ARRAY*	The body extensions, if any, passed with this call. Body extensions are GUID-tagged blobs of data which are marshaled as an array of bytes. Extensions are always marshaled with initial eight byte alignment. Body extensions which are presently defined are described below.

The various ORPCINFOFLAGS have the following meanings.

Flag	Meaning
INFO_NULL	(Not a real flag. Merely a defined constant indicating the absence of any flag values.)
INFO_LOCAL	The destination of this call is on the same machine on which it originates. This value is never to be specified in calls which are not in fact local.
INFO_RESERVED1	If INFO_LOCAL is set, then reserved for local use; otherwise, reserved for future use.
INFO_RESERVED2	If INFO_LOCAL is set, then reserved for local use; otherwise, reserved for future use.
INFO_RESERVED3	If INFO_LOCAL is set, then reserved for local use; otherwise, reserved for future use.
INFO_RESERVED4	If INFO_LOCAL is set, then reserved for local use; otherwise, reserved for future use.

Implementations may use the local and reserved flags to indicate any extra information needed for local calls. Note that if the INFO_LOCAL bit is not set and any of the other bits *are* set then the receiver should return a fault.

Comments

The cid field contains the causality id.[116] Each time a client makes a call, a new causality id is generated. If a server makes a call while processing a request from a client, the new call must have the same causality id. This allows simple servers to avoid working on more then one thing at a time (for example A calls B calls A again, meanwhile C tries to call A with a new causality id). It tells the server that he is being called because he asked someone to do something for him. There are several interesting exceptions.

- The causality id for maybe and idempotent calls must be set to CID_NULL. If a server makes a ORPC call while processing such a call, a new causality id must be generated.
- In the face of network failures, the same causality id may end up in use by two independent processes at the same time. If A calls B calls C calls D and C fails, both B and D can independently, simultaneously make calls to E with the same causality id.

The extensions field contains extensions to the channel header. Two are currently defined for Microsoft's implementation of this protocol (described below). Other implementations may define their own extensions with their own UUIDs. Implementations should skip over extensions they do not recognize or wish to support. Note that in order to force the ORPCTHIS header to be 8 byte aligned an even number of extensions must be present and the size of the extension data must be a multiple of 8.

.6 ORPCTHAT

In every Response PDU that is an ORPC, the body (CL case) or the stub data (CO case) which normally contains the marshaled output parameters in fact begins with an instance of the ORPCTHAT structure. The marshaled output parameters of the COM interface invocation follow the ORPCTHAT; thus, viewed at the DCE RPC perspective, the call has an additional output parameters. The ORPCTHAT is padded with zero-bytes if necessary to achieve an overall size that is a multiple of eight bytes; thus, the remaining output parameters as a whole eight byte aligned.

Member	Type	Semantic
flags	unsigned long	Flag values taken from the enumeration ORPCINFOFLAGS. These are elaborated above.
extensions	ORPC_EXTENT_ARRAY*	The body extensions, if any, returned by this call. Body extensions are GUID-tagged blobs of data which are marshaled as an array of bytes. Extensions are always marshaled with initial eight byte alignment.

.7 Debug Information Body Extension

This extension aids in debugging ORPC. In particular it is designed to allow single stepping over an ORPC call into the server and out of the server into the client. See << REFERENCE >> for more details. This extension is identified by the UUID {f1f19680-4d2a-11ce-a66a-0020af6e72f4}.

[116] In several early specifications the term *logical thread id* was used instead of causality id. The name was changed because the term logical thread id implies a single threaded model that is not guaranteed.

.8 Extended Error Info Body Extension

The extended error information body extension conveys extended error information concerning the original root cause of a error back to a caller can deal with it. It is intended that this error information is suitable for displaying information to a human being who is the user; this information is not intended to be the basis for logic decisions in a piece of client code, for doing so couples the client code to the implementation of the server. Rather, client code should act semantically only on the information returned through the interface that it invokes. See also the ISupportErrorInfo, IErrorInfo, and ICreateErrorInfo interfaces.

This extension is identified by the UUID f1f19681-4d2a-11ce-a66a-0020af6e72f4. It is only semantically useful in Response and Fault PDUs.

There are three variations of the error information. The first of these can be used in a local context, where a) the server knows it can reliably specify a path name to a file that can in fact be understood by the client, and b) the server knows the language of interest to the client. This first variation specifies the following members:

Member	Description
wszErrorString	An error string suitable for display to a human user.
wszHelpFile	A path to a help file that can give additional information concerning the error.
ulHelpContext	A help context topic within that help file.

The second variation is suitable for use in remote situations where one or the other of the requirements of the use of the first variation cannot be upheld. This second variation specifies the following members:

Member	Description
uuidErrorSemantic	A UUID signifying the semantic of the error.
ulErrorSemantic	A four-byte quantity that qualifies the error semantic. The semantics of these four bytes are completely determined by the uuidErrorSemantic.
cbData	The size of the data passed in pbData.
pbData	Data associated with the error marshaled as an array of bytes. The interpretation of these bytes is governed by the uuidErrorSemantic.
wszErrorString	An error string suitable for display to a human user. This is in fact of somewhat little use, as the server returning the error can usually only guess as to the appropriate language in which to form this string. However, the ability to pass such as string as a last resort is provided here.
lcidErrorString	The locale context in which the error string, if any, is formed. Locale constants are as in the Microsoft Win32 API.

The third variation allows for extensibility of the error information being passed. It specifies an object reference (an OBJREF). In practice, this reference most always contains a custom marshaled object, though this is not required.

3 IRemUnknown interface

The IRemUnknown interface is used by remote clients for manipulating reference counts on the IPIDs that they hold and for obtaining additional interfaces on the objects on which those IPIDs are found. This interface is implemented by the COM "OXID object" associated with each OXID (nb. not each Object Exporter). The IPID for the IRemUnknown interface on this object is returned from IObjectExporter::ResolveOxid; see below. An OXID object need never be pinged; its interfaces (this IPID included) need never be reference counted.

IRemUnknown is specified as follows (REMUNK.IDL):

```
//+-------------------------------------------------------------------------
//
//  Microsoft Windows
//  Copyright (C) Microsoft Corporation, 1992 - 1995.
//
//  File:   remunk.idl
//
//  The remote version of IUnknown.  Once instance of this interface exists
//  per OXID (whether an OXID represents either a thread or a process is
//  implementation specific). This interface is passed along during OXID
//  resolution.  It is used by clients to query for new interfaces, get
//  additional references (for marshalling), and release outstanding
//  references.
//
```

```
//+-------------------------------------------------------------------------
[
    object,
    uuid(99fcff28-5260-101b-bbcb-00aa0021347a)
]

import "unknwn.idl";

interface IRemUnknown : IUnknown
{
    // return structure from a QI call
    typedef struct tagREMQIRESULT
    {
        HRESULT    hResult;    // result of call
        STDOBJREF  std;        // data for returned interface
    } REMQIRESULT;

    HRESULT RemQueryInterface
    (
        [in] IPID              ipid,  // interface to QI on
        [in] unsigned long cRefs,  // count of AddRefs requested for each interface
        [in] unsigned short    cIids,  // count of IIDs that follow
        [in, size_is(cIids)] IID *iids, // IIDs to query for
        [out, size_is(,cIids)] REMQIRESULT **ppQIResults // results returned
    );


    // structure passed to AddRef/Release to specify interface and count of
    // references to Add/Release.
    typedef struct tagREMINTERFACEREF
    {
        IPID       ipid;      // ipid to AddRef/Release
        unsigned long  cRefs;     // number of refs to add/release
    } REMINTERFACEREF;

    HRESULT RemAddRef
    (
        [in] unsigned short cInterfaceRefs,
        [in, size_is(cInterfaceRefs)] REMINTERFACEREF InterfaceRefs[]
    );

    HRESULT RemRelease
    (
        [in] unsigned short cInterfaceRefs,
        [in, size_is(cInterfaceRefs)] REMINTERFACEREF InterfaceRefs[]
    );
}
```

Comments

References are kept per interface rather then per object.

.1 IRemUnknown::RemQueryInterface

HRESULT IRemUnknown::RemQueryInterface(ipid, cIids, cRefs, iids, ppQIResults)

QueryInterface for and return the result thereof for zero or more interfaces from the interface behind the IPID ipid. ipid must designate an interface derived from IUnknown (recall that all remoted interfaces must derive from IUnknown). The QueryInterface calls on the object that are used to service this request are conducted on this interface instance, not any other IUnknown instance that the object may happen to have. Thus if the client calls IFoo->QueryInterface rather then pIUnknown->QueryInterface, RemQueryInterface will also call QueryInterface on IFoo.

Argument	Type	Description
ipid	IPID	The interface on an object from whom more interfaces are desired.
cRefs	REFCNT	The number of references sought on each of the returned IIDs.
clids	USHORT	The number of interfaces being requested.
iids	IID*	The list of IIDs that name the interfaces sought on this object.
ppQIResults	REMQIRESULT**	The place at which the array of the results of the various QueryInterface calls are returned.

Return Value	Meaning
S_OK	Success. An attempt was made to retrieve each of the requested interfaces from the indicated object; that is, QueryInterface was actually invoked for each IID.
E_INVALIDARG	One or more arguments (likely ipid) were invalid. No result values are returned.
E_UNEXPECTED	An unspecified error occurred. No result values are returned.

The REMQIRESULT structure contains the following members:

Member	Type	Semantic
hResult	HRESULT	The result code from the QueryInterface call made for the requested IID.
std	STDOBJREF	The data for the returned interface. Note that if hResult indicates failure then the contents of STDOBJREF are undefined.

.2 IRemUnknown::RemAddRef

HRESULT IRemUnknown::RemAddRef(cInterfaceRefs, rgRefs)

Obtain and grant ownership to the caller of one or more reference counts on one or more IPIDs managed by the corresponding OXID.

Argument	Type	Description
cInterfaceRefs	unsigned short	The size of the rgRefs array.
rgRefs	REMINTERFACEREF	An array of IPID, cRefs pairs, cInterfaceRefs large. Each IPID indicates an interface managed by this OXID on whom more reference counts are sought. The corresponding reference count (cRefs), which may not be zero (and thus is one or more), indicates the number of reference counts sought on that IPID.

Return Value	Meaning
S_OK	Success. An attempt was made to retrieve each of the requested interfaces.
E_INVALIDARG	One or more of the IPIDs indicated were not in fact managed by this OXID, or one or more of the requested reference counts was zero. **None** of the requested reference counts have been granted to the caller; the call is a no-op.
E_UNEXPECTED	An unspecified error occurred. It is unknown whether any or all of the requested reference counts have been granted.

Comments

A useful optimization is for a caller to RemAddRef more than needed. When a process receives an out marshaled interface, it receives one reference count. If the process wishes to pass that interface as an out parameter, it must get another reference to pass along. Instead, the process (or middleman) should get a large number of references. Then if the interface is passed out multiple times, no new remote calls are needed to gain additional references.

A marshaler may optionally specify more than one reference in the STDOBJREF when marshaling an interface. This allows the middle man case to pre-fill its cache of references without making an extra RemAddRef call. The number of references passed is always specified in the STDOBJREF field.

.3 IRemUnknown::RemRelease

HRESULT IRemUnknown::RemRelease(cInterfaceRefs, rgRefs)

Argument	Type	Description
cInterfaceRefs	USHORT	The size of the rgRefs array.
rgRefs	REMINTERFACEREF	An array of IPID, cRefs pairs, cInterfaceRefs large. Each IPID indicates an interface managed by this OXID on whom more reference counts are being returned. The corresponding reference count, which may not be zero (and thus is one or more), indicates the number of reference counts returned on that IPID.

Return Value	Meaning
S_OK	Success. An attempt was made to retrieve each of the requested interfaces.
E_INVALIDARG	One or more of the IPIDs indicated were not in fact managed by this OXID, or one or more of the requested reference counts was zero. **None** of the offered reference counts have been accepted by the server; the call is a no-op.
E_UNEXPECTED	An unspecified error occurred. It is unknown whether any or all of the offered reference counts have been accepted.

4 The Object Exporter

Each machine that supports the COM network protocol supports a one-per-machine service known as the machine's 'Object Exporter.' Communication with an Object Exporter is via a DCE RPC, not an ORPC. To ensure connectivity, the Object Exporter resides at well-known endpoints. It is proposed that the Object Exporter either (1) make use of the same endpoints allocated for the DCE RPC Endpoint Mapper (listed below)[117], implying typically that these services are written within the same process on a given machine, or alternately and less preferably (2) that the Object Exporter reside at a different set of well-known endpoints

Release ownership of one or more reference counts on one or more IPIDs managed by the corresponding OXID.

TBD.

The Object Exporter performs several services:

- It caches and returns to clients when asked the string bindings necessary to connect to OXIDs of exported objects for which this machine is it either itself a client or is the server.
- It receives pings from remote client machines to keep its own objects alive.

These services are carried out through an RPC interface (not a COM interface) known as IObjectExporter.

An Object Exporter may be asked for the information required to connect to one of two different kinds of OXIDs, either the OXIDs associated with its own objects, or the OXIDs associated with objects for which it itself is a client, and which it has passed on to a second client machine. This second case, where one marshals an object from one client machine to a second, is colloquially referred to the "middleman" case. In the middleman case, the exporter is required to retain the connection information associated with the OXIDs that it passes on until it is certain that that the second client machine no longer needs them. More on this below.

[117] See [CAE RPC], Appendix H, p 613.

Protocol String Name(s)	Description	Object Exporter End Point[118]	Endpoint Mapper End Point
ncadg_ip_udp, ip	CL over UDP/IP	TBD	135
ncacn_ip_tcp	CO over TCP/IP	TBD	135
ncadg_ipx	CL over IPX	TBD	not yet listed
ncacn_spx	CO over SPX	TBD	not yet listed
ncacn_nb_nb	CO over NetBIOS over NetBEUI	TBD	not yet listed
ncacn_nb_tcp	CO over NetBIOS over TCP/IP	TBD	135
ncacn_np	CO over Named Pipes	TBD	not yet listed
ncacn_dnet_nsp	CO over DECNet Network Services Protocol (DECnet Phase IV)	TBD	69
ncacn_osi_dna	CO over Open Systems Interconnection (DECNet Phase V)	TBD	69
ncadg_dds, dds	CL over Domain Datagram Service	TBD	12
ncalrpc	Local procedure call	N/A	N/A

Table 2. Object Exporter Well-known Endpoints

IObjectExporter interface is defined as follows (OBJEX.IDL):

```
//+-------------------------------------------------------------------------
//
//  Microsoft Windows
//  Copyright (C) Microsoft Corporation, 1992 - 1995.
//
//  File:      objex.idl
//
//  Synopsis:      Interface implemented by object exporters.
//
//  This is the interface that needs to be supported by hosts that export
//  objects. Only one instance of this interface can be exported by the host.
//
//  An object exporter needs to be able to:
//  1. return string bindings that can be used to talk to objects it
//     has exported
//  2. receive pings from object importers to keep the objects alive
//
//--------------------------------------------------------------------------
[
    uuid(99fcfec4-5260-101b-bbcb-00aa0021347a),
    pointer_default(unique)
]

interface IObjectExporter
{
    import "obase.idl";

    // Method to get the protocol sequences, string bindings and machine id
    // for an object server given its OXID.

    [idempotent] error_status_t ResolveOxid
    (
    [in]       handle_t       hRpc,
    [in]       OXID           *pOxid,
    [in]       unsigned short cRequestedProtseqs,
    [in,  ref, size_is(cRequestedProtseqs)]
               unsigned short arRequestedProtseqs[],
    [out, ref] MID            *pmid,
    [out, ref] STRINGARRAY   **psaOxidBindings,
    [out, ref] IPID           *pipidRemUnknown
    );

    // Simple ping is used to ping a Set. Client machines use this to inform
    // the object exporter that it is still using the items inside the set.
    // Returns S_TRUE if the SetId is known by the object exporter,
    // S_FALSE if not.
```

[118] Only if reuse of the DCE RPC Endpoint Mapper endpoints is unacceptable.

```
    [idempotent] error_status_t SimplePing
    (
    [in]  handle_t  hRpc,
    [in]  SETID    *pSetId
    );

    // Complex ping is used to create sets of OIDs to ping. The whole set
    // can subsequently be pinged using SimplePing, thus reducing network
    // traffic.

    [idempotent] error_status_t ComplexPing
    (
    [in]  handle_t        hRpc,
    [in]  SETID          *pSetId,
    [in]  unsigned short  SequenceNum,
    [in]  unsigned short  SetPingPeriod,
    [in]  unsigned short  SetNumPingsToTimeout,
    [out] unsigned short *pReqSetPingPeriod,
    [out] unsigned short *pReqSetNumPingsToTimeout,
    [in]  unsigned short  cAddToSet,
    [in]  unsigned short  cDelFromSet,
    [in,  unique, size_is(cAddToSet)]   GUID AddToSet[],   // add these OIDs to the set
    [in,  unique, size_is(cDelFromSet)] GUID DelFromSet[]  // remove these OIDs from the set
    );
}
```

.1 IObjectExporter::ResolveOxid

```
[idempotent] error_status_t ResolveOxid
    (
    [in]       handle_t        hRpc,
    [in]       OXID           *pOxid,
    [in]       unsigned short  cRequestedProtseqs,
    [in,  ref, size_is(cRequestedProtseqs)]
               unsigned short  arRequestedProtseqs[],
    [out, ref] MID            *pmid,
    [out, ref] STRINGARRAY   **psaOxidBindings,
    [out, ref] IPID           *pipidRemUnknown
    );
```

Return the string bindings necessary to connect to a given OXID object.

On entry, arRequestedProtseqs contains the protocol sequences the client is willing to use to reach the server. These should be decreasing order of protocol preference, with no duplicates permitted. Local protocols (such as "ncalrpc") are not permitted.

On exit, psaOxidBindings contains the string bindings that may be used to connect to the indicated OXID; if no such protocol bindings exist which match the requested protocol sequences, NULL may be returned. The returned string bindings are in decreasing order of preference of the server, with duplicate string bindings permitted (and not necessarily of the same preferential priority), though of course duplicates are of no utility. Local protocol sequences may not be present; however, protocol sequences that were not in the set of protocol sequences requested by the client may be. The string bindings returned need not contain endpoints; the endpoint mapper will be used as usual to obtain these dynamically.

If a ResolveOxid call is received for which the recipient Object Exporter is a middleman, the action required of the middleman depends on how the ordered list of requested protocol sequences (arRequestedProtseqs) relate to lists of protocol sequences previously known by the middleman to have been previously requested of the server. If the list of requested protocol sequences is a (perhaps non-proper) subset in order of a protocol sequence list previously requested of the server, then the corresponding cached string bindings may be returned immediately to the caller without actually communicating with the server. Otherwise, the actual psaRequestedProtseqs must be forwarded to the server, and the returned string bindings propagated back to the client. In such cases, it behooves the middleman to cache the returned string bindings for use in later calls.

In order to support the middleman case, Object Exporters are required to remember the OXID mapping information for remote OXIDs they have learned for some period of time beyond when they themselves have released all references to objects of this OXID. Let *t* be the full time-out period (ping period × number of pings to time-out) for some OID (any OID) for which this Object Exporters is a client and which resides in

OXID. Then the Object Exporter must keep the binding information for OXID for at least an amount of time *t* following the release of a the local reference to any object in OXID.[119]

Returned through pmid is an identifier that uniquely identifies the machine. Clients can use this to learn which OXIDs are collocated on the same machine, and thus which OIDs may be appropriately grouped together in ping sets (see ComplexPing). The machine identifier is guaranteed not to change so long as there are remote references to objects on the machine which remain valid. Thus, specifically, the machine id may change as the machine reboots.

ResolveOxid also informs the caller of the IPID of the OXID object associated with this OXID.

Argument	Type	Description
hRpc	handle_t	An RPC binding handle used to make the request.
pOxid	OXID*	The OXID for whom string bindings are requested. . The OXID may or may not represent a process on the machine that receives the ResolveOxid call
cRequestedProtseqs	unsigned short	The number of protocol sequences requested.
arRequestedProtseqs	unsigned short[]	arRequestedProtseqs must be initialized with all the protocol id's the client is willing to use to reach the server. It cannot contain local protocol sequences. The object exporter must take care of local lookups privately. The protocol sequences are in order of preference or random order. No duplicates are allowed. See the Lazy Use Protseq section for more details.
pmid	MID*	The machine identifier associated with OXID.
psaOxidBindings	STRINGARRAY**	The string bindings supported by this OXID, in preferential order. Note that these are Unicode strings.
pipidRemUnknown	IPID*	The IPID to the IRemUnknown interface the OXID object for this OXID.

Return Value	Meaning
S_OK	Success. The requested information was returned.
RPC_E_INVALID_OBJECT	
RPC_E_INVALID_OXID	This OXID is unknown to this Object Exporter, and thus no information was returned.
RPC_E_SERVER_DIED	
E_OUTOFMEMORY	
E_UNEXPECTED	An unspecified error occurred. Some of the requested information may not be returned.

Comments

Since the object exporter ages string bindings and discards them, object references are transient things. They are not meant to be stored in files or otherwise kept persistently. The preferred method of storing persistent references will depend on the activation models available. On platforms that support them, monikers should be used for any persistent reference. In any case, well known object references can be constructed from well known string bindings, IPIDs and OIDs.

Conversely, since object references are aged, it is the responsibility of each client to unmarshal them and begin pinging them in a timely fashion.

The basic use of the ResolveOxid method is to translate an OXID to string bindings. Put another way, this method translates an opaque process and machine identifier to the information needed to reach that machine and process. There are four interesting cases: looking up an OXID the first time an interface is unmarshaled on a machine, looking up an OXID between a pair of machines that already have connections, looking up an OXID from a middleman, and looking up string bindings with unresolved endpoints (lazy use protseq). Another interesting topic is garbage collection of stored string binding vectors.

Lookup Between Friends

[119] This time-out period does not *guarantee* that OXID information will not be discarded before clients may need it, but is a very good heuristic, and indeed better than a hard-coded time-out value.

OX C	Process D	OX E	Process F	Process G
	call F			
			pass out ref to G	
	receive out ref to G			
	ask local OX to resolve OXID G			
ask OX E to resolve OXID G				
		look up G and return G's end-points.		
Construct and cache string binding vector for G				
return results to D				
	ready to call G directly			

The case of a lookup between two machines that have already established communication is the easiest. In this scenario there are two machines, A and B. Process D already has an interface pointer to process F. Object exporter C already knows the string bindings for object exporter E and process F, but not process G. Object exporter E knows the string bindings for all the servers on its machine, i.e. processes F and G. Process D calls process F and gets a reference to process G. Since process D has never seen the OXID for G before, it asks its local object exporter to resolve G. Process D also has to tell object exporter C where it got the reference from, in this case, process F. Object exporter C does not recognize the OXID G. However it does recognize the OXID F and knows the object exporter E is on the same machine as process F. So OX C calls ResolveOxid on OX E. OX E recognizes G and passes the string bindings back to OX C with the machine id B. OX C caches this information so that if D ever gets a reference from G, it knows who to ask to resolve that reference.

My First Lookup

The previous example assumes that OX C already knows about OX E and process D is already talking to process F. Setting up the first connection between D and F (as well as C and E) is a tricky business known as activation. ORPC as described in this specification does not include activation models. Thus different vendors may have different activation models. However there is one basic form of activation shared by all ORPC. If two processes can communicate via DCE RPC, they can pass long standard object references. While this is not expected to be a common form of activation, it is a simple one that should certainly work across all ORPC implementations. Thus if D and E have established DCE RPC (or raw RPC) communication, they can bootstrap ORPC communication as follows.

OX C	Process D		OX E	Process F
			register endpoints and OXID for F	
	call F with raw RPC			
				pass an out ref to F
				pass IID as additional parameter
	tell C the OXID_INFO and MID for F. Include network address(es).			
compute the string bindings for OX E from F				

ask E to resolve F to get machine id and endpoints for F					
			return endpoints and machine id		
	ready to call F via ORPC				

This example points out that there has to be a local interface between processes and the local object exporter.

Middleman Lookup

The next case shows how lookup works between multiple machines. Suppose that E has a reference to G and G has a reference to I. Similarly, D knows about F and G and F knows about H and I. What happens if G passes a reference to I over to E?

OX D	Process E	OX F	Process G	OX H	Process I
	call G				
			return a long reference to I		
	ask D to lookup I				
Since it's a long ref to I call ResolveOxid on H.					
				Return endpoints to I	
Compute string bindings to I from endpoints and network addresses.					
return string bindings to E					
	ready to call I				

Note that when process G returned a reference to I, it used he long form of the OBJREF which includes the protocol id's and network addresses of the OXID resolver for process I (in this example, the addresses for OX H). This would results in OX D calling OX H directly, rather than needing to call OX F. The advantage of this is that if no references to process I needed by OX F, it could remove it from its OXID cache at any time, rather than keeping it around at least until OX D has had a chance to call it back to resolve OXID I.

Lazy Use Protseq

In a homogeneous network, all machines communication via the same protocol sequence. In a heterogeneous network, machines may support multiple protocol sequences. Since it is often expensive in resources to allocate endpoints (RpcServerUseProtseq) for all available protocol sequences, ORPC provides a mechanism where they may be allocated on demand. To implement this extension fully, there are some changes in the server. However, changes are optional. If not implemented, ORPC will still work correctly if less optimally in heterogeneous networks.

There are two cases: the server implements the lazy use protocol or it does not.

If the server is using the lazy use protseq protocol, the use of ResolveOxid is modified slightly. When the client OX calls the server OX, it passes the requested protseq vector. If none of the requested protseqs have endpoints allocated in the server, the server OX performs some local magic to get one allocated.

If the server does not implement the lazy use protseq protocol, then all protseqs are registered by the server and contain complete endpoints. However, if they are not, the endpoint mapper can be used to forward calls to the server. This requires that all server IIDs are registered in the endpoint mapper. It also allows a

different lazy use protseq mechanism. The endpoint mapper can perform some local magic to force the server to allocate an endpoint. This is less efficient since no OXs ever learn the new endpoints.

The client will always pass in a vector of requested protseqs which the server can ignore if it does not implement the lazy use protseq protocol.

Aging String Bindings

Each object exporter must keep all the string bindings for references to remote machines as well as string bindings for all processes that are ORPC servers on its machines. However, unless the middle man marshaler always marshals proxy interfaces using the long form OBJREF, string bindings cannot be discarded as soon as remote references are. In the middleman example above, a process could pass out a reference to a remote object and immediately release any remaining references to that remote object. When the poor client called back to translate the OXID, the string bindings would be gone. To deal with that case, the object exporter must keep the string binding/OXID translation for one full time-out period (round up if the release occurs in the middle of a ping period) after the last local reference is released. A time-out period is the number of pings times the ping period.

.2 IObjectExporter::SimplePing

```
[idempotent] error_status_t SimplePing
    (
    [in]  handle_t  hRpc,
    [in]  SETID    *pSetId
    );
```

Pings provide a mechanism to garbage collect interfaces. If an interface has references but is not being pinged, it may be released. Conversely, if an interface has no references, it may be released even though it has recently been pinged. SimplePing just pings the contents of a set. The set must be created with ComplexPing (see below).

Ping a set, previously created with IObjectExporter::ComplexPing, of OIDs owned by this Object Exporter. Note that neither IPIDs nor OIDs may be pinged, only explicitly created SETIDs.

Argument	Type	Description
hRpc	handle_t	An RPC binding handle used to make the request.
pSetId	SETID*	A SETID previously created with IObjectExporter::ComplexPing on this same Object Exporter.

Return Value	Meaning
S_OK	Success. The set was pinged.
RPC_E_INVALID_SET	This SETID is unknown to this Object Exporter, and thus the ping did not occur.
E_UNEXPECTED	An unspecified error occurred. It is not known whether the ping was done or not.

.3 IObjectExporter::ComplexPing

```
[idempotent] error_status_t ComplexPing
    (
    [in]  handle_t        hRpc,
    [in]  SETID          *pSetId,
    [in]  unsigned short  SequenceNum,
    [in]  unsigned short  SetPingPeriod,
    [in]  unsigned short  SetNumPingsToTimeout,
    [out] unsigned short *pReqSetPingPeriod,
    [out] unsigned short *pReqSetNumPingsToTimeout,
    [in]  unsigned short  cAddToSet,
    [in]  unsigned short  cDelFromSet,
    [in,  unique, size_is(cAddToSet)]   GUID AddToSet[],
    [in,  unique, size_is(cDelFromSet)] GUID DelFromSet[]
    );
```

Ping a ping set. Optionally, add and / or remove some OIDs from the set. Optionally, adjust the ping timing parameters associated with the set. After a set is defined, a SimplePing will mark the entire contents of the set as active. After a set is defined, SimplePing should be used to ping the set. ComplexPing need only be used to adjust the contents of the set (or the time-out).

Ping set ids (SETIDs) are allocated unilaterally by a client Object Exporter. The client Object Exporter then communicates with the server Object Exporter to add (and later remove) OIDs from the ping set. Clients must ensure the SETIDs pinged at a given server are unique over all of that server's clients. Thus, the client must only use SETIDs that it knows not to be in use as SETIDs by other clients on that server. (In practice, clients allocate SETIDs as globally unique). A client may use as many sets as it likes, though using fewer sets is more efficient.

Each OID owned by a server Object Exporter may be placed in zero or more ping sets by the various clients of the OID. The client owner of each such set will set a ping period and a ping time-out count for the set, thus determining an overall time-out period for the set as the product of these two values. The time-out period is implicitly applied to each OID contained in the set and to future OIDs that might add be added to it. The server Object Exporter is responsible for ensuring that an OID that it owns does not expire until at least a period of time *t* has elapsed without that OID being pinged, where *t* is the maximum time-out period over all the sets which presently contain the given OID, or, if OID is not presently in any such sets but was previously, *t* is the time-out period for the last set from which OID was removed at the instant that that removal was done;[120] otherwise, OID has never been in a set, and *t* is a default value (TBD).

Clients are responsible for pinging servers often enough to ensure that they do not expire given the possibility of network delays, lost packets, and so on. If a client only requires access to a given object for what it would consider less than a time-out period for the object (that is, it receives and release the object in that period of time), then unless it is certain it has not itself passed the object to another client it must be sure to nevertheless ping the object (a ComplexPing that both adds and removes the OID will suffice). This ensures that an object will not expire as it is passed through a chain of calls from one client to another.

An OID is said to be pinged when a set into which it was previously added and presently still resides is pinged with either a SimplePing or a ComplexPing, or when it is newly added to a set with ComplexPing. Note that these rules imply that a ComplexPing that removes an OID from a set still counts as a ping on that OID.

In addition to pinging the set SETID, this call sets the time-out period of the set as the product of a newly-specified ping period and a newly-specified "ping count to expiration;" these values take effect immediately. Ping periods are specified in tenths of a second, yielding a maximum allowable ping period of about 1 hr 50 min. Adjustment of the time-out period of the set is considered to happen before the addition of any new OIDs to the set, which is in turn considered to happen before the removal of any OIDs from the set. Thus, an OID that is added and removed in a single call no longer resides in the set, but is considered to have been pinged, and will have as its time-out at least the time-out period specified in that ComplexPing call.

On exit, the server may request that the client adjust the time-out period; that is, ask it to specify a different time-out period in subsequent calls to ComplexPing. This capability can be used to reduce traffic in busy servers or over slow links. The server indicates its desire through the values it returns through the variables pReqSetPingPeriod and pReqSetNumPingsToTimeOut. If the server seeks no change, it simply returns the corresponding values passed by the client; if it wishes a longer time-out period, it indicates larger values for one or both of these variables; if it wishes a smaller period, it indicates smaller values. When indicating a larger value, the server must start immediately acting on that larger value by adjusting the time-out period of the set. However, when indicating a smaller value, it must consider its request as purely advice to the client, and not take any action: if the client wishes to oblige, it will do so in a subsequent call to ComplexPing by specifying an appropriate time-out period.

[120] That is, adjusting the set's time-out period after the OID has been removed from it has no effect on the time-out of the OID.

Argument	Type	Description
hRpc	handle_t	An RPC binding handle used to make the request.
pSetId	SETID	The SETID being manipulated.
SequenceNum	USHORT	The sequence number allows the object exporter to detect duplicate packets. Since the call is idempotent, it is possible for duplicates to get executed and for calls to arrive out of order when one ping is delayed.
SetPingPeriod	USHORT	**This parameter is going away.**
SetNumPingsToTimeOut	USHORT	**This parameter is going away**.
pReqSetPingPeriod	USHORT*	**This parameter is changing**. The server uses pReqSetPingPeriod to request a new ping period. If the requested period is shorter then the current period, the server must continue to use the current period until the client calls back. When the client calls back the old requested period will only be used if the client specifies it as the new SetPingPeriod. If the requested period is longer then the current period, the server must immediately begin using the new period. However, if the client doesn't accept it, the next call will contain the old, shorter period.
pReqSetNumPingsToTimeOut	USHORT*	**This parameter is changing**. The server uses pReqSetNumPingsToTimeout to request a new number of pings. If the number of pings is less then the current number of pings, the server must continue to use the current number of pings until the client calls back. When the client calls back the old requested period will only be used if the client specifies it as the new SetNumPingsToTimeout. If the requested number of pings is larger then the current number of pings, the server must immediately being using the new number of pings. However, if the client doesn't accept it, the next call will contain the old, smaller number of pings.
cAddToSet	USHORT	The size of the array AddToSet.
cDelFromSet	USHORT	The size of the array DelFromSet.
AddToSet	OID[]	The list of OIDs which are to be added to this set. Adding an OID to a set in which it already exists is permitted; such an action, as would be expected, is considered to ping the OID.
DelFromSet	OID[]	The list of OIDs which are to be removed from this set. Removal counts as a ping. An IPID removed from a set will expire after the number of ping periods has expired without any pings (not the number of ping periods - 1). If an id is added and removed from a set in the same ComplexPing, the id is considered to have been deleted.

Return Value	Meaning
S_OK	Success. The set was pinged, etc.
RPC_E_INVALID_OBJECT	Indicates that some OID was not recognized. There is no recovery action for this error, it is informational only.
RPC_E_ACCESS_DENIED	
RPC_E_OUT_OF_ORDER	Returned when a call is received with a sequence number which is less then the last sequence number executed successfully.
E_OUTOFMEMORY	There was not enough memory to service the call. The caller may retry adding OIDs to the set on the next ping.
E_UNEXPECTED	An unspecified error occurred. It is not known whether the ping or any of the other actions were done or not.
Security related errors	TBD

5 Service Control Manager

The Service Control Manager (SCM) is the component of the COM Library responsible for locating class implementations and running them. The SCM ensures that when a client request is made, the appropriate server is connected and ready to receive the request. The SCM keeps a database of class information based on the system registry that the client caches locally through the COM library.

Machines in a COM environment which support the ability to instantiate objects on behalf of a remote client offer SCM services remotely via an ORPC interface.[121] To ensure connectivity, such SCM services

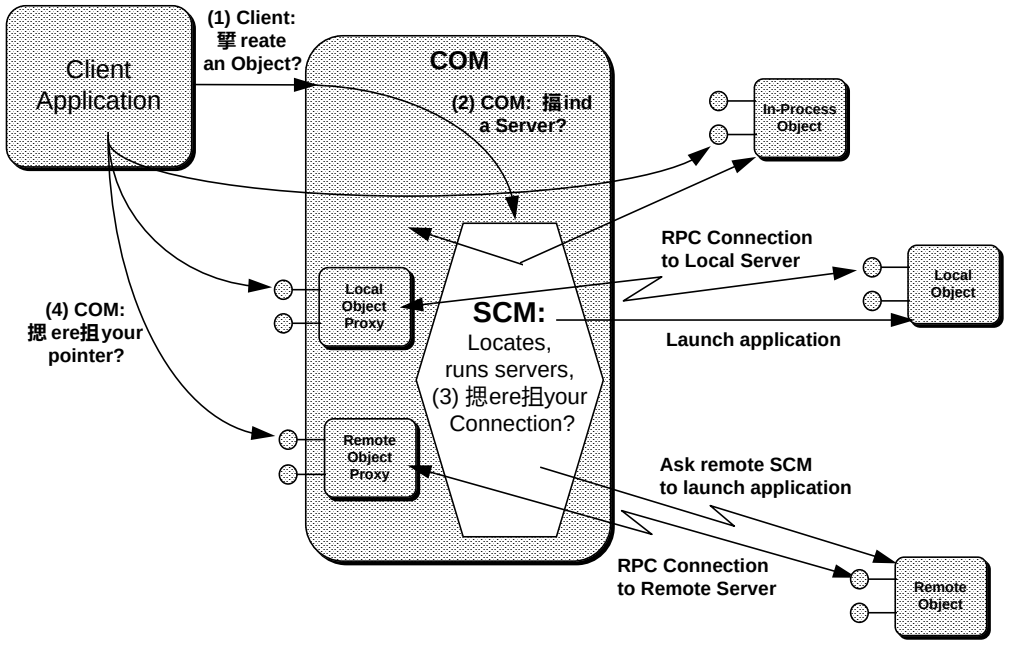


Figure 15-1: COM delegates responsibility of loading and launching servers to the SCM.

reside at the same well-known endpoints as the COM Object Exporter[122] on each machine. Note that unlike the Object Exporter service, which is required for a machine to expose COM objects remotely, the exposed SCM service is in fact *optional* and some machines may not offer it. Clients may receive references to existing objects on such a machine or cause objects to be instantiated on that machine through means besides the services offered by the SCM, such as a through a moniker binding mechanism.

These capabilities are the basis for COM's implementation locator services as outlined in Figure 15-1.

When a client makes a request to create an object of a CLSID, the COM Library contacts the local SCM (the one on the same machine) and requests that the appropriate server be located or launched, and a class factory returned to the COM Library. After that, the COM Library, or the client, can ask the class factory to create an object.

The actions taken by the local SCM depend on the type of object server that is registered for the CLSID:

In-Process	The SCM returns the file path of the DLL containing the object server implementation. The COM library then loads the DLL and asks it for its class factory interface pointer.
Local	The SCM starts the local executable which registers a class factory on startup. That pointer is then available to COM.
Remote	The local SCM contacts the SCM running on the appropriate remote machine and forwards the request to the remote SCM. The remote

[121] Not surprisingly, to get around the chicken and egg problem, getting to a remote machine's SCM interface is not done via normal CoCreateInstance means. Richer mechanisms for creating an interface to a remote SCM are TBD, but note that the clarity about the SCM-to-SCM interface and its endpoints ensure interconnectivity.

[122] This is true despite final word whether the Object Exporter will reuse the DCE RPC Endpoint Mapper's endpoints or a different well-known set TBD, as noted above.

SCM launches the server which registers a class factory like the local server with COM on that remote machine. The remote SCM then maintains a connection to that class factory and returns an RPC connection to the local SCM which corresponds to that remote class factory. The local SCM then returns that connection to COM which creates a class factory proxy which will internally forward requests to the remote SCM via the RPC connection and thus on to the remote server.

Note that if the remote SCM determines that the remote server is actually an in-process server, it launches a "surrogate" server that then loads that in-process server. The surrogate does nothing more than pass all requests on through to the loaded DLL.

ISCMSCM interface is defined as follows (SCMSCM.IDL):

```
//+-------------------------------------------------------------------------
//
//  Microsoft Windows
//  Copyright (C) Microsoft Corporation, 1992 - 1995.
//
//  File:      scmscm.idl
//
//  Synopsis:      Interface for SCM to SCM communication.
//
//  This is the interface that needs to be supported by hosts that allow
//  activation of objects. Only one instance of this interface can be exported
//  by the host.
//
//--------------------------------------------------------------------------
[
  uuid(00000137-0000-0000-C000-000000000046),
  version(1.0),
  pointer_default(unique)
]

interface ISCMtoSCM
{
    HRESULT ActivationRequest(
        [in] handle_t                               hRpc,
        [in] ORPCTHIS *                             orpcthis,
        [out] ORPCTHAT *                            orpcthat,
        [in] const GUID *                           rclsid,
        [in, string, unique] WCHAR *                pwszObjectName,
        [in] DWORD                                  clsctx,
        [in] DWORD                                  grfMode,
        [in] DWORD                                  dwCount,
        [in,unique,size_is(Interfaces)] IID *       pIIDs,
        [out,size_is(Interfaces)] OBJREF **         ppInterfaces,
        [out,size_is(Interfaces)] HRESULT *         pResults
        );
}
```

.1 ISCMtoSCM::ActivationRequest

HRESULT ISCMtoSCM::ActivationRequest(hRpc, orpcthis, orpcthat, rclsid, pwszObjectName, clsctx, grfMode, dwCount, pIIDs, ppInterfaces, pResults);

This single method encapsulates several different forms of activating and instantiating objects on the machine of this SCM.

When pIIDs is NULL, this function behaves roughly as CoGetClassObject(rclsid, clsctx, NULL, IID_IClassFactory,?), returning a class-object to the caller.

When pwszObjectName and pObjectStorage are NULL, this function behaves roughly as CoCreateInstanceEx(rclsid, NULL, clsctx, ?).

Otherwise, this function acts roughly similar to CreateFileMoniker(pwszObjectName, ?) followed by IMoniker::BindToObject(?) to retrieve the required interfaces.

Argument	Type	Description
hRpc	handle_t	An RPC binding handle used to make the request.
orpcthis	ORPCTHIS*	ORPCTHIS identifying this object.
orpcthat	ORPCTHAT*	ORPCTHAT holding return values.
rclsid	CLSID*	Identifies the class to be run to service the request.
pwszObjectName	WCHAR*	Identifies the persistent representation of the object to this machine. Typically, this is a file name which is used to determine the class, as in CreateFileMoniker(pwszFileName, &pmk) followed by BindMoniker(pmk?).
clsctx	DWORD	Values taken from the CLSCTX enumeration.
grfMode	DWORD	Values taken from the STGM enumeration.
dwCount	DWORD	The number of interfaces to return.
pIIDs	IID*	An array of interfaces to QueryInterface for on the new object.
ppInterfaces	OBJREF**	Location to return an array of interfaces on the object.
pResults	HRESULT*	Location to return an array of return codes about the successful retrieval of each of the dwCount interfaces.

Return Value	Meaning
S_OK	Success.
CO_S_NOTALLINTERFACES	Some but not all of the dwCount interfaces were returned in ppInterfaceData. Examine pResults to identify exactly which interf

6 Wrapping DCE RPC calls to interoperate with ORPC

This is an example of a server side wrapper for the Bar method. It assumes the existence of small helper functions to import and export object references and lookup previously exported object references.

```
RPC_STATUS Bar(handle_t h, short i, OBJREF * prIB, OBJREF ** pprIW) {
        UUID ipid;
        RPC_STATUS status;
        IFoo * pIF;
        IBar * pIB;
        IWaz * PIW;
        HRESULT hR;

        status = RpcBindingInqObject(h, &ipid);
        if (status) return(SOMETHING);

        status = ORpcLookupIPID(ipid, &pIF);
        if (status) return(SOMETHING);

        status = ORpcImportObjRef(prIB, &pIB);
        if (status) return(SOMETHING);

        hR = pIF->Bar(i, pIB, &pIW);            // actual call to the method!

        pIB->Release();
        status = ORpcExportObjRef(pIW, pprIW);
        return(hR ? hR : SOMETHING);
        };
```

This is an example of the client side wrapper for Bar:

```
// assume some class CFoo that implements Bar method
class CFoo : IUnknown, IFoo {
        UUID ipid;          // one for each interface?
        handle_t h;

        virtual HRESULT QueryInterface(UUID iid, void **ppvoid);
        virtual HRESULT AddRef();
        virtual HRESULT Release();
        virtual HRESULT Bar(short i, IFoo * pIF, IWaz ** ppIW);
        };
```

```
HRESULT CFoo::Bar(short i, IFoo * pIF, IWaz ** ppIW) {
                OBJREF * prIF;
                OBJREF * prIW;
                HRESULT hR;
                RPC_STATUS status;

                status = RpcBindingSetObject(this->h, this->ipid);
                if (status) return(SOMETHING);

                status = ORpcExportObjRef(pIF, &prIF);
                if (status) return(SOMETHING);

                hR = Bar(this->h, i, prIF, &prIW);

                status = ORpcImportObjRef(prIW, ppIW);

                ORpcFreeObjRef(prIF);
                ORpcFreeObjRef(prIW);

                return(hR ? hR : SOMETHING);
        };
```

7 Implementing ORPC in RPC

Since the implicit parameters are specified as IDL, the ORPC header received by RPC will contain many fields inserted by MIDL. Here are the definitions for the header on the wire.

```
/*
    An inbound header would be laid out as follows where the
extent array is optional and there may be zero of more extents.
An outbound header is laid out similarly.

    ORPCTHIS_WITHNOEXTENSIONS
        [
          ORPC_EXTENT_ARRAY
        [ORPC_EXTENT]*
        ]

*/
typedef struct
{
  COMVERSION       version;          // COM version number
  unsigned long   flags;             // INFO flags for presence of other data
  unsigned long   reserved1;         // set to zero
  LTID             ltid;             // logical thread id of caller
  unsigned long   unique;            // tag to indicate presence of extensions
} ORPCTHIS_WITHNOEXTENSION;

typedef struct
{
    unsigned long   rounded_size;    // Actual number of extents.
    uuid_t           id;               // Extension identifier.
    unsigned long   size;              // Extension size.
    byte             data[];           // Extension data.
} ORPC_EXTENT;


// Array of extensions.
typedef struct
{
  unsigned long   rounded_size;    // Actual number of extents
  unsigned long   size;              // Number of extents
  unsigned long   unique_flag[];   // Flags to indicate presense of ORPC_EXTENTs
} ORPC_EXTENT_ARRAY;

typedef struct
{
  unsigned long   flags;             // INFO flags for presence of other data
  unsigned long   unique;            // tag to indicate presence of extensions
} ORPCTHAT_WITH_NOEXTENSIONS;
```


This page intentionally left blank.

Appendix B: Bibliography

[CAE RPC] *CAE Specification, X/Open DCE: Remote Procedure Call*, X/Open Company Limited, Reading, Berkshire, UK (xopen.co.uk), 1994. X/Open Document Number C309. ISBN 1-85912-041-5.

This page intentionally left blank.

Appendix C: Specification Revision History

06 March 1995	First major draft.
24 October 1995	Second major draft.

This page intentionally left blank.

Appendix D: Index

