DCOM Architecture

Markus Horstmann and Mary Kirtland

July 23, 1997

Contents

Introduction

The Microsoft® Distributed Component Object Model (DCOM) extends the Component Object Model (COM) to support communication among objects on different computers—on a local area network (LAN), a wide area network (WAN), or even the Internet. With DCOM, your application can be distributed at locations that make the most sense to your customer and to the application.

Because DCOM is a seamless evolution of COM, the world's leading component technology, you can take advantage of your existing investment in COM-based applications, components, tools, and knowledge to move into the world of standards-based distributed computing. As you do so, DCOM handles low-level details of network protocols so you can focus on your real business: providing great solutions to your customers.

Why Should I Read This Article?

This article focuses on the inner workings of DCOM—the Transport Control Protocol/Internet Protocol (TCP/IP) of objects. It targets the application developer who wishes to create "state of the art" applications, which scale equally well on the intranet, on the Internet, and beyond. In some areas this paper assumes that you are familiar with the basic concepts of the COM, although some of these concepts are revisited from the perspective of distributed application development.

How Should I Read This Article?

This white paper is part of a series introducing COM and DCOM material. The section "References," at the end of this paper indicates where you can get other papers in the series.

If you are not sure what DCOM is all about, the white paper, "DCOM Technical Overview," will give you a high-level overview of the kinds of problems DCOM helps solve. It also contains references to the necessary COM basics, so you can take full advantage of the technical details in this DCOM Architecture paper. There is a good chance that you will be able to apply many of the ideas in these solutions to your own work. Read this paper from beginning to end, or use it as a reference and pick the areas that are most interesting to you.

Where Can I Get DCOM?

DCOM currently ships with the Microsoft Windows NT® 4.0 operating system. DCOM for the Microsoft Windows® 95 operating system is available for download on the Microsoft COM Web site (http://www.microsoft.com/com/), and will ship with Microsoft Internet Explorer 4.0 and the next version of Windows 95.

DCOM implementations on all major UNIX platforms are available from Software AG (http://www.sagus.com). At publication time, beta versions for Solaris, Linux, and HP/UX were available for download.

The COM and DCOM specifications are managed by the Active Group, a consortium of vendors interested in the future evolution of COM and DCOM. Members of the Active Group are actively using COM and DCOM in their applications and as their infrastructure. A reference implementation of COM and DCOM in source-code form will be available for licensing through The Open Group in 1997.

DCOM Architecture

We'll now take you on a guided tour through the inner and outer workings of DCOM to show you how DCOM realizes the promise of easy distributed computing, without compromising flexibility, scalability, and robustness.

DCOM sits right in the middle of the components of your application; it provides the invisible glue that ties things together. Figure 1 shows how it all fits together.

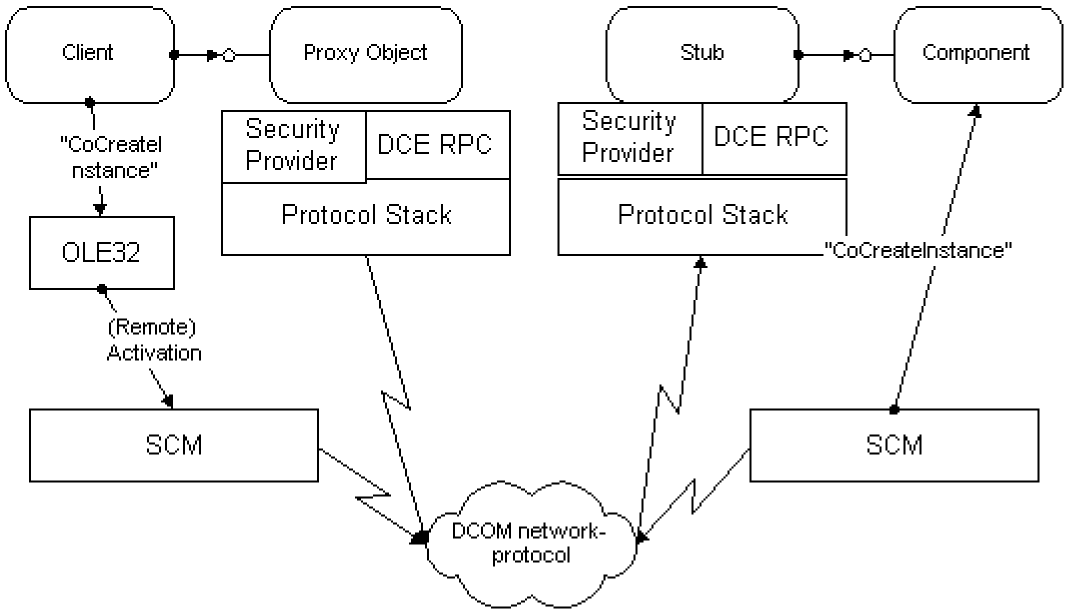


Figure 1

Return to Contents

Locating Objects: Activation

"In the beginning, there was . . . a lonely, solitary client component looking for something to do. It wanted to connect to some other, smarter and stronger components that would help it out of its isolated existence on the overloaded client machine."

One of the central pieces of COM is a mechanism for establishing connections to components and creating new instances of components. These mechanisms are commonly referred to as activation mechanisms.

Creating Objects, Local or Remote

One of the most basic requirements of a distributed system is the ability to create components. In the COM world, object classes are named with globally unique identifiers (GUIDs). When GUIDs are used to refer to particular classes of objects, they are called Class IDs. These Class IDs are nothing more than fairly large integers (128 bits) that provide a collision free, decentralized namespace for object classes. If a COM programmer wants to create a new object, she calls one of several functions in the COM libraries, as displayed in Table 1.

Table 1

CoCreateInstance(Ex) (*<CLSID>Ã,Â…*)	Creates an interface pointer to an uninitialized instance of the object class *<CLSID>*.
CoGetInstanceFromFile	Creates a new instance and initializes it from a file.
CoGetInstanceFromStorage	Creates a new instance and initializes it from a storage.
CoGetClassObject (*<CLSID>Ã,Â…)*	Returns an interface pointer to a "class factory object" that can be used to create one or more uninitialized instances of the object class *<CLSID>*.
CoGetClassObjectFromURL	Returns an interface pointer to a class factory object for a given class. If no class is specified, this function will choose the appropriate class for a specified Multipurpose Internet Mail Extension (MIME) type. If the desired object is installed on the system, it is instantiated. Otherwise, the necessary code is

	downloaded and installed from a specified Universal Resource Locator (URL).

The COM libraries look up the appropriate binary (dynamic-link library or executable) in the system registry, create the object, and return an interface pointer to the caller.

For DCOM, the object creation mechanism in the COM libraries is enhanced to allow object creation on other machines. In order to be able to create a remote object, the COM libraries need to know the network name of the server. Once the server name and the Class Identifier (CLSID) are known, a portion of the COM libraries called the service control manager (SCM) on the client machine connects to the SCM on the server machine and requests creation of this object.

DCOM provides two fundamental mechanisms that allow clients to indicate the remote server name when an object is created (a third mechanism involves monikers and is described below):

1. As a fixed configuration in the system registry or in the DCOM Class Store (see below for details)
2. As an explicit parameter to **CoCreateInstanceEx**, **CoGetInstanceFromFile**, **CoGetInstanceFromStorage**, or **CoGetClassObject**

External RemoteServerName configuration

The first mechanism is extremely useful for maintaining location transparency: clients should not know whether a component is running locally or remotely. By making the remote server name part of the server component's configuration information on the client machine, clients do not have to worry about maintaining or obtaining the server location. All a client ever needs to know is the CLSID of the component. It simply calls **CoCreateInstance** (or **CreateObject** in Microsoft Visual Basic® or "new" in Java), and the COM libraries transparently create the correct component on the preconfigured server. Even existing COM clients that were designed before the availability of DCOM can transparently use remote components using this mechanism.

Note that a server machine cannot forward creation requests to yet another machine using **RemoteServerName**. If machine *X* uses **RemoteServerName** to indicate that objects of CLSID C should be created on machine *Y*, and if machine *Y* has a RemoteServerName specified for CLSID C pointing to machine *Z*, objects requested by machine *X* will still only be created on machine *Y*. See "Referrals" in the section "Connection Management," for more information.

For many applications, having a single, externally configured server name for each component is sufficient. It keeps the client's code free from managing this configuration data: if the server name changes, the registry (or the class store) is changed and the application continues to work without further action.

The remote server name is stored in the system registry under a new key in HKEY_CLASSES_ROOT (HKCR):

```
[HKEY_CLASSES_ROOT\APPID\{<appid-guid>}]
    "RemoteServerName"="<DNS name>"
```

The Class ID entry for the component in turn has a new named value that points to the Application ID (AppID):

```
[HKEY_CLASSES_ROOT\CLSID\{<clsid-guid>}]
    "AppId"="<appid-guid>"
```

The AppID concept was introduced as part of the security support in COM and is fully described in the security section below. The AppID essentially represents a process that is shared by multiple CLSIDs. All objects in this process share the same default security settings.

The APPID concept can be used to avoid redundant registry keys that all contain the same server name. CLSIDs that are known to always run on the same server machine (typically because they are implemented in the same executable or DLL) can all point to the same AppID key and thus all share the same **RemoteServerName** registry key.

Example: Auction service

An auction system is implemented using a central DCOM component, to which all auction participants and the auction administrator connect. Participants place bids, which are distributed in real time through Connection Points (see "Connection Points" in the section "Connection Management") to all connected clients.

In this application, clients do not care which specific server machine is connected. They simply need to connect to the one-and-only auction server machine. (Note: Load balancing and fault tolerance might require that the client actually be connected to different physical servers; however, conceptually the client does not care which of the servers it connects to. Several techniques are presented in the following example, which provide load balancing and fault tolerance in a completely transparent fashion.) The Domain Name Service (DNS) name of the auction server is written into the client's registry and is automatically used by the COM libraries when asked for an instance of class "Auction Server" (that is, CLSID_AuctionServer).

The following illustrates what the entries in the client's registry could look like:

```
REGEDIT4
[HKEY_CLASSES_ROOT\CLSID\{<CLSID_AuctionServer>}]
    "AppID"="{<APPID_AuctionServer>}"
[HKEY_CLASSES_ROOT \APPID\{<APPID_AuctionServer>}]
    "RemoteServerName"="auctions.r.us.com"
; Note: APPID_AuctionServer can have the same value as CLSID_AuctionServer.
```

An Auction client could then execute the following code:

```
IAuction* pAuction = NULL;

HRESULT hr=CoCreateInstance(

                CLSID_AuctionService,    // Request an instance of class CLSID_AuctionService

                NULL,                     // No aggregation.

                CLSCTX_SERVER,         // Any server is fine.

                IID_IAuction,             // Ask for an interface of type IID_IAuction

                (void**) &pAuction);    // Pointer to returned interface pointer.
if (SUCCEEDED(hr))
{
    pAuction->PlaceBid(1324, 100000.00);    // Bid US$ 100,000.00 on item # 1324.
    pAuction->Release();                    // Release this AuctionService instance.
}
```

or in Visual Basic:

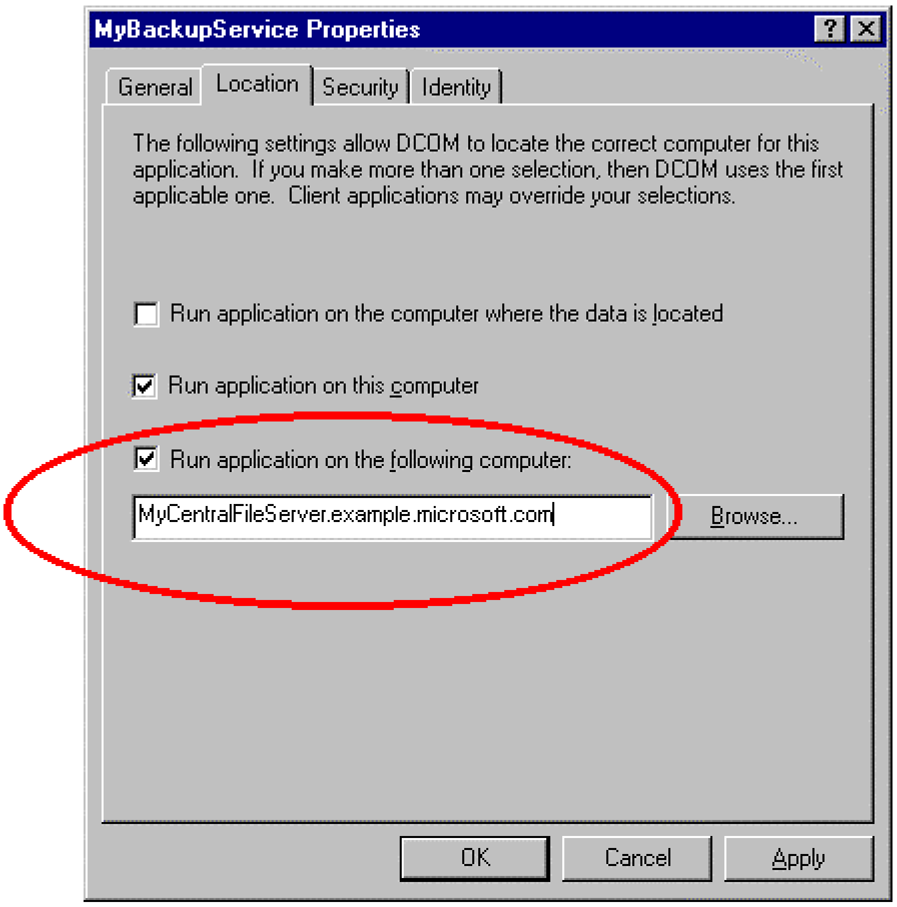
```
Dim Auction as IAuction

Set Auction = CreateObject("AuctionService") 'Uses ProgID on client machine.
Auction.PlaceBid(1324, 100000.00)
REM Auction object gets released when the variable goes out of scope
REM or is explicitly set to "Nothing":
set Auction = Nothing
```

or in Java:

```
IAuction Auction = new AuctionService; // Uses generated AuctionService.class file.
Auction.PlaceBid(1324, 100000.00);
// Auction object gets garbage collected.
```

The DCOM configuration tool (DCOMCNFG.EXE, which ships as part of Windows NT and DCOM for Windows 95) allows configuration of remote server names without the need to modify the registry directly, as seen in Figure 2.



The following settings allow DCOM to locate the correct computer for this application. If you make more than one selection, then DCOM uses the first applicable one. Client applications may override your selections.

- [ ] Run application on the computer where the data is located
- [✓] Run application on this computer
- [✓] Run application on the following computer:

MyCentralFileServer.example.microsoft.com

Figure 2

With the Windows 2000 operating system, COM will add the concept of a central store for COM classes. All activation-related information about a component will optionally be stored in the Active Directory on the domain controller, just as the logon and authentication service stores user credentials on the domain controller. The COM libraries will transparently retrieve activation information—including remote server name configurations—from the Active Directory. Changing the components' configuration information in the Active Directory will automatically propagate to all clients connected to this portion of the Active Directory.

Programmatic control over RemoteServerName

Some applications require explicit run-time control over the server a client connects to. Examples include chat applications, multiplayer games, and administrative tools that need to perform remote administration on specific machines.

For this kind of application, COM allows the remote server name to be explicitly specified as a parameter to **CoCreateInstanceEx**, **CoGetInstanceFromFile**, **CoGetInstanceFromStorage**, or **CoGetClassObject**. The developer of the client code is in complete control of the server name being used by COM for remote activation.

Example: Remote administration

A backup service is implemented as a COM object with CLSID_MyBackupService. The administrative front end is a client of this COM object and can be used to configure the backup service and trigger immediate backups or restores. Using DCOM, the backup service can easily be administered remotely:

```
HRESULT hr=S_OK;
// Will request two interfaces on the backup service object.

MULTIQI mqi[]={

    {IID_IBackupAdmin, NULL , hr1},

    {IID_IBackupConfig, NULL, hr2} };
```

```
// Will connect to server "MyCentralFileServer.example.microsoft.com"

COSERVERINFO srvinfo = {0, L"MyCentralFileServer.example.microsoft.com", NULL, 0};

// Create the object and query for two interfaces.

HRESULT hr=CoCreateInstanceEx(

    CLSID_MyBackupService,            // Request an instance of class CLSID_MyBackupService.

    NULL,                            // No aggregation.

    CLSCTX_SERVER,                    // Any server is fine.

    &srvinfo,                        // Contains remote server name.

    sizeof(mqi)/sizeof(mqi[0]),    // number of interfaces we are requesting (2)

    &mqi);                            // structure indicating IIDs and interface pointers
if (SUCCEEDED(hr))
{
    if (SUCCEEDED(mqi[0].hr))
    {
        IBackupAdmin* pBackupAdmin=mqi[0].pItf;    // Retrieve first interface pointer.
        hr=pBackupAdmin->StartBackup();            // use itÃ‚…
        pBackupAdmin->Release();                    // Release it.
    }
    if (SUCCEEDED(mqi[1].hr))
    {
        LPWSTR pStatus=NULL;
        IBackupConfig* pBackupConfig=mqi[1].pItf;    // Retrieve second interface pointer.
        hr=pBackupConfig->GetCurrentBackupStatus(&pStatus);    // use itÃ‚…
        pBackupConfig->Release();                    // Release it.
    }
}
```

Running in-process components remotely: Surrogate

In order to run in-process components remotely, a *surrogate process* on the remote machine is required. In addition to enabling remote execution, surrogate processes offer the following benefits:

- Faults in the in-process server are isolated to the surrogate process.
- One surrogate process can service multiple clients simultaneously.
- Clients can protect themselves from untrusted server code, while accessing the services the server provides.
- Running an in-process server in a surrogate gives the DLL the surrogate's security.

Windows NT 4.0 Service Pack 2.0 and DCOM for Windows 95 introduced a default surrogate process, as well as a protocol for writing custom surrogates. The default implementation of the surrogate process is a mixed-threading, model-style, pseudo-COM server. When multiple DLL servers are loaded into a single surrogate process, this process ensures that each DLL server is instantiated using the threading model specified in the registry for that server. If a DLL server supports both threading models, then COM will choose multithreading. This surrogate process is written so that COM handles both the unloading of DLL servers and the termination of the surrogate process.

An in-process server will be loaded into a surrogate process under the following conditions:

1. There must be an AppID value specified under the CLSID key in the registry, and a corresponding AppID key.
2. In an activation call, the CLSCTX_LOCAL_SERVER bit is set, and the CLSID key does not specify LocalServer32, LocalServer, or LocalService.
3. The CLSID key contains the InProcServer32 subkey.
4. The DLL specified in the InProcServer32 key exists.
5. The DllSurrogate value exists under the AppID key.

If there is a LocalServer or LocalServer32 or LocalService, indicating the existence of an EXE, the EXE server or service will always be launched in preference to loading a DLL server into a surrogate process.

The DllSurrogate named-value must be specified for surrogate activation to occur. To launch an instance of the system-supplied surrogate, set the value of DllSurrogate either to an empty string or to NULL. To specify the launch of a custom surrogate, set the value to the path of the surrogate.

For remote surrogate activation, specify **RemoteServerName** but not DllSurrogate on the client, and specify DllSurrogate on the server.

It is best to configure a DLL server that is designed to run alone in its own surrogate process and to service multiple clients across a network with a RunAs value specified under the AppID registry key. Whether the RunAs specifies "Interactive User" or a specific user identity depends upon the user interface (UI), security, and other server requirements. Specifying a RunAs value ensures that only one instance of the server is loaded to service all of the clients, regardless of the identity of the client. On the other hand, do not configure the server with RunAs If the intention is to have one instance of the DLL server running in surrogate to service each remote client identity.

Multiple servers will share a surrogate if they are launched under matching security identities and share the same AppID value. If two servers are launched under different security identities, they must be in different surrogates, regardless of whether their AppIDs match.

Connecting to Specific Object Instances

Often object instances are not replaceable or interchangeable; once certain methods have been called, the object has a valuable state that differentiates it from other instances of the same class. Clients need to be able to recognize this and reconnect to a specific object instance easily.

Example: Database engines

A database engine implemented as a COM object serves fundamentally different purposes if it is initialized to access a general ledger database of a huge corporation than if it is initialized to access the database containing your favorite CD titles. Although the code (CLSID) might be the same for both (assuming wide scalability of the database engine), the real distinguishing characteristic of an instance of this database engine is the actual data it accesses.

Example: HTML Editor

A component that displays and modifies Hypertext Markup Language (HTML) is implemented as a COM object. In addition to its CLSID, the concrete behavior of the component as it is used to manipulate elements in the HTML page is defined by the actual HTML data.

Naming object instances: Monikers

COM defines a naming mechanism for object instances. As the examples above show, object instances can be characterized by quite different identifiers (a database name plus a database server name, a URL to an HTML page, and so on). COM's instance-naming mechanism is designed to be extremely flexible and extensible. To achieve this flexibility, instance names (commonly called "monikers") for COM objects are themselves objects. These naming objects contain the logic necessary to find a currently running instance of the object that they are naming. They also can recreate and initialize an object instance in case there is no running instance. COM defines a standard interface to these naming objects, **IMoniker**, which can be used to bind to a named-object instance. Whatever the moniker object needs to do to connect to the actual object instance is hidden from the client.

Typically, object instances are in charge of naming themselves; they create a moniker object and hand it out to any client that is interested in reconnecting to them at a later time. Object instances also register their moniker object with the COM libraries. The system maintains a table of currently running, explicitly named, object instances called the Running Object Table (ROT).

Moniker objects use the ROT to quickly find a running-object instance. When a moniker object is asked to bind to the object instance it names, it looks in the ROT for a moniker object containing the same name. If it finds a match, it simply returns the pointer to the active object instance that is stored in the ROT. If the object instance is not running (or not registered), the moniker uses the object creation mechanisms described above to create an uninitialized instance. The moniker then restores the object's state through whatever mechanism is appropriate for the naming scheme, the moniker, and the object implement.

Most moniker objects can provide a human-readable representation of the name they are encapsulating. The typical human-readable name of a moniker conforms to fundamental URL syntax as defined by the Internet Engineering Task Force (IETF): the type of the moniker (the moniker's programmatic identifier [ProgID]) corresponds to the protocol prefix of a URL and the remainder of the URL is defined entirely by the moniker object.

Example: Sales information

Sales information for product X for Central Region for Q4 1996 in the NORTHAMERICA division might be stored on the \\dtwmkt server, in a specific table and a specific range of cells. An application needs to perform some typical operations on this data: "*How do sales of product X compare to sales of product Y this quarter?*" "*How do sales of product X compare between Q1 this year and Q1 last year?*" "*Add this figure with 10% of this figure, and update the table for next year's sales quota that resides on* www.example.microsoft.com/mkt/productsx/budget.xyz".

The following code illustrates how this last example could be encapsulated using monikers and a COM object that implements the necessary business logic:

```
HRESULT hr=S_OK;
IBindCtx* pBC=NULL;
hr=CreateBindCtx(NULL, &pBC);
if (SUCCEEDED(hr))
{
    DWORD dwEaten;
    IMoniker* pMoniker=NULL;

    // Create the moniker object.

    hr=MkParseDisplayName(pBC, L"file:\\\\dtwmkt\\mkt\\productx1\\salesQ496.xyz!Summary", &dwEat

                    &pMoniker);

    if (SUCCEEDED(hr))
    {

        // Connect to the actual business object, create and initialize it if neccessary.

        hr=pMoniker->BindToObject(pBC, NULL, IID_ISalesInfo, &pSales);

        if (SUCCEEDED(hr))
        {
            // Perform the operation.
            pSales->Add( 1.1, "http://www.example.microsoft.com/mkt/productx/budget.xyz")
            pSales->Release();
        }
        pMoniker->Release();
    }
    pBC->Release();
}
```

or in Visual Basic:

```
Dim Sales As Object
Set Sales = GetObject("file:\\dtwmkt\mkt\productx1\salesQ496.xyz!Summary")
Sales.Add(1.1, "http://www.example.microsoft.com/mkt/productx/budget.xyz")
```

"At Storage" activation

Monikers contain some reference to an object's persisted state, which is stored in "some form" (file, database) on "some server." DCOM uses this "location information" to provide yet another transparent object location mechanism: objects can indicate to a moniker that they wish to be activated where this persisted state has been stored. When a client binds to the moniker, it transparently creates a remote object and returns it to the client.

Running an object on the server where its data resides rather than on the client's local workstation offers a number of potential benefits. If the object is accessing large amounts of data, the network traffic and latency incurred could create a performance bottleneck. In the case of objects that are designed for multiple client use, it makes sense to run them where they can be accessed by a number of clients simultaneously across the network, rather than limit their use to a single workstation.

A COM object can be configured to run on the server where its data is located using the **ActivateAtStorage** value of its AppID registry key:

```
REGEDIT4
[HKEY_CLASSES_ROOT\APPID\{<appid-guid>}]
    "ActivateAtStorage" = "Y"
```

If the CLSID is not registered on the client machine, the moniker will try to "activate at storage" before it fails the call.

Example: Sales information

In the previous example, the component that implements the ISales interface would automatically run on the example "dtwmkt" server machine. DCOM would be used to perform the single invocation of **ISales::Add** and all processing would happen locally on the "dtwmkt" server machine.

As of this writing, the only moniker that supports activation at storage is the file moniker implemented by COM.

Overriding the server name: BIND_OPTS2

It is possible to programmatically override the remote server name that should be used in a moniker bind operation. The BIND_OPTS2 structure used in **IBindCtx::GetBindOptions** and **IBindCtx::SetBindOptions** has been extended to include new parameters. One of these new parameters is an optional pointer to the same **COSERVERINFO** structure that is used by **CoCreateInstanceEx**.

The following code illustrates the use of BIND_OPTS2:

```
// Will connect to server "MyCentralFileServer.example.microsoft.com".

COSERVERINFO srvinfo = {0, L"MyCentralFileServer.example.microsoft.com", NULL, 0};

HRESULT hr=S_OK;
IBindCtx* pBC=NULL;
hr=CreateBindCtx(NULL, &pBC);
if (SUCCEEDED(hr))
{

    BIND_OPTS2 bind;

    bind.cbStruct=sizeof(bind);

    pBC->GetBindOptions(&bind);
    CoTaskMemFree(bind.pServerInfo); // Free memory in case it's non NULL.
    bind.pServerInfo=&srvinfo;
    pBC->SetBindOptions(&bind);
    // Use the bind context.

    pBC->Release();
}
```

File moniker—*file:*

The file moniker is a moniker implemented as part of COM. It encapsulates the *file:* protocol which—in terms of COM—consists of the following elements:

- The moniker ProgID: *file:*. This maps the COM file moniker to the URL *file:* protocol. Typical display names for file monikers are "file:c:\my documents\July Report.doc" or "file:\\dtwoffice\public\productsX.doc".
- The protocol used for accessing the persisted data is the native-file access mechanism. On the Windows platform it is the Microsoft Win32® file API.
- The initialization protocol between the moniker and an object is **IPersistFile:** as the moniker creates an object it calls **QueryInterface** for **IPersistFile** and passes the file name to the object. The object in turn uses native file APIs to access the file.
- The CLSID detection mechanism is the **GetClassFile** API: If a file is a storage file, it returns the CLSID that was written with the **IStorage::SetClass** method. If the file is not a storage file, it attempts to match

various patterns in a file against entries under the HKEY_CLASSES_ROOT\FileType registry key. If this fails, **GetClassFile** uses the file extension to look up a ProgID associated with this file extension. Figure 3 illustrates this.

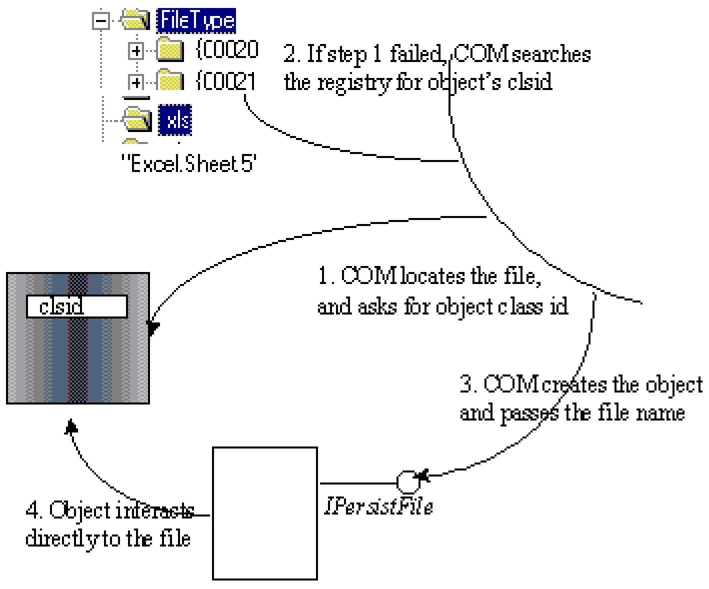

FileType
{C0020
{C0021
xls
"Excel.Sheet5"
2. If step 1 failed, COM searches the registry for object's clsid
1. COM locates the file, and asks for object class id
clsid
3. COM creates the object and passes the file name
IPersistFile
4. Object interacts directly to the file

Figure 3

Like any other moniker, file monikers can be composed to the "left" of item monikers. File monikers can also be composed to the "right" of class monikers or any other moniker that resolves to **IClassFactory** or **IClassActivator**. This can be used to override the class detection that is normally performed by calling **GetClassFile**.

URL moniker—*http:*, *ftp:*, *gopher:*, and similar

The URL moniker encapsulates several standard Internet protocols, including *http: https:*, *ftp:*, and *gopher:*. It is used by Microsoft Internet Explorer to bind to objects that encapsulate common Internet data types like .gif or .html files. In terms of COM, URL monikers can be described as follows:

- The moniker ProgID: *http:*, *ftp:*, and so on. These map the moniker object to the corresponding standard URL protocols. Typical display names for URL monikers are "http://www.example.microsoft.com" or "http://www.msn.com".
- The protocol used to access the persisted data depends on the Internet protocol being handled. URL monikers on the Windows platforms use the **WinINet** APIs.
- The initialization protocol between the moniker and an object supports multiple initialization interfaces. The first interface the URL moniker requests is **IPersistMoniker**, which essentially hands a pointer to the moniker to the object, which then can use **IMoniker::BindToStorage** (or any other binding mechanism it might know about) to get to the actual data. If the object does not implement **IPersistMoniker**, URL moniker falls back to other initialization interfaces, including **IPersistStream**, **IPersistStorage**, **IPersistPropertyBag**, **IPersistMemory** and **IPersistFile**.
- The CLSID detection mechanism is similar to the **GetClassFile** mechanism, but extended to recognize MIME types and other Internet-specific type detection mechanisms.

With URL monikers, COM clients can bind to objects whose persistent state is stored on the Internet without ever having to worry which Internet protocol is being used. In the "Sales Information" example presented previously, it is sufficient to replace the string "file:\\dtwmkt\mkt\productx1\salesQ496.xyz" with, for instance, "http://www.dtwmkt.com/mkt/public/productx1/salesq496.xyz".

URL monikers also support asynchronous binding, where the call to **IMoniker::BindToObject** returns instantly and progress indications are passed to the caller using an **IBindStatusCallBack** interface passed in the bind context. Please refer to the Microsoft Internet Client Software Development Kit (SDK) documentation (MSDNÃ‚Â™ Library, SDK Documentation) for more details on asynchronous moniker binding.

When binding to a URL moniker, COM picks the right transport protocol object. For example, if the user passes "http://www.example.microsoft.com/products.xls!summary!R2C1", COM will use the Hypertext Transfer Protocol (HTTP) object and download the "products.xls" file, search for an object to associate with this file, create the object, and pass it the "summary!R2C1" item moniker.

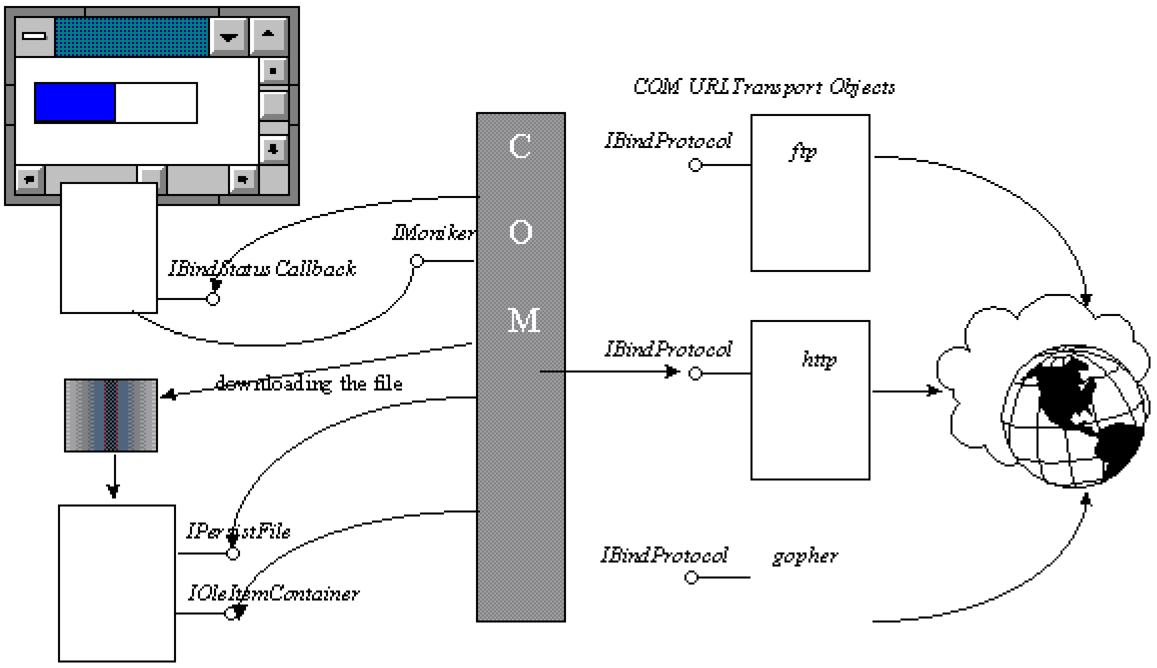


Figure 4

The URL moniker that is used by Microsoft Internet Explorer 3.0 does not support activation at storage.

Class moniker—*clsid:*

The class moniker is primarily used in conjunction with other monikers, like file monikers, to override the CLSID lookup mechanism in the other moniker.

The display name for class monikers is of the form:

```
display-name = "CLSID:" string-clsid-no-curly-braces *[";" clsid-options] ":"
clsid-options = clsid-param "=" value
clsid-param = none currently defined
```

You can programmatically compose the class moniker with file monikers. This is especially useful if you want to programmatically associate an object with a resource. For example, a file with extension .doc is normally associated with the ProgID "Word.Document". If you have an object that can read the Microsoft Word document, you can tell the moniker to use your object instead of Word objects when the binding process occurs, without having to change the registration for the .doc file extension.

The first step is to create a class moniker using **CreateClassMoniker** or **MkParseDisplayName**. The class moniker takes your object's Class ID or an intermediate object that supports **IClassActivator**. By implementing this intermediate object, you have full flexibility with how and where the object will be created. The **IClassActivator** only has one method: **GetClassObject**. If you pass your object's Class ID directly, COM will look at the registry entry for the object creation. Once you get the class moniker interface, you will create a file moniker as you normally do. In the **BindToObject** operation, you pass the class moniker as the *pmkLeft* parameter. This effectively tells COM to create the object with the current moniker—in this case, a file moniker.

Pseudo C++ example:

```
ProgIDFromCLSID( &clsid, "xyz.activator.1")
CreateClassMoniker( clsid, &pmkClass )
MkParseDisplayName( pcb, "\\northamerica\central\employee.doc", &dwEaten, pmkFile )
pmkFile->BindToObject( pcb, pmkClass, IID_IDispatch, &pDisp )
```

Return to Contents

Packaging Parameters and Objects: Marshaling

Although marshaling has been part of COM since its inception in 1993, we'll discuss marshaling and the mechanism to extend and customize marshaling in the context of distributed applications. These mechanisms enable sophisticated optimizations of the actual remote method invocations without changing the client's view of the object.

Remote Method Calls: Marshaling and Unmarshaling

When a client wants to call an object in another address space, the parameters to the method call must somehow be passed from the client's process to the object's process. The client places the parameters on the stack. (Actually, some parameters are passed directly in CPU registers, but for the purpose of this discussion, there is no fundamental difference between parameters on the stack or in CPU registers. To simplify the presentation, we merely refer to the "stack" as the mechanism through which parameters are passed.) In the case of a direct object invocation, the object reads the parameters from the stack and writes return values back to the stack.

For remote invocations (more specifically, when caller and object don't share the same stack), some code (typically the COM libraries) needs to read all parameters from the stack and write them to a flat memory buffer so they can be transmitted over a network. The process of reading parameters from the stack into a flat memory buffer is called "marshaling." Parameter marshaling is nontrivial because parameters can be arbitrarily complex—they can be pointers to arrays or pointers to structures. Structures can in turn contain arbitrary pointers and many data structures even contain cyclic pointer references. In order to successfully remote a method call with such complex parameters, the marshaling code has to traverse the entire pointer hierarchy of all parameters and retrieve all the data so that it can be reinstated in the object's process space.

The counterpart to marshaling is the process of reading the flattened parameter data and recreating a stack that looks exactly like the original stack set up by the caller. This process is called unmarshaling. Once the stack is recreated, the object can be called. As the call returns, any return values and output parameters need to be marshaled from the object's stack, sent back to the client, and unmarshaled into the client's stack.

COM provides sophisticated mechanisms for marshaling and unmarshaling method parameters that build on the remote procedure call (RPC) infrastructure defined as part of the distributed computing environment (DCE) standard. DCE RPC defines a standard data representation, the Network Data Representation (NDR), for all relevant data types. In order for COM to be able to marshal and unmarshal parameters correctly, it needs to know the exact method signature, including all data types, types of structure members, and sizes of any arrays in the parameter list. This description is provided using an interface definition language (IDL), which is also built on top of the DCE RPC standard IDL. IDL files are compiled using a special IDL compiler (typically the Microsoft IDL (MIDL) compiler, which is part of the Win32 SDK). The IDL compiler generates C source files that contain the code for performing the marshaling and unmarshaling for the interface described in the IDL file. The client-side code is called the "proxy," while the code running on the object side is called the "stub." The MIDL generated proxies and stubs are COM objects that are loaded by the COM libraries as needed. When COM needs to find the proxy/stub combination for a particular interface, it simply looks up the Interface Id (IID) under the HKEY_CLASSES_ROOT\Interfaces key in the system registry and reads the ProxyStubClsid key:

```
REGEDIT4
[HKEY_CLASSES_ROOT\Interfaces\{<IID_Interface>}\ProxyStubClsid32]
    @={<CLSID_ProxyStub>}
[HKEY_CLASSES_ROOT \CLSID\{<CLSID_ProxyStub>}\InprocServer32]
    @="c:\proxy-stub.dll"
```

Marshaling Interface Pointers

New interface pointers can appear as the result of an activation call like **CoCreateInstance** or as a parameter to a method call. To handle these cases in a uniform way, COM adds a special data type not in DCE RPC—the interface pointer. Marshaling and unmarshaling of this new data type simply means creating a proxy/stub pair capable of handling all the methods in the interface. The recipient (= unmarshaler) of the "interface pointer" data type simply gets a pointer to the proxy object, while the originating interface pointer (on the side where marshaling happens) is handed to the stub in the following case:

- When a method contains an out parameter that is an interface pointer, COM simply marshals this interface pointer on the object side (creating a stub) and unmarshals it on the client side (creating a proxy). The client can then call the proxy, which calls the stub, which calls the interface method.
- When a method contains an in parameter that is an interface pointer, COM marshals the interface pointer on the client side (creating a stub) and unmarshals the interface pointer on the object side (creating a proxy). The object can then call the proxy, which calls the stub (on the client side!), which calls the interface method on the client side.
- When a client calls activation APIs like **CoCreateInstance**, COM typically creates an object and obtains an interface pointer representing this object. It then marshals this interface pointer on the object side as if it was an out parameter to a method and unmarshals it on the client side.

A single mechanism covers all cases!

Like most areas of COM, the process of marshaling an interface pointer can be extended or even overridden. COM notifies the object that it is about to create a proxy and stub for a given interface. The object then can decide to use COM's standard marshaling routines or can provide custom proxy and stub objects.

All COM really is interested in when marshaling an interface pointer is a chunk of data to be placed into the NDR buffer. When COM unmarshals the chunk of data, it needs to be able to get to a COM object that implements the interface being marshaled. This chunk of data is also called an object reference (OBJREF). COM defines three kinds of OBJREFs:

- OBJREF_STANDARD
- OBJREF_HANDLER
- OBJREF_CUSTOM

The **CoMarshalInterface** API is used to create OBJREFs from interface pointers. **CoMarshalInterface** first determines which interface marshaling mechanism the object wants to use. It does this by querying the object for two different interfaces: **IMarshal** and **IStdMarshalInfo**. If the object exposes **IMarshal**, it is indicating that it wants to completely override marshaling for all of its interfaces. If the object exposes **IStdMarshalInfo**, it merely wants to modify some functionality of the proxy while relying on COM's standard marshaling for connections between the proxy and the object. If the object does not expose either interface, COM assumes standard marshaling.

Given an OBJREF, the **CoUnMarshalInterface** API is used to recreate an interface pointer. The exact sequence of steps depends on the kind of OBJREF being unmarshaled and is described below.

Standard interface marshaling

If an object implements neither **IMarshal** nor **IStdMarshalInfo**, COM uses standard interface marshaling based on the Interface ID of the interface to be marshaled. On the object side, COM looks up the IID in the registry and obtains the CLSID of the stub in-process object. COM then creates the stub object and initializes it with the interface pointer to be marshaled. The stub holds on to the interface pointer (it calls **IUnknown::AddRef**) and will use the pointer when it is asked to perform a method invocation. During marshaling, COM writes the IID to the OBJREF, along with information that enables a proxy to connect back to the correct server (object exporter [OXID]), object (object identifier [OID]), and interface (interface pointer identifier [IPID]).

When unmarshaling the OBJREF, COM reads the IID for the object and looks up the CLSID for this IID. COM then creates a proxy object and initializes it with the remainder of the OBJREF information (OXID, OID, and IPID). Finally, COM queries the newly created proxy object for the IID and returns the resulting interface pointer as the result of the unmarshaling operation.

If the interface pointer is used, the proxy will receive the method call. It will then copy the parameters from the stack into an NDR buffer and send this buffer to the "address" indicated by the combination of OXID, OID, and IPID. The COM libraries on the receiving side will find the stub based on the OID and IPID information and hand it the NDR buffer. The stub will read the NDR buffer and set up a stack that corresponds to the stack on the client side. The stub then calls the object.

For most interfaces defined by Microsoft, marshaling support is provided automatically. For example,

IDispatch, **IPersistStorage**, and **IDataObject** all have marshaling support. Interfaces available on your machine are defined in the HKEY_CLASSES_ROOT\Interface section of the registry. Each interface that supports marshaling will have a ProxyStubClsid32 entry.

Proxy/stubs for standard marshaling can be generated using MIDL.

Custom object marshaling

There are times that you want to take full control over the communication between the client and the object. Custom object marshaling lets the implementor of an object take full control over the way interface pointers to this object will be marshaled. The decision to use custom marshaling is an all-or-nothing decision; an object has to custom marshal all its interfaces or none of them. The object indicates that it wants to custom marshal itself by implementing the **IMarshal** interface.

Whenever an interface pointer is to be marshaled, COM queries the object for **IMarshal**. Through **IMarshal**, the object indicates the CLSID of the custom proxy to be used for the object as well as arbitrary data to be used to initialize the proxy object when the interface pointer is being unmarshaled. COM provides an **IStream** interface that the object can use to write the custom proxy initialization data. COM uses the information obtained through **IMarshal** to generate an object reference of type OBJREF_CUSTOM. This object reference contains the Interface ID being marshaled, the CLSID to use for the custom proxy and the custom initialization data for the proxy.

When unmarshaling an object reference of type OBJREF_CUSTOM, COM simply reads the CLSID and creates an instance of this class through **CoCreateInstance**. It then queries this object (the custom proxy) for **IMarshal** and hands it the initialization data through **IMarshal::UnMarshalInterface**. The custom proxy then needs to validate that it is the first proxy for this instance of the original object in this process. If it fails to do so, multiple proxy objects for the same original object can be created, which breaks the COM rule for object identity. Clients that test whether two objects are identical will receive erroneous information.

The newly created custom proxy then returns a pointer to the correct custom proxy (either to itself or to a previously created custom proxy for the same original object) from **IMarshal::UnMarshalInterface**.

Replacing the transport

One use of custom marshaling is to plug in legacy transports into the COM and DCOM world.

Example: Replacing ORPC with a non-DCE–compliant RPC transport

Often it is necessary to interface a COM/DCOM-based application with existing applications or systems. One approach to doing this is to wrap the legacy component with a COM component. If you must use a non-DCE–compliant RPC transport (or a non-ORPC–compliant DCE RPC transport), then you could use custom marshaling in the following manner.

As the object is asked to marshal itself, it performs the necessary steps to set itself as a server for the non-DCE–compliant RPC via whatever mechanism might be required. The object then obtains some sort of handle, string binding, or other means of identifying the exact RPC server endpoint and writes this handle as the custom proxy initialization data. The object also indicates the CLSID of the custom proxy to be used on the unmarshaling end.

When COM unmarshals the interface pointer, it will create a new instance of the custom proxy and pass it the initialization data. The custom proxy reads the handle (or string binding) and uses whatever custom mechanism is available to connect to the original server using the non-DCE–compliant RPC transport. The custom proxy will then receive standard COM calls and will marshal them using whatever marshaling mechanism is available for the non-DCE–compliant RPC transport.

To clients of the object, the new transport is completely transparent: all they ever see is the custom proxy object and they call it just like any other COM object.

Static objects: Marshaling by value

Some objects can essentially be considered static: regardless of which methods are called, the state of the object does not change. Instead of accessing such an object remotely, it is possible to copy the static state of the object and create a new object with the same state information on the caller side. The caller won't be able to notice the difference, but calls will be more efficient because they do not involve network round trips.

Objects can use custom marshaling to effectively copy themselves into the client process. The object implements **IMarshal** and writes all its state into the marshaling buffer that is passed to it via the **IStream** parameter of **IMarshal::MarshalInterface**. It also indicates its own CLSID via **IMarshal::GetUnmarshalClass**.

When COM unmarshals the interface pointer, it will create a new instance of the object and call **IMarshal::UnMarshalInterface** on it. The object can then read all its state information from the marshaling buffer.

Example: Monikers

COM uses marshaling by value for the moniker objects it implements. The state of a file moniker is essentially the file name and path, and the state of a URL moniker is the stringized URL. Marshaling an interface to a file or URL moniker causes a new instance of the moniker object to be created in the client process and initialized with the file path or URL.

Example: Query results

Many applications involve data retrieval. For example, a client calls an object to retrieve a set of parameters or rows of data in a relational table. One way to implement this is by defining a method for each set of data that is to be returned: each column in a row corresponds to a parameter in the method. The disadvantage of this approach is that changes to the data schema (adding or removing a column) require changes to the method signature and thus changes to both clients and servers.

Another approach defines individual **GetProperty** and **SetProperty** methods for each column in the table, making it easy for clients to access the data. The disadvantage of this approach is that this interface cannot be efficiently remoted. Each request for each property causes a network round trip to the server.

With custom marshaling, the benefits of the two approaches can be combined. The actual server object can return a pointer to another, independent object, whose only purpose is to hold the data. This "data object" can be custom marshaled by value, and once the query has been finished, the data in the object is static and immutable. A single network round trip retrieves all the data, and clients use the same simple programming model to actually access the data.

The Microsoft Active Data Connector (ADC) uses by-value marshaling to efficiently remote OLE DB row sets. Future versions of ADC will keep a pointer to the original row-set object using a technique described below.

Handler marshaling

In the previous examples, the connection between the proxy and the original object was purposefully broken since no roundtrips were necessary or desired. In many cases, it makes sense to have special logic in the proxy object and still be able to connect back to the original object for some operations.

Objects can choose to do handler marshaling on a per-object basis. Each and every interface on the object will first be routed through the handler (the custom proxy), however, the handler can choose to forward any of these calls to the original object. Objects indicate their desire for implementing handler marshaling by implementing the **IStdMarshalInfo** interface. Through this interface, the object simply indicates the CLSID for the proxy (handler) to be used in the client process.

COM will create the handler as an in-process object on the client side. The handler then has to aggregate into the OLE default handler, which in turn is initialized with the "backdoor" pointer to the original object. When the handler needs to call the original object, it simply calls the OLE default handler, which in turn calls the original object using standard marshaling.

Note that the OLE default handler is not currently thread safe, which prohibits the use of handler marshaling in free-threaded apartments.

Example: Progressive query results

In the previous example, the entire set of data is sent to the client at once. This is not desirable for large result sets of multiple megabytes or even gigabytes of data. To change this, the query object could use handler marshaling instead of custom marshaling and return the query results incrementally. The proxy keeps a reference to the original object. As the client requests the data, the proxy can use the network round trip to actually read more data than the client asked for. If the client later requests more data, the proxy can satisfy the request locally.

With handler marshaling however, the proxy object can not be "primed" with data. After being created it has to call the original object to retrieve the initial batch of data.

Custom marshaling using standard marshaling: Free-threaded handlers

By combining standard marshaling and custom marshaling an object can implement the functionality of handler marshaling with cache priming in a thread-safe manner.

COM provides a mechanism to create a standard-marshaled object reference (OBJREF_STANDARD) for any interface regardless of whether the object chooses to implement custom marshaling or not. The **CoGetStandardMarshal** API returns a COM object that implements **IMarshal** given an arbitrary interface pointer. This **IMarshal** can be used to write an object reference to a marshaling buffer by calling **IMarshal::MarshalInterface**. The marshaling buffer can then be passed to **CoUnMarshalInterface**, which will return an interface pointer to a standard interface proxy.

Objects that implement **IMarshal** can call **CoGetStandardMarshal** on one or more of their interfaces inside of their own **IMarshal** implementation. They can then write one or more standard object references into their custom marshaling data. The custom proxy can then call **CoUnMarshalInterface** inside of its implementation of **IMarshal::UnMarshalInterface** for each of the object references.

The DBCOMNET sample, available at the Microsoft Web site (http://www.microsoft.com/), illustrates how this can be done.

Return to Contents

Connection Management

Distributed Garbage Collection

The primary mechanism for controlling an object's lifetime is reference counting, using the **AddRef** and **Release** methods of **IUnknown**. **AddRef** and **Release** are called quite often, and sending every call to a remote object would introduce a serious network performance penalty. Hence, DCOM optimizes **AddRef** and **Release** calls for remote objects.

The optimization process uses OXID objects, which implement the **IRemUnknown** interface. An OXID determines the RPC string bindings used to contact a group of objects. Remote reference counting is conducted per interface (per IPID) using the **RemAddRef** and **RemRelease** methods of **IRemUnknown**. Using a single call, **RemAddRef** and **RemRelease** can 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**. For example, the standard proxy implementation defers the actual remote release of all interfaces on an object until all local references to all interfaces on that object have been released. Further, one actual remote reference count is used to service many local reference counts.

Pinging

Remote reference counting would be entirely adequate if clients never terminated abnormally, but in fact they do, and the system needs to be robust in the face of clients terminating abnormally when they hold remote references.

Pinging is a well-known mechanism for detecting when clients terminate abnormally. At the server machine, each exported object (each exported OID) has a *pingPeriod* time value and a *numPingsToTimeOut* count, which combine to determine the overall amount of time known as the *ping period*. If the ping period elapses without receiving a ping on an OID, all the remote references to interfaces associated with that OID are considered "expired" and can be garbage collected, based on local reference information. (Existing implementations of DCOM use pingPeriod = 2 minutes and numPingsToTimeOut = 3. These values can not be changed.)

Pinging on a per-object basis can be very inefficient, thus DCOM contains an optimized pinging infrastructure.

Pings are sent and received by OXID resolvers. The resolver determines which OIDs are on the same machine and generates a ping set. A single ping is sent for the entire set.

Connection Points

Many real-world distributed applications require bidirectional communication between two objects. An object may need to notify a client of a certain event or objects might want to push data in real time as it becomes available.

COM's symmetric programming model makes it extremely easy for both clients and objects to implement this kind of infrastructure. A client simply passes an interface pointer to the server; the server can use this interface pointer to send notifications or data. This is the COM equivalent of callback interfaces.

Connection points standardize passing an interface pointer to an object. The mechanism also solves the problem of a cyclic reference between two objects: since both objects hold references to each other neither object can ever be released if not through some out-of-band mechanism beyond **AddRef** and **Release**. With connection points, the object implementing the interface for registering the "callback" (**IConnectionPoint**) is required to be a separate object from the actual object.

The COM connection point architecture is described in detail in "COM Specification" (MSDN Library, Specifications).

For distributed applications, the multicast feature of connection points can be extremely useful. Multiple clients register themselves with the same connection point, and the connection point sends notifications to all the clients. However, since the multicast must synchronously call each client in turn, dealing with fragile network links, slow clients, and network time-outs require additional design.

One approach is to use multiple threads to notify several clients simultaneously. If a client does not respond within a reasonable time, the object spawns an additional thread to notify the next client.

With Asynchronous DCOM (scheduled to be available in Microsoft Windows 2000) it will be significantly easier to implement multicasting in connection points. Objects will just call the client and COM automatically frees the thread for further calls.

Referrals

An useful feature of the COM programming model that proves especially valuable for distributed applications is the ability to efficiently pass object references from one machine to the other; the marshaling infrastructure described previously handles interface marshaling transparently. Suppose that a client has an interface pointer to an object on another machine. This object in turn holds an interface pointer to a second object on a third machine. The client can then call a method on the first object, which returns the interface pointer to the second object. If the client uses this interface pointer the calls go directly to the second object without even involving the other object or the other machine. Even if the first object (or the machine it is running on) disappeared, the client could still call the second object.

When the client calls the first object and the object returns the interface pointer to the second object, COM's marshaling code will generate an NDR buffer containing all out parameters of the call. One of these parameters is the interface pointer. As described before, COM will call **CoMarshalInterface** on this interface pointer, which in turn checks for custom and handler marshaling (**IMarshal** and **IStdMarshalInfo** interfaces on the second object). If the object does not request custom marshaling, COM writes a standard object reference (OBJREF_STANDARD), which contains the endpoint information (OXID, OID, and IPID) for the second object.

As the call returns to the client, COM unmarshals the object reference, creates a proxy by looking up the Interface ID in the registry and initializes this proxy with the endpoint information. The proxy connects to the machine (as indicated by the OXID) and to the second object (as indicated by OID and IPID). The client receives an interface pointer to the proxy object, which it can use to directly call the second object.

Example: "DcOOM" multiplayer game

The "DcOOM" adventure game (similarities with eventually existing adventure games are purely coincidental) is implemented as a COM object, the "DcOOM Object" with CLSID_DcOOM. This COM object is at the same time a client to other DcOOM objects. When a DcOOM object connects to another DcOOM object, it passes a pointer to itself to the other DcOOM object, establishing a totally symmetric peer-to-peer interaction as shown in Figure 5.

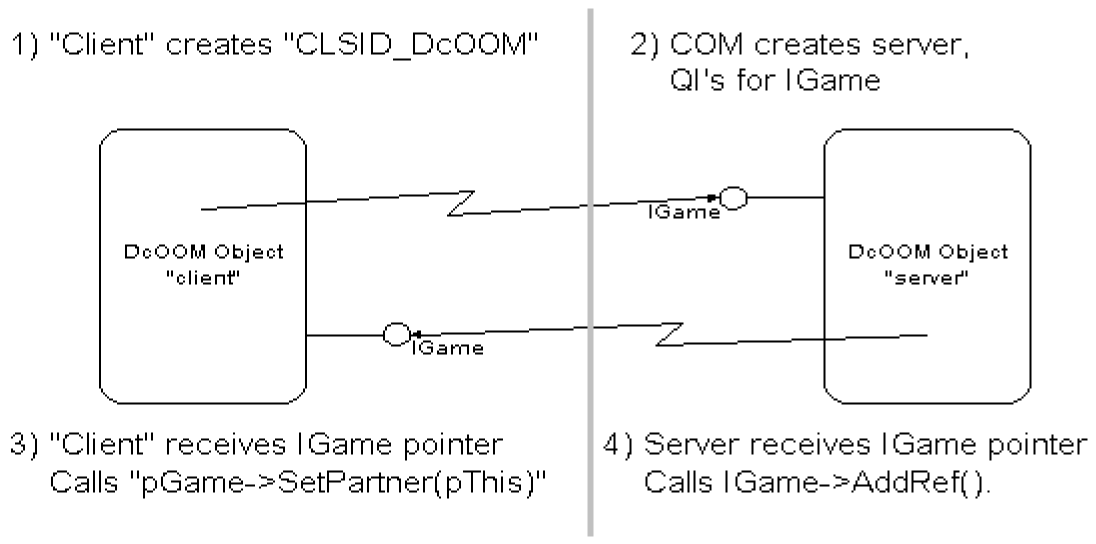

1) "Client" creates "CLSID_DcOOM"
2) COM creates server, QI's for IGame
DcOOM Object "client"
DcOOM Object "server"
IGame
IGame
3) "Client" receives IGame pointer Calls "pGame->SetPartner(pThis)"
4) Server receives IGame pointer Calls IGame->AddRef().

Figure 5

DcOOM players can register their "DcOOM" objects with a central directory component. Other players connect to this "DcOOM" directory component and select another player. At this point, the directory component returns an interface pointer to the selected "DcOOM" object. COM's marshaling infrastructure connects the first "DcOOM" object directly to the selected "DcOOM" object; the directory component is not required anymore.

Return to Contents

Concurrency Management: Threading Models

Why does COM require a threading architecture? In a certain sense, saying that there is a threading architecture within COM is a misnomer. COM merely utilizes the native multithreading capabilities of the operating system that it is running on. The Win32 programming model, for example, provides a certain threading model, which provides multiple threads of execution within a completely shared-memory space synchronized by various system implementations of semaphore primitives.

DCE RPC requires that RPC messages be processed on arbitrary threads. RPC manages a thread pool from which it freely draws a thread whenever an RPC call arrives from the network. RPC developers need to make sure that shared data structures are sufficiently protected against potential simultaneous access from multiple threads.

Most operating systems provide simpler threading models for user interface elements. Win32 for example, only dispatches messages for a particular window to the thread which created that window. This means that a message handler for a particular window need only be reentrant if it explicitly releases the thread to the Win32 dispatcher. In no case does a message handler have to be prepared for parallel execution with two or more threads. This immensely simplifies the programming model since the developer has to deal with concurrency only in very special and limited situations.

COM's programming model scales with the need of the application developer. It provides totally free-threaded method invocations, just like RPC does, but it also provides an automatic synchronization of method invocations to a single thread, even synchronized with the dispatching of window messages.

Single-Threaded Apartments (STA)

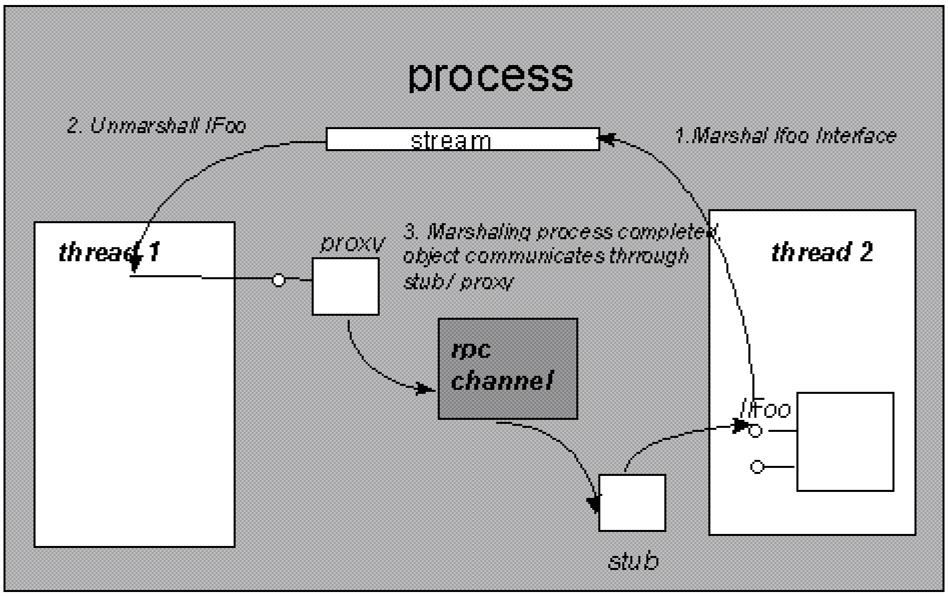
In a single-threaded apartment, each object lives in a thread, just like a Win32 window lives on a given thread. Each thread must initialize COM using **CoInitialize** or using **CoInitializeEx** with COINIT_APARTMENTTHREADED as the second parameter. Each thread must also have a message pump that dispatches window messages, since COM creates a hidden window that it uses to synchronize incoming RPC calls with messages to other windows that might be created on the same thread. If the thread does not dispatch messages, certain messages sent to the hidden window created by COM (for example, a **SendMessage** call to all windows on a system, as used by the shell) will never be retrieved, causing the caller to wait infinitely.

If a client running on another thread (in another "apartment") wants to call an object in a single threaded apartment, the two threads need to be forcibly synchronized. COM achieves this by marshaling interface pointers across threads, just as it does for marshaling across process or across machines; because the two threads have different stacks, the parameters need to be copied from one stack to the other.

A simple way to think of the apartment model from a marshaling perspective is that each thread is a separate process. Same-thread access to an object is direct; access to an object from a different thread is indirect through proxy/stub code. Multiple client threads may access an object; however, they may not retrieve an interface pointer to the object directly (that is, through a global variable). Instead the interface pointer must be explicitly marshaled on the object thread and unmarshaled on the client thread. Instead of placing an interface pointer in the global variable, the marshal buffer with the marshaled interface pointer (an object reference as described above) must be stored. To obtain the marshaling buffer, application developers can use the COM marshaling APIs as follows:

1. On the object's thread, the object's interface has to be written into a stream using CoMarshalInterface or CoMarshalInterThreadInterfaceInStream. This triggers the same marshaling process as described above. COM queries for an IMarshal interface to the object and asks it to put necessary information in the stream. If the object does not implement IMarshal, COM falls back to standard marshaling and creates a stub for the interface.
2. The resulting stream interface pointer can then either be placed into a global variable or the data can be read and written into a global memory block.
3. On the client's thread, the same stream (or the memory block rewritten into a stream object) needs to be passed to CoUnMarshalInterface or CoGetInterfaceAndReleaseStream. Under the covers, COM will create a proxy, query for IMarshal interface on the proxy, and pass the stream so that the proxy will get the necessary information to connect to the stub.

Once this process is done, the client thread can call the interface pointer freely and COM will synchronize all calls to the object.



Fingure 6

The basic concurrency unit in an STA is the individual thread that initializes COM. If two objects, for example A and B, live in the same apartment, and A is processing a call, no other client can make a call, to either A or B, until A completes its service. However, if B lives in a different apartment, B can accept a call while A is still processing and vice versa.

Classes that need to be created simultaneously in different apartments have to protect any global data that is shared between object instances using thread synchronization primitives. However, instance data that is exclusive to an object need not be protected, because only one thread can ever enter this instance of the object.

COM can still run objects that do not protect their global data; these objects are always created in the STA that was initialized first in the process. This apartment is called the main apartment of the process. This ensures that only this single thread will ever enter instances of this class.

In all cases, objects in an STA need to protect their instance data against reentrancy, similar to the way window procedures need to be reentrant. Any action that might cause dispatching of window messages can trigger reentrancy. Calling another COM object in a different apartment is one of these actions: COM will process incoming calls in this situation to avoid deadlocks.

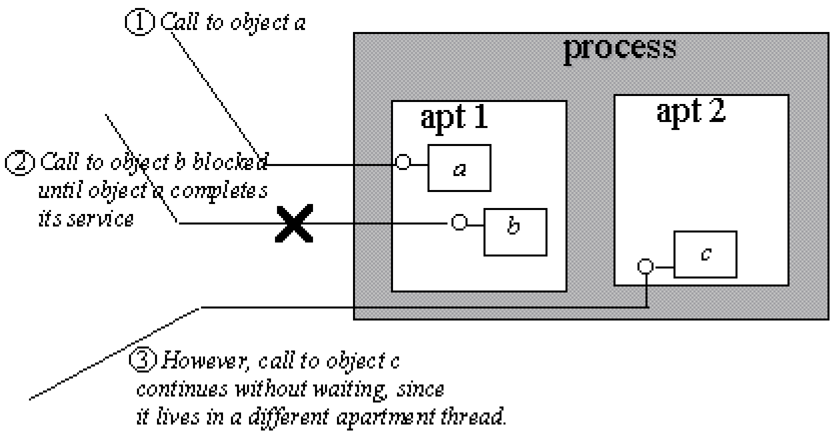

① Call to object a
process
apt 1
apt 2
a
b
c
② Call to object b blocked until object a completes its service
③ However, call to object c continues without waiting, since it lives in a different apartment thread.

Figure 7

Method invocation between STAs

What exactly happens when a client calls an object in another apartment?

The journey starts when a thread calls a method on an interface. Since the interface pointer actually points to a proxy, the proxy reacts by preparing packet information and sending it to the RPC channel. The RPC channel detects that the target of the call is actually a thread in the same process and posts a message to the thread of the target apartment (using **PostThreadMessage**). The channel then blocks the calling thread using **MsgWaitForMultipleObjects**. If messages arrive on the caller's thread, the channel dispatches these messages so that the calling application remains responsive to the user and repaints its windows. Applications can hook into this message dispatch mechanism by implementing a message filter, as discussed in the section "Controlling incoming calls: IMessageFilter," in "Concurrency Management: Threading Models."

The thread of the receiving apartment processes the message that was posted by the channel of the calling apartment. The hidden COM window processes this message and invokes the appropriate stub. The stub unmarshals the packet onto the stack and invokes the object's implementation. Once the invocation has completed it returns to the stub. The stub packages any out parameters and return values, and posts a message to the calling thread.

The calling thread is still waiting inside the channel and dispatches messages as they arrive. The message posted by the channel in the target apartment is dispatched and reaches COM's window procedure. The window procedure retrieves the marshaled return values and returns. As the windows procedure returns, the channel returns the marshaled return values to the proxy, which in turn places them onto the caller's stack and returns to the caller.

If another method call occurs for an object within the calling apartment while the cross-apartment call is in progress, the call will be handled by COM. The message loop provided by COM for the outgoing call is dispatching any messages in the queue. If a call comes in for an object in this apartment, COM posts a message to the hidden COM window. This message will be picked up by the message pump and rerouted to the window procedure implemented by COM. This window procedure calls the stub to unmarshal and invoke the object. Upon returning from the object's method, the stub will marshal the results and return to the window procedure, which sends it to the receiving thread. The thread execution returns to the COM-provided message loop, which is waiting for the original outgoing call to succeed.

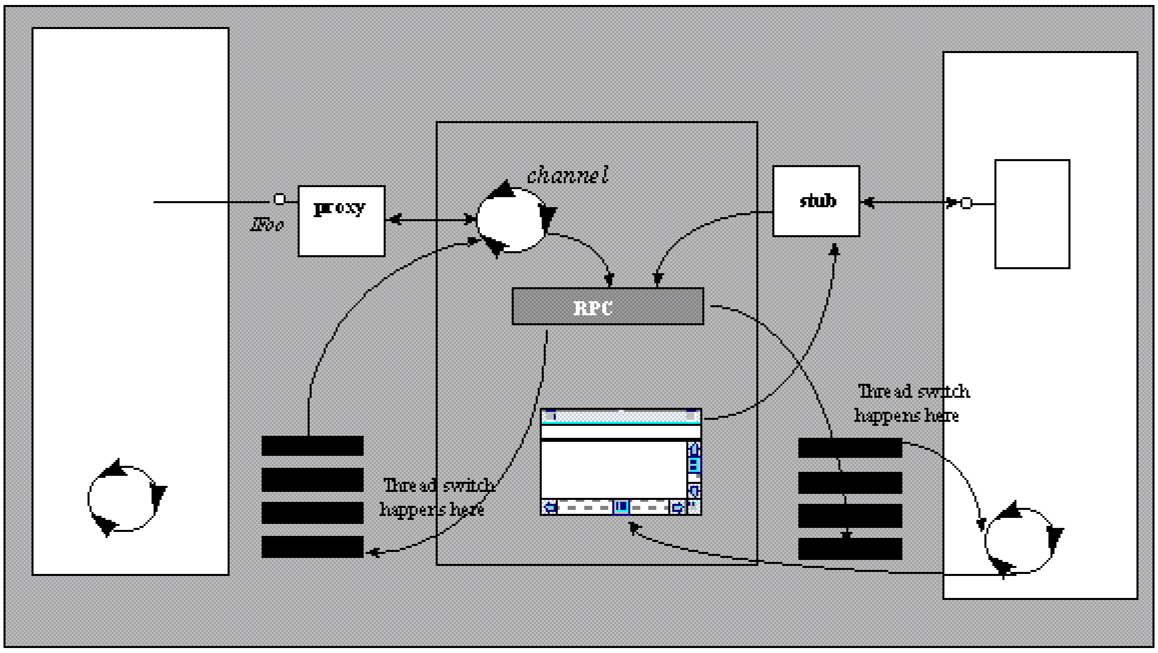


Figure 8

Method invocation from an STA to another process

When a client in an STA calls out to another process—be it on the same machine or one a different machine—an additional thread gets involved to handle the actual RPC call. Instead of posting a message to the other object's thread—as in the STA-to-STA case—the proxy picks an RPC thread from the RPC thread pool and hands it the marshaled parameter data. This RPC thread then blocks until the call returns. The calling thread, however, enters a message loop that keeps the user interface responsive and prevents deadlocks when other calls enter this apartment as part of the initial call, as discussed above.

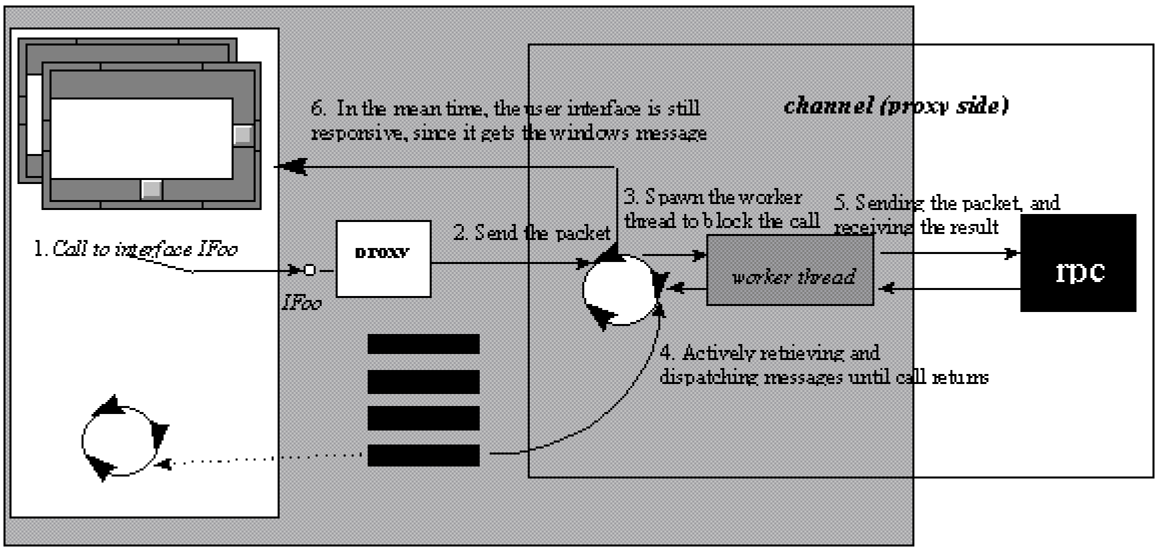

6. In the mean time, the user interface is still responsive, since it gets the windows message
channel (proxy side)
3. Spawn the worker thread to block the call
5. Sending the packet, and receiving the result
1. Call to interface IFoo
proxy
2. Send the packet
worker thread
rpc
IFoo
4. Actively retrieving and dispatching messages until call returns

Figure 9

Method invocation into an STA from another process

Similarly, when a call from another process arrives in an STA, it first arrives on an RPC thread that the RPC run-time picks from the RPC thread pool. The RPC thread then posts a message to the hidden COM window, passing it the marshaled parameter data. From here on the sequence is identical to the sequence in calls between STAs.

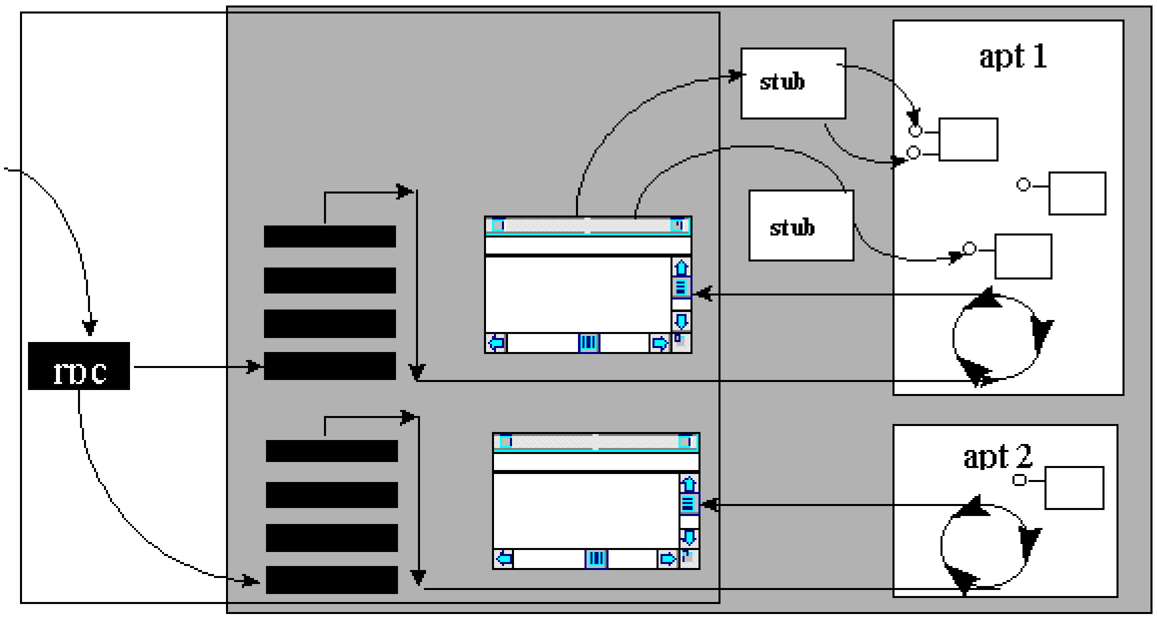


Figure 10

Controlling incoming calls: IMessageFilter

COM provides a process-wide mechanism to control how incoming calls and window messages in an STA are handled. Any incoming call to a server, or outgoing call via sink or notification from a server, can be rejected, retried later, or—the default—handled by the server.

For outgoing calls, the filter can detect if the RPC connection still exists, if a server rejected the call, or if the server told the client to retry the call. The filter can also control how standard window messages are handled while COM is waiting for an outgoing call to return.

To install a message filter, you call **CoRegisterMessageFilter** passing the **IMessageFilter** interface. To uninstall the filter, use the same API and pass NULL for both parameters. The message filter is only used in an STA.

Class factory for in-process servers

In designing in-process servers for STAs, all DLL entry points (**DllGetClassObject** and **DllCanUnloadNow**) must be thread-safe, since it is possible for two or more threads to call **CoCreateInstance(Ex)** at the same time. If **DllGetClassObject** returns the same **IClassFactory** on every call, then the returned **IClassFactory** must be thread-safe. For example, the reference count of the class factory must be thread-safe. If **DllClassObject** returns different class factories (**IClassFactory**) on every call, then the returned **IClassFactory** does not need to be thread-safe.

Multithreaded Apartment (MTA)

From COM's perspective, a multithreaded apartment (MTA) is an easier model compared to an STA. Incoming RPC calls directly use the thread assigned by the RPC run time. No message pump or hidden window is used for communication between the client and object. The object does not live in any specific thread. Within a process, clients from any thread can directly call any object inside the MTA. (There is only one MTA per process.) Interface pointers between threads do not need to be marshaled.

However, an MTA requires extreme caution on the part of the object implementor. Multiple threads can call an object method at the same time. Therefore the object must provide synchronized access to any instance using synchronization primitives like events, mutexes, or semaphores. It must be completely thread safe and reentrant.

Although complex to implement, MTA-aware objects offer the possibility of higher performance and better scalability than STA objects. The generic synchronization that COM performs on STA is relatively expensive. On each method call across apartments, two window messages need to be sent. On remote calls, two additional thread switches are required on the server side: the RPC thread switches to the apartment thread, calls the method, and switches back to the RPC thread upon completion. (Note: Additional thread switches are required on the client side, as well.)Thus objects in MTAs provide better scalability in terms of the fixed COM-related overhead per message call, even if they process only one call at a time (using "brute force" synchronization that locks the entire object instance as soon as a method enters the object).

The generic synchronization is also expensive in a more subtle way. Often methods don't access instance state at all, or different methods access different parts of the instance state. By optimizing the locks that an object takes for each method invocation, the same object can in fact process multiple simultaneous calls on different methods or even on the same method. Optimizing locks in this fashion is hard, since a single overseen race condition can introduce severe and hard-to-trace faults and data corruption. However, depending on the nature of the object, this manual lock optimization can provide significant performance improvements. In particular, objects that depend on lengthy external actions (for example, a database query or a file operation) or that perform lengthy calculations without accessing instance state can benefit hugely and are relatively easy to optimize. A good design technique is to isolate such critical areas of an application into separate objects. Noncritical parts can use "brute force" synchronization or even run in STAs, while the critical objects can be moved to MTAs and hand-tuned to achieve high overall performance and scalability with minimum effort.

A thread that wants to be in a multithreaded apartment must initialize COM by calling **CoInitializeEx** (NULL, COINIT_MULTITHREADED). If an object is created from this thread (using **CoCreateInstance** or moniker binds), COM will assume that the object is free-threaded and does not require any synchronization.

Method invocation from an MTA

If a client is running in an MTA (on any thread that has been initialized as free-threaded), calls to objects in the same apartment do not require any synchronization. Whichever thread the client uses is the thread that the object will be invoked on. If calls go to other apartments or leave the process or even the machine, the proxy/stub infrastructure will be used. The proxy and stub are automatically inserted whenever an interface pointer is made available across apartments (using activation APIs, out parameters on method calls, or explicit marshaling using **CoMarshalInterface** and **CoUnMarshalInterface**).

The client calls the proxy, which in turn calls the RPC channel. Because the client is running in an MTA, the RPC channel blocks the caller's thread instead of switching to an RPC thread and pumping messages on the caller's tread. Windows that were created in this thread freeze for the duration of the call. Windows that were created in different threads, however, can still process their messages. Once the call returns, the thread is signaled and the RPC channel returns to the proxy, which unmarshals results and returns to the caller.

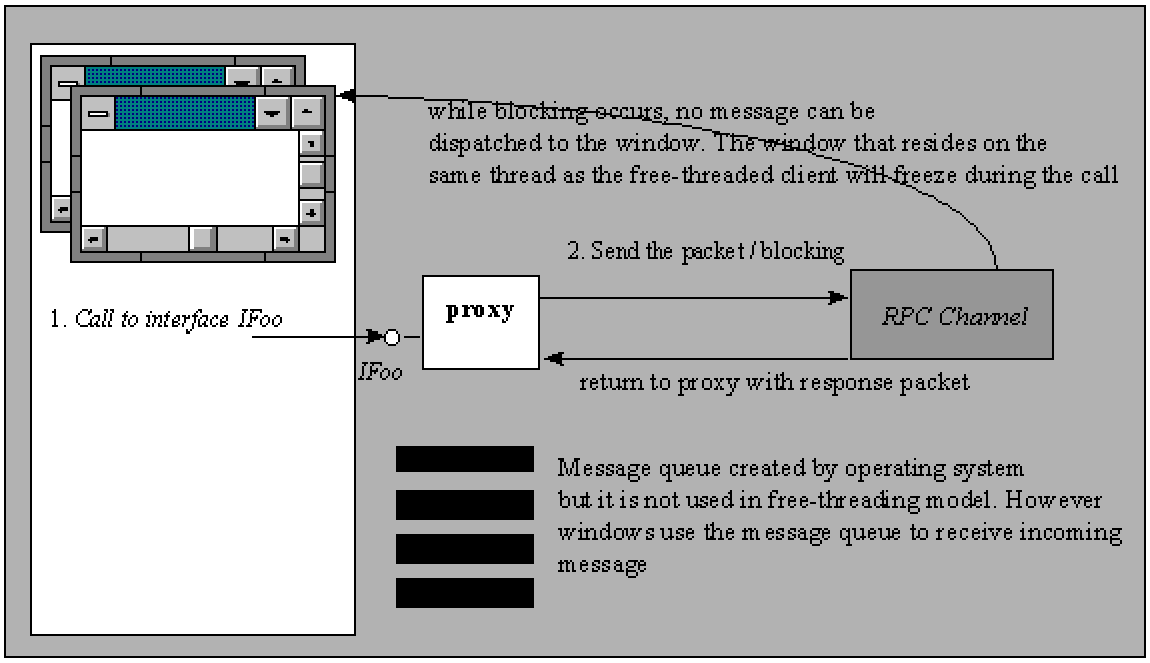

while blocking occurs, no message can be dispatched to the window. The window that resides on the same thread as the free-threaded client will freeze during the call
2. Send the packet / blocking
1. *Call to interface IFoo*
proxy
RPC Channel
IFoo
return to proxy with response packet
Message queue created by operating system but it is not used in free-threading model. However windows use the message queue to receive incoming message

Figure 11

When a call arrives to an object in an MTA, the RPC channel directly uses the RPC thread that it picked from the

RPC thread pool. The channel calls the stub directly and the stub returns directly to the channel, without any additional thread switches.

If multiple clients call this object simultaneously, then each call runs on a different thread. If an excessive number of threads have been created, then COM may block some calls temporarily. If an object makes a call to other object, the calling object can still accept incoming calls. They simply arrive on another thread. The object can not use thread local storage to maintain any state between calls, since there is no correlation between threads and object instances.

How Do Objects Indicate the Threading Models They Support?

COM currently defines three different kinds of apartments that require different assumptions on behalf of the object:

- Single-threaded apartment, main thread only: all instances are created on the same thread.
- Single-threaded apartment: an instance is tied to a single thread, and different instances can be created on different threads.
- Multithreaded apartment: instances can be created on multiple threads, and instances can be called on arbitrary threads.

Local Servers: In full control!

Objects implemented as local servers are in full control of the kind of apartment that COM is using for their objects. The local server calls **CoInitializeEx** on one or more threads and registers the class factory (or multiple class factories) from the appropriate thread with the appropriate threading model.

Example: Worker vs. admin objects

A server might implement several objects that perform calculations. These objects are implemented as free-threaded objects. Other objects are used for administering the calculator objects, and they will call certain methods on the worker objects but perform additional housekeeping under the covers. Since these objects are not performance critical, they are implemented as apartment threaded objects.

The server initializes itself as follows (pseudo-code):

```
main()
{
    // Initialize the main thread as an STA.
    CoInitialize(NULL);
    // Create the class factory object (derived from IUnknown).
    CAdminComponentCF* pAdminCF = new CAdminComponentCF();
    // Register the admin component.
    CoRegisterClassObject(CLSID_AdminComponent,
        (IUnknown*) pAdminCF, CLSCTS_ALL, REGCLS_MULTIPLEUSER, &dwAdminCookie);
    // Create the worker thread.
    CreateThread(NULL, 0, &WorkerThread, NULL, 0, &dwThreadID);
    // Message pump for the apartment threaded components.
    while (GetMessage(&msg, (HWND) NULL, 0, 0)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    // Tell the worker thread that we are done.
    SetEvent(g_evntExit);
    // Wait until the worker thread is done.
    WaitForSingleObject(g_evntWorkerDone, INFINITE);
    // Revoke the admin class factory.
    CoRevokeClassFactory(dwAdminCookie);
    // Uninitialize COM.
    CoUninitialize();
    return 0;
}
DWORD WorkerThread(LPDWORD lpdwParam) {
    // Make this thread be part of an MTA.
    CoInitializeEx(COINIT_MULTITHREADED);
    // Create the class factory for worker components.
    CWorkerClassFactory* pWorkerCF = new CWorkerClassFactory();
    CoRegisterClassObject(CLSID_WorkerComponent,
        (IUnknown*) pWorkerCF, CLSCTS_ALL, REGCLS_MULTIPLEUSER, &dwWorkerCookie);
    // Start working. RPC will call the class factory on its own threads, so we can let this thr
```

```
     WaitForSingleObject(g_envtExit, INFINITE);
     // Main thread has signaled that it's time to leave. Revoke the worker class factory.
     CoRevokeClassfactory(dwWorkerCookie);
     // Signal the main thread that we are done.
     SetEvent(g_evntWorkderDone);
     // Uninitialize COM.
     CoUninitialize();
     return 0;
}
```

The admin components are running in an STA on the main thread. The server needs to implement a message pump for this apartment.

The worker components are created in an MTA on a background thread. The worker components don't need a message pump—RPC will "lend" them threads from its thread pool.

In-process servers: In (almost) total dependency on their client

In-process servers run in the context of their client. They share the same process, and they need to run in the apartment that the client gives them. Unless they spawn their own worker threads for their own internal objects, in-process servers do not get to call **CoInitializeEx** to indicate their preferred threading model.

When COM is asked to create an object on behalf of a client, it needs to know if the object can be loaded directly into the apartment that the client is using. More precisely, COM needs to know whether the object is compatible with the threading model of the thread that is creating the object. If the object is not compatible, COM tries to load the object into a different, compatible apartment—if one exists—and places proxy/stubs between the client and the object.

Table 2 illustrates which assumptions an in-process object has to make when claiming to be compatible with a given threading model.

Table 2

	Message Pump	**One thread for all instances**	**One thread per instance**	**Same thread for all calls**	**Multiple threads per instance**	**Marshal other objects to private threads**
STA-Main	Yes	Yes	Yes	Yes	No	Yes
STA (Apartment)	Yes	No	Yes	Yes	No	Yes
MTA (Free)	No	No	No	No	Yes	No
STA & MTA (Both)	No	No	No	No	Yes	Yes

Note that when a component wants to support all threading models (that is, loadable directly into any apartment), it has to comply with the assumptions in the last row of the table.

In-process components indicate which assumptions they are ready to satisfy by placing a named value (ThreadingModel) under their InprocServer32 key in the registry:

```
REGEDIT4
[HKEY_CLASSES_ROOT\CLSID\{clsid}\InprocServer32]
"ThreadingModel"="Both"
```

or

```
"ThreadingModel"="Apartment"
```

or

```
"ThreadingModel"="Free"
```

If ThreadingModel is not specified, the component is assumed to follow the assumptions for "STA-Main" and can only be loaded into the main STA in a process. A value of "Both" indicates that the component can be loaded in both MTAs and STAs. A value of "Apartment" indicates that the component can be loaded into any STA. A value of "Free" indicates that the component can be loaded into an MTA, but not into an STA (that is, it doesn't marshal pointers that it receives through method calls before calling them from private background threads, the last column in Table 2).

Table 3 indicates how COM matches the capabilities of a component with the current threading model of the thread that is requesting the creation of an in-process object.

Table 3

		Object			
		STA, main thread only	**STA**	**MTA**	**STA and MTA**
	STA, main thread	Load directly	Load directly	Load into MTA thread, marshal to creating thread	Load directly
Thread	**STA, secondary thread**	Load into main thread, marshal to creating thread	Load directly	Load into MTA thread, marshal to creating thread	Load directly
	MTA	Load into main thread, marshal to creating thread	Load into other apartment, marshal to creating thread	Load directly	Load directly

Figure 12 illustrates the possible combinations of clients running in the three kinds of apartments and objects marked as complying with a certain kind of apartment.

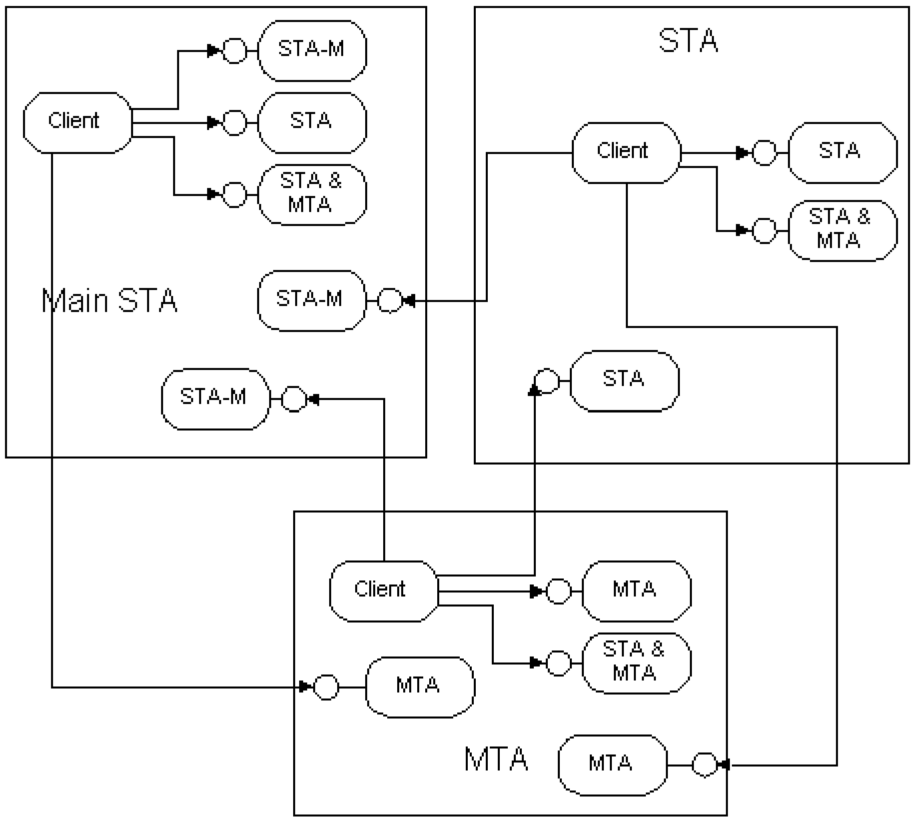


Figure 12

Customizing threading models: The free-threaded marshaler

Since objects living in different apartments always receive a marshaled interface pointer, calls between apartments incur a significant overhead over direct method invocations. A direct invocation is simply an indirect function call (a C++ virtual function call), while a marshaled method invocation requires that the parameters be copied from one stack to the other, as well as two thread switches.

Some objects in an in-process server might be free-threaded while others require an STA. COM provides a backdoor to make clients in other apartments call these free-threaded objects in a "heterogeneous" DLL: custom marshaling! As described in detail previously, COM allows objects to take complete control over the way they are marshaled. This also applies to marshaling between threads.

If an object wants to treat as a special case some objects in a DLL that is otherwise marked as requiring an STA, it can simply implement **IMarshal** for this object. When asked for the marshal data (in **IMarshal::MarshalInterface**), it can test whether it is marshaled across threads (MSHCTX_INPROC), and if so, it can simply write the object pointer into the marshaling buffer (treating the **IUnknown*** pointer as a DWORD). The object indicates its own CLSID as the custom proxy to be used during unmarshaling, and when it is asked to unmarshal, it simply returns the **IUnknown** pointer from the marshaling package in the call to **IMarshal::UnMarshalInterface**.

The effect of this is that any client in any apartment in the object's process will receive a direct pointer to the object. The object will receive multiple calls on arbitrary threads. The object must also be careful not to call out to any other objects that could live in other apartments without previously marshaling their interface pointers to the thread that it is using to call out. This is equivalent to the requirement for supporting both STA and MTA, but is emphasized here again. If an object is marked as STA (ThreadingModel value "Apartment") and customizes cross-thread marshaling, it must marshal external interface pointers even if it never spawns its own private threads nor shares interface pointers between instances. The single instance of the object will be called by multiple threads.

To make implementing this cross-thread custom marshaling easier, COM provides a helper object, the free-threaded marshaler. Objects can aggregate the free-threaded marshaler and delegate all unknown Interface IDs to it. The object implements **IMarshal** to perform the steps described above. The free-threaded marshaler can easily be created using the **CoCreateFreeThreadedMarshaler** API.

The following pseudo-code illustrates how an object can use this helper object:

```
HRESULT CMyFreeThreadedAndWellBehavedSTAObject::QueryInterface(REFIID iid, void* ppvObject)
{
    if (iid==IID_Foo)
    {
        // standard QI implementation for the object's own interfaces
    }
    // Just before the object is ready to give up, it simply calls the free threaded marshaler.
    if (m_punkMarshalHelper) // must be prepared for QIs from the helper m_punkMarshalHelper is
    {
        return m_punkMarshalHelper->QueryInterface(iid, ppvObject); // Delegate to the marshal h
    }
    return E_NOINTERFACE;
}
CMyFreeThreadedAndWellBehavedSTAObject:: CMyFreeThreadedAndWellBehavedSTAObject()
{
    // Standard initialization here.
    Ã,Â…
    m_punkMarshalHelper=NULL;     // must set this since the helper might call QI on its control
    CoCreateFreeThreadedMarshaler(
        (IUnknown*) this,            // Pointer to object aggregating the marshaler object.
            &m_punkMarshalHelper);    // Indirect pointer to the marshaler object.
}
```

The following pseudo-code illustrates how an object can conform to the apartment rules when storing an interface pointer in its instance data:

```
// This method illustrates how an object stores an interface pointer to another object in its in
// Instead of storing the actual pointer we store the marshaled interface pointer (an IStream!).
HRESULT CMyFreeThreadedAndWellBehavedSTAObject::SetOtherObject(IFooNotify* punkOtherObject)
{
    // Need to protect any access to the instance data since we claim to be free-threaded!!!!!
    EnterCriticalSection(m_sectOtherObject);
    // Do we already have a marshaled interface pointer?
    if (m_pstmOtherObject)
    {
        IUnknown* punkTemp=NULL;
        // Yes, we must release the marshaling packet and the stream.
        CoGetInterfaceAndReleaseStream(
             m_pstmOtherObject,    // the marshaled interface pointer
             IID_IFooNotify,            // the IID we want to retrieve
            punkTemp);            // the unmarshaled interface pointer
        // The stream is released, now we need to release the object.
        punkTemp->Release();
        m_pstmOtherObject=NULL; // just to be safe
    }
    // Marshal the new interface pointer and store it in the instance variable "m_pstmOtherObjec
    CoMarshalInterThreadInterfaceInStream(
         IID_IFooNotify,                // the IID of the pointer we received
        punkOtherObject,            // the pointer we received
        &m_pstmOtherObject);        // OUT: the marshaled interface pointer (an IStream)
    LeaveCriticalSection(m_sectOtherObject);
    return S_OK;
}
// This method illustrates how the object uses the stored marshaled interface pointer.
// Since this method could be called on a different thread than "SetOtherObject", the object has
// the pointer to this thread before using it!
HRESULT CMyFreeThreadedAndWellBehavedSTAObject::NotifyOtherObject(DWORD data)
{
    // Need to protect access to instance data, since we claim to be free-threaded.
    EnterCriticalSection(m_sectOtherObject);
    if (!m_pstmOtherObject) // No object yet: fail.
    {
        LeaveCriticalSection(m_sectOtherObject);
        return E_FAIL;
    }

    // Unmarshal the marshaled interface pointer into this thread and release the marshaling buf
    CoGetInterfaceAndReleaseStream(
         m_pstmOtherObject,            // the marshaled interface pointer
         IID_IFooNotify,                    // the IID we want
        punkOtherObjectForThisThread);    // OUT: the interface pointer for our current thread
    // Remarshal the interface pointer so we can unmarshal it again in the future.
    CoMarshalInterThreadInterfaceInStream(
         IID_IFooNotify,                    //Reference to the identifier of the interface.
        punkOtherObjectForThisThread,    //Pointer to the interface to be marshaled.
        &m_pstmOtherObject);            // Marshaled interface
    // Done with the instance data. Let other threads use the object. We have a safe reference a
    LeaveCriticalSection(m_sectOtherObject);
    // Use the interface pointer from this thread.
    Ã‚Â…
    punkOtherObjectForThisThread->Notify(data); // just an example
    Ã‚Â…

    // Release the interface for this thread.
    punkOtherObjectForThisThread->Release();
    return S_OK;
}
```

The following code illustrates how an STA-only object would implement the same functionality:

```
// This method illustrates how an object stores an interface pointer to another object in its in
// Since we are STA we can store and freely use the actual pointer in the instance data.
HRESULT CMySTAObject::SetOtherObject(IFooNotify* punkOtherObject)
{
    // Do we already have an interface pointer?
    if (m_unkOtherObject)
    {
        // Yes, we must release the old interface pointer.
        m_punkOtherObject->Release();
        m_punkOtherObject=NULL; // Just to be safe.
    }
    // Store the pointer.
    m_punkOtherObject=punkOtherObject;
    // Keep a reference, since we are holding on to the pointer.
    m_punkOtherObject->AddRef();
    return S_OK;
}
// This method illustrates how the object uses the stored interface pointer.
//Since we are STA we can store and freely use the actual pointer in the instance data.
HRESULT CMySTAObject::NotifyOtherObject(DWORD data)
{
    if (!m_punkOtherObject) // No object yet: fail.
```

```
        {
            return E_FAIL;
        }

        // Use the interface pointer from this thread.
        Ã‚Â…
        m_punkOtherObject->Notify(data); // just an example
        Ã‚Â…

        return S_OK;
    }
```

A few lines of extra code make the object free-threaded.

If an STA client uses the MTA & STA object, the code for marshaling and unmarshaling the pointers, as well as the critical section, is actually unnecessary. The pointers will always be already usable on the thread and the critical sections will never be taken by any other thread. The overhead for both is very small and may be justified for all but the most demanding STA/inproc-only environment.

If an MTA client uses the MTA & STA object, the only additional overhead is that of the marshaling and unmarshaling code for the interface pointer. Again the overhead is small, but might be relevant for high-performance and highly scalable servers.

One potential problem for MTA & STA objects is storing direct pointers to other objects. If an STA client C creates an MTA & STA object, M, and object M creates an STA object, S, the interface pointer to S is a direct pointer. Now consider if M aggregates with the free-threaded marshaler and C marshals its pointer to M into another apartment A. When A calls through this pointer to M, M can not use the direct interface pointer it has stored to S—it is running in a different apartment than S expects to be called from. The global interface table introduced with Windows NT 4.0 Service Pack 3.0 must be used in this scenario.

Return to Contents

Securing Distributed Applications

Designing a distributed application poses several challenges to the developer. One of the most difficult design issues is that of security. Who can access which objects? Which operations are objects allowed to perform? How can administrators manage secure access to objects? How secure does the content of a message need to be as it travels over the network?

Mechanisms to deal with security-related design issues have been built into DCOM from the ground up. DCOM provides an extensible and customizable security framework upon which developers can build when designing applications.

Different platforms use different security providers and many platforms even support multiple security providers for different usage scenarios or for interoperability with other platforms. DCOM and RPC are architected in such a way that they can simultaneously accommodate multiple security providers.

Common to all these security providers is that they provide a means of identifying a security principal (typically a user account), a means of authenticating a security principal (typically through a password or private key), and a central authority that manages security principals and their keys. If a client wants to access a secured resource, it passes its security identity and some form of authenticating data to the resource and the resource asks the security provider to authenticate the client. Security providers typically use low-level custom protocols to interact with clients and protected resources.

Security Policies

DCOM distinguishes between four fundamental aspects of security:

- **Access security:** Which security principals are allowed to call an object?
- **Launch security:** Which security principals are allowed to create a new object in a new process?
- **Identity:** What is the security principal of the object itself?
- **Connection policy:** Integrity—can messages be altered? Privacy—can messages be intercepted by others? Authentication—can the object find out or even assume the identity of the caller?

Protecting the object: Access security

The most obvious security requirement on distributed applications is the need to protect objects against unauthorized access. Sometimes only authorized users are supposed to be able to connect to an object. In other cases, nonauthenticated or unauthorized users might be allowed to connect to an object, but must be limited to certain areas of functionality.

Current implementations of DCOM provide declarative access control on a per-process level. Existing components can be securely integrated into a distributed application by simply configuring their security policy as appropriate. New components can be developed without explicit security awareness yet still run as part of a completely secure distributed application.

If an application requires more flexibility, objects can programmatically perform arbitrary validations, be it on a per-object basis, per-method basis, or even per-method parameter basis. Objects might also want to perform different actions depending on who the caller is, what specific access rights the caller has, or which user group the caller belongs to.

Protecting the server machine: Launch security

Another related requirement on a distributed infrastructure is to maintain control of object creation. Since all COM objects on a machine are potentially accessible via DCOM, it is critical to prevent unauthorized users from creating instances of these objects. This protection has to be performed without any programmatic involvement of the object itself, since the mere act of launching the server process could be considered a security breach and would open the server to denial of service attacks.

The COM libraries thus perform special security validations on object activation. If a new instance of an object is to be created, COM validates that the caller has sufficient privileges to perform this operation. The privilege information is configured in the registry, external to the object.

Controlling the object: Security identity

Another aspect of distributed security is that of controlling the object. Since an object performs operations on behalf of arbitrary callers, it is often necessary to limit the capabilities of the object. One obvious approach is that of making the object assume the identity of the caller. Whatever action the object performs—a file access, network access, registry access, and so on—is limited by the caller's privileges. This approach works well for objects that are used exclusively by one caller since the security identity is established once at object creation time (see "Run As Activator" in section "Fundamentals: Windows NT Security Infrastructure"). The approach can also be used for shared objects if the object performs an explicit action on each method call (see "Impersonation" in the section "Programmatic Security").

However, for applications with a large number of users, the approach of making the object assume the identity of the caller can impose problems. All resources that are potentially used by an object need to be configured to have exactly the right set of privileges. If the privileges are too restrictive, some operations on the object will fail. If the privileges are too generous (that is, there is write access to some files where only read access is required), security violations might be possible. Although managing access can be simplified by defining user groups, it is often simpler to have the object run under a dedicated security identity, independent of the security identity of the current caller.

Other applications may not even be able to determine the security identity of the caller. Many Internet applications, for example, do not assign a dedicated user account for every user. Any user can use the application and yet the objects still need to be secure when accessing resources. Again, assigning objects a security identity of their own makes this kind of application manageable in terms of security.

Protecting the data: Connection policy

As the "wire" between callers and objects becomes longer, the possibility of data that is being transported as part of method invocations being altered or intercepted by third parties increases. DCOM gives both callers and objects a range of choices to determine how the data on the connection is to be secured. The overhead in terms of machine and network resources tends to grow with the level of security. DCOM therefore lets applications dynamically choose the level of security they require.

Physical data integrity is usually guaranteed by the low-level network transport. If a network error alters the data, the transport automatically detects this and retransmits the data.

However, for secure distributed applications, data integrity really means being able to determine if the data actually originated from a legitimate caller and if it has been logically altered by anyone. The process of authenticating the caller can be relatively expensive, depending on the security provider and the security protocol it implements. DCOM lets applications choose if and how often this authentication occurs (see the following section, "Identifying the caller: Authentication," for details).

DCOM currently offers two fundamental choices with regard to data protection: integrity and privacy. Clients or objects can request that data be transferred with additional information that ensures data integrity. If any portion of the data is altered on its way between the client and the object, DCOM will detect this and automatically reject the call. Data integrity implies that each and every data packet contains authentication information.

However, data integrity does not imply that no one can intercept and read the data being transferred. Clients or objects can request that all data be encrypted. (See "Packet Privacy" in Table 4). Encryption implies an integrity check as well as per-packet authentication information.

Since both privacy and integrity require authentication, the mechanisms for specifying privacy and authentication are unified into a single enumeration of authentication levels, which are described in the next section.

Identifying the caller: Authentication

The above mechanisms for access check, launch permission check, and data protection require some mechanism for determining the security identity of the client. This client authentication is performed by one of the security providers, which returns unique session tokens that are used for ongoing authentication once the initial connection has been established. The initial authentication often requires multiple round trips between caller and object. The Windows NT LAN Manager (NTLM) security provider for example authenticates by challenging the caller: the security provider knows the password of the user (more precisely an MD4 hash of the password). It encrypts a randomly generated block of data using the MD4 hash of the password and sends it back to the client (the challenge). The client then decrypts the data block and returns it to the server. If the client also knows the correct password, the decryption will be successful as the server knows that the client is "authentic." The NTLM security provider then generates a unique access token, which it returns to the client for future use. For future authentication, the client can simply pass in the token, and the NTLM security provider would not perform the extra round trips for the "challenge/response" protocol.

DCOM uses the access token to speed up security checks on calls. However, to avoid the additional overhead of passing the access token on each and every call, DCOM by default only requires authentication when the initial connection between two machines is established (RPC_C_AUTHN_LEVEL_CONNECT). It then caches the access token on the server side and uses it automatically whenever it detects a call from the same client.

For many applications this level of authentication is a good compromise between performance and security. However, some applications may require additional authentication on every call. Often, certain methods in an object are more sensitive than others are. An online shopping mall might only require authentication on connection establishment as long as the client is only calling methods for "browsing" the shopping mall. But when the client actually orders merchandise and passes in credit card information or other sensitive information, the object might require calls to be individually authenticated.

Depending on the transport used and the size of the method data to be transmitted, a method invocation can actually require multiple data packets on the network. DCOM lets applications choose whether only the first packet of a method invocation contains authentication information (RPC_C_AUTHN_LEVEL_CALL) or if each packet should be individually authenticated (RPC_C_AUTHN_LEVEL_PKT).

As discussed in the previous section, authentication, integrity, and privacy are tightly related. For this reason DCOM defines a single sets of constants that convey the level of authentication and privacy. These constants are the same constants as defined for DCE RPC and are listed in Table 4.

Table 4

Authentication Level	RPC Authentication Constant	Description
None	RPC_C_AUTH_LEVEL_NONE	Performs no authentication.
Connect	RPC_C_AUTHN_LEVEL_CONNECT	Authenticates only when the client

		establishes a relationship with the server. Datagram transports always use packet authentication (RPC_C_AUTHN_LEVEL_PKT) instead.
Call	RPC_C_AUTHN_LEVEL_CALL	Authenticates only at the beginning of each remote procedure call when the server receives the request. Datagram transports always use packet authentication (RPC_C_AUTHN_LEVEL_PKT) instead.
Default	RPC_C_AUTHN_LEVEL_DEFAULT	In the current implementation of DCOM this setting always maps to RPC_C_AUTHN_LEVEL_CONNECT
Packet	RPC_C_AUTHN_LEVEL_PKT	Authenticates that all data received is from the expected client.
Packet Integrity	RPC_C_AUTHN_LEVEL_PKT_INTEGRITY	Authenticates and verifies that none of the data transferred between the client and the server has been modified.
Packet Privacy	RPC_C_AUTHN_LEVEL_PKT_PRIVACY	Authenticates all previous levels and encrypts the argument values of each remote procedure call.

Since DCOM enforces the authentication policy on behalf of the object, the object needs to indicate the authentication level it is willing to accept. This can be done either programmatically or via external configuration.

Protecting the caller: Impersonation levels

A more subtle implication of security in distributed applications is the issue of protecting callers from malicious objects. For Internet applications in particular, this is a critical concern. Since DCOM allows objects to impersonate callers, objects can actually perform operations they do not have sufficient privileges to perform alone. To prevent malicious objects from using the caller's credentials, the caller can indicate what it wants to allow objects to do with the security token it obtains. The following options are currently defined:

- Anonymous (**SecurityAnonymous**): the object is not allowed to obtain the identity of the caller. This is the safest setting for the client but the least powerful for the object.
- Identify (**SecurityIdentification**): the object is only able to detect the security identity of the caller (that is, the user name), but can not impersonate the caller. This call is still safe for the client in that the object will not be able to perform operations using the security credentials of the caller. However, the client's user name will be disclosed to the object.
- Impersonate (**SecurityImpersonation**): the object can impersonate and perform local operations, but it can not call other objects on behalf of the caller. This mode is potentially unsecure for the caller, since it allows the object to use the client's security credential to perform arbitrary operations on the machine where the object is running.
- Delegate (**SecurityDelegation**): the object can impersonate the caller and, in addition, it can perform other method invocations using the security identity of the caller. In this mode, the caller essentially delegates ownership of its security identity to the object so that the object can perform arbitrary—including remote—operations using the caller's security identity. As of Windows NT 4.0, no security provider supports delegation.

These options are defined as part of the Windows NT security infrastructure (SECURITY_IMPERSONATION_LEVEL enumeration).

Again, DCOM allows these settings to be both programmatically controlled and externally configured.

Fundamentals: Windows NT security infrastructure

As described previously, DCOM ties into the extensible Windows NT security infrastructure. This section describes some aspects of the Windows NT security infrastructure.

Whenever DCOM needs to know the identity of a caller, it asks the security provider to authenticate the caller. The security provider returns an access token, which contains a security identifier (SID) that uniquely identifies the authenticated security principal.

DCOM then uses the security APIs to find the caller's security token in a list of security principals. This list of security principals is called a Discretionary Access Control List (DACL). Access control lists consist of multiple Access Control Entries (ACE), which correspond to individual security principals. Each access control entry can indicate that the corresponding security principal is to be allowed access or that it is to be denied access. The security APIs traverse the list of access control entries in the access control list. If a security principal matches an access-denying ACE, the caller is assumed not to have access. If a security principal matches an access-allowing entry, the caller is assumed to have access. Access-denying entries are usually placed before access-allowing entries.

The DACL is combined with another SID that indicates the owner of the list, and with another list of ACEs. This other list of ACEs is called the System Access Control List (SACL). Its purpose is to indicate whether access to the list should trigger an audit event. The current implementations of DCOM do not implement auditing and simply ignore the SACL.

The combination of owner (owning SID), list of principals (DACL), and list of principals to audit (SACL), is called a security descriptor (SD): a security descriptor completely describes the security policy in terms of object owner, access, and auditing.

The security descriptor (in its self-relative form) is a memory structure that references its elements (DACL, SACL, and so on) using offsets instead of pointers. This self-relative security descriptor can thus simply be written into a registry key and safely retrieved.

Sample: Enumerating a security descriptor

The following sample program illustrates how the security descriptor for default access permissions can be read from the registry and how the security principals in the DACL can be enumerated:

```
#include "windows.h"
#include "stdio.h"
main()
{
    // Open registry key.
    HKEY hKeyOLE=NULL;
    RegOpenKeyEx(HKEY_LOCAL_MACHINE, "SOFTWARE\\Microsoft\\Ole",  0, KEY_READ, &hKeyOLE);

    // Read security descriptor length.
    DWORD dwSDSize=0;
    RegQueryValueEx(hKeyOLE, "DefaultAccessPermission",  NULL, NULL, NULL, &dwSDSize);
    // Read security descriptor data.
    SECURITY_DESCRIPTOR* pSD = (SECURITY_DESCRIPTOR*) CoTaskMemAlloc(dwSDSize);
    RegQueryValueEx(hKeyOLE, "DefaultAccessPermission",  NULL, NULL, (LPBYTE) pSD, &dwSDSize);
    RegCloseKey(hKeyOLE);
    // Get the DACL.
    BOOL bHasDacl, bDefaulted;
    ACL* pDACL=NULL;
    GetSecurityDescriptorDacl(pSD, &bHasDacl, &pDACL, &bDefaulted);
    if (pDACL)
    {
        // Enumerate the ACEs in the DACL.
        ACE_HEADER* pAce=NULL;
        for (int i=0; i<pDACL->AceCount; i++)
        {
            // Retrieve ACE i.
            GetAce(pDACL, i, (void**) &pAce);
            // Test for AceType.
            switch (pAce->AceType)
            {
                case ACCESS_ALLOWED_ACE_TYPE:
                case ACCESS_DENIED_ACE_TYPE:
                {
                    // Find the account name given the SID in the ACE.
                    TCHAR szUser[_MAX_PATH];
                    TCHAR szDomain[_MAX_PATH];
                    DWORD dwUserSize=sizeof(szUser);
                    DWORD dwDomainSize=sizeof(szDomain);
                    SID_NAME_USE use;
                    LookupAccountSid(
                        NULL,
                        &(((ACCESS_ALLOWED_ACE*)pAce)->SidStart), // ACCESS_DENIED_ACE is polymo
                        szUser,
                        &dwUserSize,
                        szDomain,
                        &dwDomainSize,
                        &use);
                    // Print the principal information.
                    if (pAce->AceType==ACCESS_ALLOWED_ACE_TYPE)
                        printf("Allow");
                    else
                        printf("Deny");
                    printf(" access to principal %s\\%s\n", szDomain, szUser);
                    break;
                }
                // Only allow and deny ACEs are allowed in DACLs!
                case SYSTEM_AUDIT_ACE_TYPE:
                    printf("Audit ACE's not allowed in DACL!\n");
                    break;
                case SYSTEM_ALARM_ACE_TYPE:
                    printf("Alarm ACE's not allowed in DACL!\n");
                    break;
                default:
                    printf("Unknown ACE type in DACL\n");
                    break;
            }
        }
    }
    CoTaskMemFree(pSD); // Free the security descriptor.
    return 0;
}
```

Designing for Security

DCOM provides multiple choices to secure applications. On one end of the spectrum, DCOM can enforce security without any cooperation on behalf of the object or the object's caller— the security settings for an object can be externally configured and DCOM enforces them automatically. On the other end of the spectrum, DCOM exposes its entire security infrastructure to the developer so that both clients and objects can obtain complete programmatic control over their security policies.

Application designers can choose whichever mechanism is most appropriate for the specific application they are designing. Both approaches have specific advantages. Keeping the security policy externally configurable provides more flexibility at deployment time, because the same binary component can be used in different applications or in different environments that require different security policies. However, the security configuration needs to be provided at deployment time and has to be correct for the application to work properly. Programmatically controlling security policies provides additional flexibility to the developer, but hard-codes certain security decisions into the components or their clients.

Configuring security settings: Per-process

DCOM provides mechanisms to externally configure security settings for objects and clients. In the current implementations of DCOM all security policies are enforced at the process level. All objects in a process share the same security policies, unless they programmatically override them (see below for details on programmatic security). To match this process-wide security configuration, DCOM introduces the concept of an AppID.

AppIDs group the configuration options for one or more DCOM objects into one centralized location in the registry. COM objects hosted by the same executable must map into the same AppID. In-process COM objects that are run in a surrogate process can be forced into the same process by assigning the same AppID to their CLSID entries.

AppIDs are 128-bit GUIDs. Individual classes are mapped onto their AppIDs by adding a named value, "AppID" under their CLSID key in the registry:

```
[HKEY_CLASSES_ROOT\CLSID\{<clsid>}]
    "AppID" = "{<appid>}"
```

A named-value "AppID" of type REG_SZ contains the string representation of the AppID. This mapping is used during activation to obtain the default launch permissions.

The actual configurations for an AppID are stored in a newly created registry key hierarchy under:

```
[HKEY_CLASSES_ROOT\AppID\{<AppID>}]
```

Applications or their setup programs can directly modify registry keys under the AppID key. Administrators can use the DCOM Configuration utility (DCOMCNFG.EXE) to manually configure default security settings.

Access security

DCOM enforces access security on every method call on an object. If the process does not programmatically override security settings (see below), DCOM reads a security descriptor from the registry. When a call arrives, DCOM authenticates the caller (using whatever security provider is configured) and obtains an access token. It then performs an access check on the security descriptor with the access token.

The content of the value in the registry is a simple copy of the in-memory representation of a self-relative security descriptor:

```
[HKEY_CLASSES_ROOT \AppId\{<Appid>}]
    "AccessPermission" = hex: [self-relative security descriptor]
```

Launch security

Launch security is enforced whenever a new process needs to be created as part of object activation. DCOM finds the AppID corresponding to the activation request and reads a security descriptor from the registry. It then authenticates the activator and checks the activator's access token against the security descriptor.

The launch security settings are stored in the following registry key:

```
[HKEY_CLASSES_ROOT \AppId\{<Appid>}]
    "LaunchPermission" = hex: [self-relative security descriptor]
```

Identity

DCOM chooses the identity of an object at the time it launches the process containing the object. It determines, again, the AppID of the process being created and obtains the identity setting from the registry. Three choices are available:

- **Run as Activator.** The process is created in a new Window Station under the credentials of the caller. Although this option is the default, it is not typically recommended. If multiple users create the same object on a machine, independent Window Stations and processes are created for each user. Window Stations consume a significant amount of resources and their number is limited on current versions of Windows NT.
- **Run as Interactive User.** The process is created in the Window Station of the locally logged on user. If no user is logged on, the object creation fails. This option is useful for interactive distributed applications as well as during debugging or troubleshooting.
- **Run as a fixed user account.** The process is created in a non-interactive Window Station corresponding to a specific user account. If a non-interactive Window Station for the account exists, DCOM reuses it. In order to obtain security credentials, DCOM needs the current password for the user account. This password is stored separately in a secured section of the registry.

The security identity is indicated using the following registry entries:

```
[HKEY_CLASSES_ROOT \AppId\{<Appid>}]
    "RunAs" = "InteractiveUser"
[HKEY_CLASSES_ROOT \AppId\{<Appid>}]
    "RunAs" = "mydomain\myaccount"
```

Absence of the RunAs value indicates Run as Activator.

> **Note** COM objects running as Windows NT services (using the LocalService) do not obtain their security identity from the RunAs value. Instead their identity is configured via the normal Windows NT service configuration in the **Control Panel**.

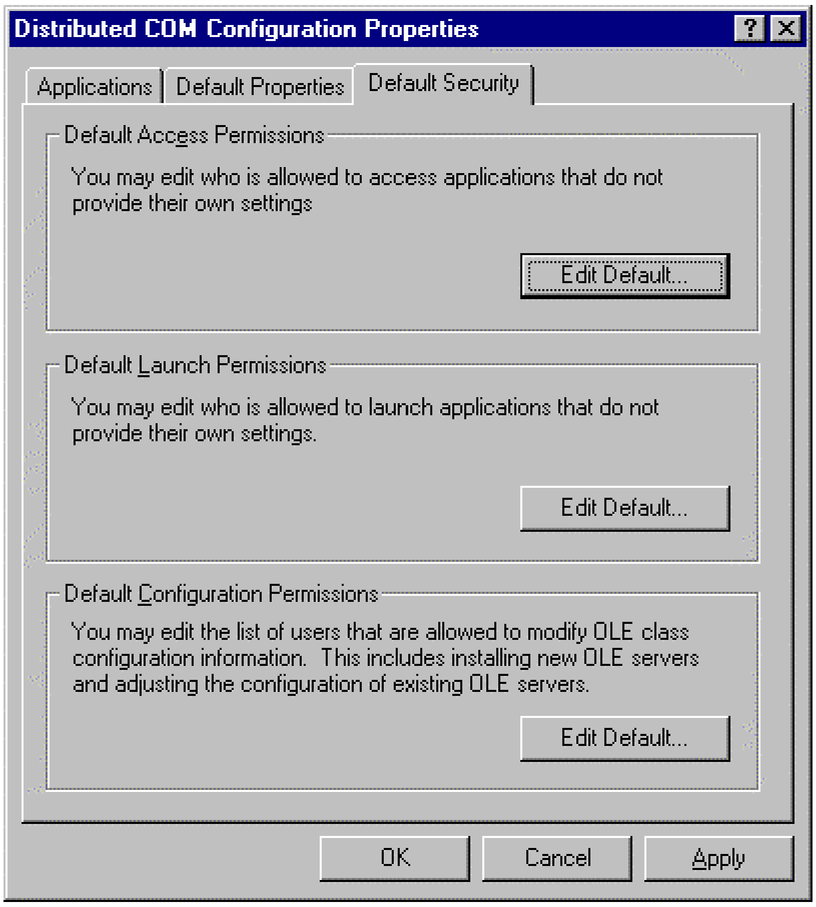
The security identity is typically configured externally, although it can be programmatically changed using standard Win32 process and thread APIs.

Machine-wide security defaults

If an object is not explicitly configured, DCOM applies machine-wide default settings for all security parameters. These defaults are stored in the registry as follows:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Ole]
   "EnableDCOM"="Y"
   "DefaultAccessPermission" = hex: [self-relative security descriptor]
   "DefaultLaunchPermission" = hex: [self-relative security descriptor]
   "LegacyImpersonationLevel" = dword:2
   "LegacyAuthenticationLevel" = dword:2
```

The DCOM configuration utility provides easy access to the registry configurations.


Distributed COM Configuration Properties

Applications | Default Properties | Default Security

Default Access Permissions

You may edit who is allowed to access applications that do not provide their own settings

Edit Default...

Default Launch Permissions

You may edit who is allowed to launch applications that do not provide their own settings.

Edit Default...

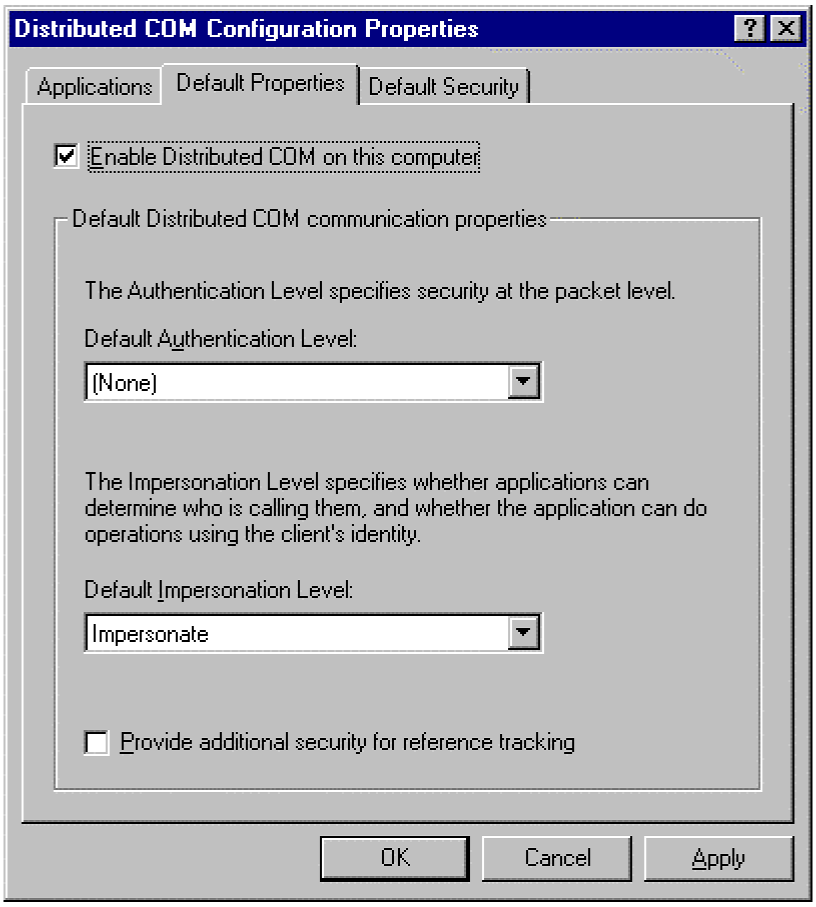
Default Configuration Permissions

You may edit the list of users that are allowed to modify OLE class configuration information. This includes installing new OLE servers and adjusting the configuration of existing OLE servers.

Edit Default...

OK | Cancel | Apply

Figure 13


Distributed COM Configuration Properties

Applications | Default Properties | Default Security

☑ Enable Distributed COM on this computer

Default Distributed COM communication properties

The Authentication Level specifies security at the packet level.

Default Authentication Level: (None)

The Impersonation Level specifies whether applications can determine who is calling them, and whether the application can do operations using the client's identity.

Default Impersonation Level: Impersonate

☐ Provide additional security for reference tracking

Figure 14

Launch security

Launch security controls the classes a client is allowed to launch and retrieve objects from.

By definition, launch security needs to be enforced by the COM libraries themselves, because the object that could potentially perform this check has not been instantiated yet! For this reason, launch security can only be externally configured and can not be controlled programmatically.

Launch security is automatically applied by the SCM of a particular machine. Upon receipt of a request from a remote client to activate an object, the SCM of the machine checks the request against the HKEY_CLASSES_ROOT\AppID\{Ã‚Â…}\LaunchPermission key which contains data describing the ACL. If the user's ACE is not contained within the ACL, access is denied. If the AppID does not have a LaunchPermission key, then the Service Control Manager checks the request against the ACL in the DefaultLaunchPermission key under HKEY_LOCAL_MACHINE\Software\Microsoft\OLE.

Launch security affords administrators granular control over who can instantiate DCOM objects for use by others and, therefore, provides protection of business processes (as an example) from malicious or accidental initialization. For example, you may want to restrict instantiation of accounts payable processes to employees contained in the accounting group; or you may wish to develop a DCOM server as a Windows NT service that can only be started by a specific service account.

By utilizing the DCOM configuration utility included with Windows NT (DCOMCNFG.EXE), an administrator can control activation security (launch security) at both the machine level and at the object level. In addition to launch security, an administrator can control who can access a particular object (access security) and who can alter registry settings related to a particular object (configuration access).

If an account may both launch and access an object, the account must appear on both the launch and access security access lists. DCOMCNFG does not enforce this constraint.

Programmatic security

A process must initialize the DCOM security layer by calling **CoInitializeSecurity** to establish the default authentication and impersonation level. DCOM servers must call this API prior to registering their class objects (**CoRegisterClassObject**) if they want nondefault authenticated connections. Clients must call this API prior to receiving their first interface pointer or giving out their first interface pointer. By using **CoInitializeSecurity**, and thereby establishing default call security for the process, the client avoids having to use **IClientSecurity** on individual proxies. Implementations of **QueryInterface** must never perform ACE verification because COM caches interface pointers and will not call **QueryInterface** on the server every time a client does a query.

Regarding DCOM, there are multiple implementations of security. Calls between processes rely on the Windows NT kernel. Since the call goes through the kernel, no one can fake where a call originates and no one can read the information. In fact, when you are making calls on the same machine the information is at encrypt level. Calls between machines rely on the Windows NT LAN Security Service Provider (NTLMSSP) for security. Calls between objects in the same apartment are direct function calls. Thus, security is not purely location transparent.

Another security service provider (SSP) that future versions of Windows 2000 will support is Kerberos security through the concept of proxy tickets. Consider the following scenario. Process A calls an application B, which impersonates A (**CoImpersonateClient**, **RpcImpersonate**, or **IServerSecurity::ImpersonateClient**). Now B is acting as A in certain ways while running as/in B. If B calls another application C while acting as A, C will impersonate B, not A, since the security privileges of A are not "delegated" to C. True delegation means that if the acting A thread calls another application, C, that C can impersonate A. The Kerberos security service provider will address this need.

Access security

COM provides two mechanisms to secure calls. The first mechanism COM provides is APIs and interfaces that applications may use to perform their own security checking. The second mechanism is done automatically by the COM infrastructure. An important item to note is that automatic mechanism does security checking for the process, not for individual objects or methods. Applications requiring more fine-grained security may perform their own security checking. Furthermore, the two mechanisms are not exclusive; an application may ask COM to perform automatic security checking and also perform its own.

In a typical scenario, the client queries an existing object for **IClientSecurity**, which is implemented locally by the interface 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 that 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** allows a server to register automatic authentication services for the process. Registering authentication services with **CoRegisterAuthenticationServices** does not prevent calls from arriving without an authentication service or with an unregistered authentication service.

Implementations of **QueryInterace** must never perform ACL verification. COM requires that an object that supports a particular IID always return success when queried for that IID. Aside from that requirement, verification of an ACL 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, COM 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.

IClientSecurity interface

IClientSecurity gives the client control over the call-security of individual interfaces on a remote object. Since all proxies generated by the COM MIDL compiler support the **IClientSecurity** interface automatically, a **QueryInterface** for **IClientSecurity** will only fail if:

1. The object is implemented in-process.
2. The object is remoted by a custom marshaler that does not implement the IClientSecurity interface and therefore does not support security.

Each proxy to an object offers a distinct **IClientSecurity** interface that controls security to the specific proxy. Therefore, calling **IClientSecurity** on the proxy to one object and then passing the secured proxy to another object in a different process or on a different machine will not convey the security settings of the proxy.

An example illustrating a specific use of the **IClientSecurity** interface would be to ensure the integrity and privacy of a credit card transaction. You could use the **IClientSecurity** interface to escalate the RPC authentication level to include packet encryption (RPC_C_AUTHN_LEVEL_PKT_PRIVACY).

IServerSecurity interface

The **IServerSecurity** interface is used by a server to help identify the client and is also used to impersonate the client during a call. By calling **IServerSecurity::ImpersonateClient**, the server can impersonate a client for the duration of the call. One should note that **CoImpersonateClient** is a helper function that calls **CoCallGetContext** in order to retrieve an **IServerSecurity** interface pointer and then calls **IServerSecurity::ImpersonateClient**.

```
//--------------------------------------------------------------------
//
//  Function:   CoImpersonateClient
//
//  Synopsis:   Get the server security for the current call and ask it
//              to do an impersonation.
//
//--------------------------------------------------------------------
WINOLEAPI CoImpersonateClient()
{
    HRESULT          hr;
    IServerSecurity *pSS;
    // Get the IServerSecurity.
    hr = CoGetCallContext( IID_IServerSecurity, (void **) &pSS );
    if (FAILED(hr))
        return hr;
    // Ask IServerSecurity to do the impersonate.
    hr = pSS->ImpersonateClient();
    pSS->Release();
    return hr;
}
```

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 client's name and perform ACL verification; 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 until the end of the current method call. At that time **IServerSecurity::RevertToSelf** will automatically be called if necessary.

Traditionally, in Win32, the last call to any impersonation mechanism overrode any previous impersonation. In the apartment model, however, impersonation information is maintained during nested calls. Consider the following case assuming that each thread is executing in a different process. Thread A invokes a method on a DCOM object living in thread B; thread B impersonates A. B impersonating as A then calls out to a DCOM object living in thread C; thread C impersonates B (which is currently acting in the security context of A), but instead of thread C having the security context of A, it impersonates the security context of B. Why? Prior to method invocation, the security context of the current thread is "suspended" and the security context of the current process is assumed. In true delegation, thread C would have the security context of A. The following diagram further illustrates that thread C (which is currently under the security context of B) calls a method in thread B. B then impersonates thread C and obtains the security context of C. All method invocations then return. Note that upon return of the method invocations, the security context prior to invocation is "resumed." There are two important points to make here. First, all client impersonations last only for the duration of the method invocation. Second, if a method does not call **IServerSecurity::RevertToSelf** or an equivalent prior to returning to the caller, the DCOM infrastructure will automatically handle the task for you. This example is illustrated in Figure 15.

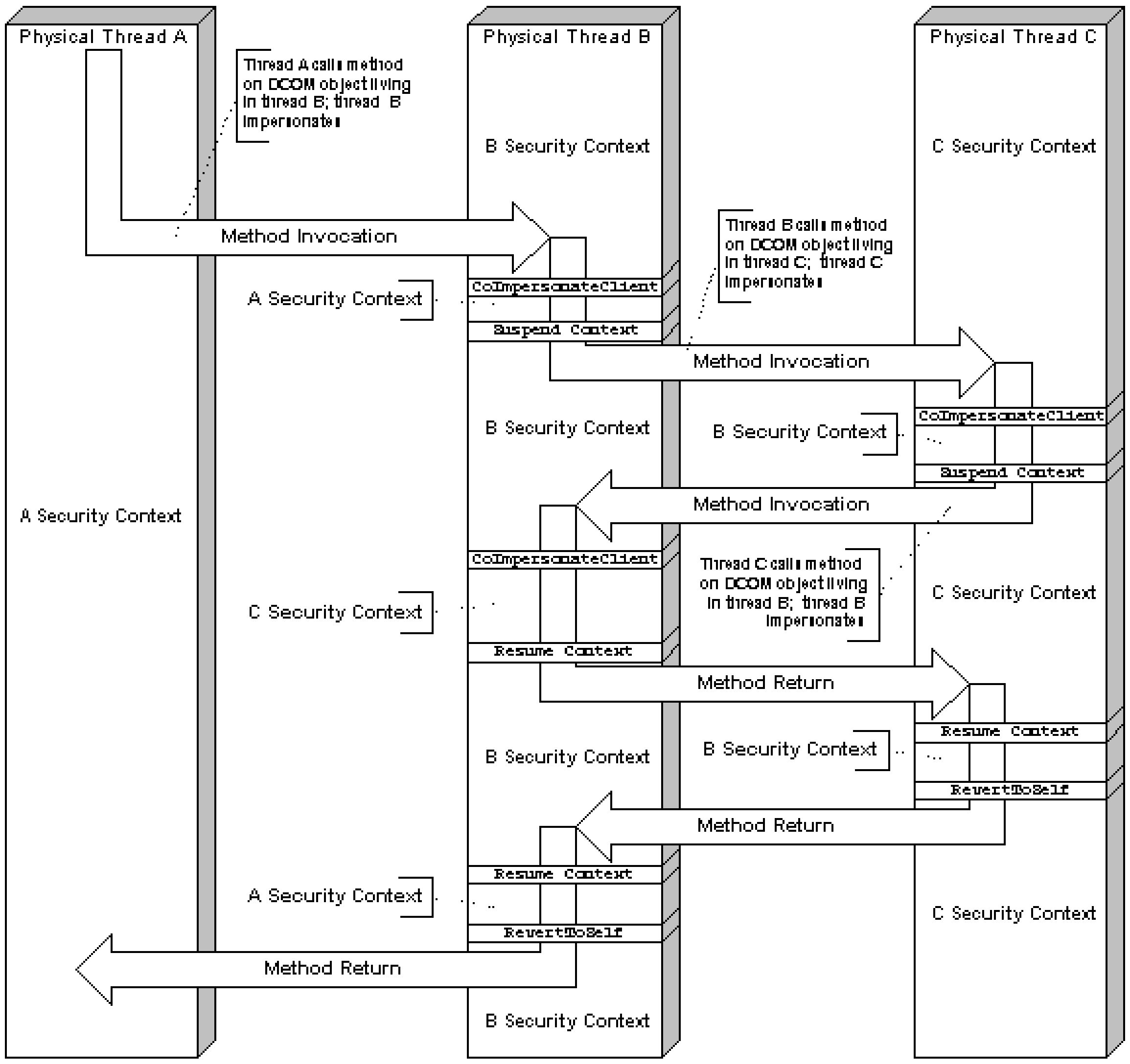


Figure 15

Return to Contents

Object RPC (ORPC)

The DCOM protocol, known as Object RPC (ORPC), is a set of definitions that extend the standard DCE RPC protocol. It has been designed specifically for the DCOM object-oriented environment, and specifies how calls are made across the network and how references to objects are represented and maintained. The ORPC protocol has been submitted as an Internet Draft to the Internet Engineering Task Force (IETF), as it is suited to both Internet and intranet component communication.

At the wire level, ORPC uses standard RPC packets, with additional DCOM-specific information—in the form of an Interface Pointer Identifier (IPID), versioning information, and extensibility information—conveyed as additional parameters on calls and replies. The IPID is used to identify a specific interface on a specific object on a server machine where the call will be processed. The marshaled data on an ORPC packet is stored in standard Network Data Representation (NDR) format, so that issues of byte order and floating point formats are automatically handled. DCOM uses one new NDR type, which represents a marshaled interface. DCOM clients machines are also responsible for periodically ensuring that objects are kept alive on server machines by 'pinging' between machines in the background, a process that has been optimized to reduce unnecessary pinging and minimize network traffic (see "Pinging" in the section "Connection Management").

Programmers for the most part do not have to work at the ORPC level. The Microsoft Interface Definition Language (MIDL) compiler can be used to automatically generate the code that is needed to transfer the data across the network, based simply on an IDL file. Strictly speaking, MIDL is not part of DCOM, and any tool can be used to generate marshaling code, but it is convenient to use MIDL, with its C-like semantics.

As part of the migration to DCOM, IDL has been extended to include the functionality found in the Microsoft Object Definition Language (ODL) and, as a result, the MKTYPLIB utility is no longer needed, its functionality having been subsumed into version 3.0 of MIDL. The MIDL compiler can take an IDL specification and generate the C++ code needed to transfer, or marshal, the information across the network. Marshaling is only required where the client is calling a server that exists in another address space or on another machine. See the section "Packaging Parameters and Objects: Marshaling" for more information on marshaling.

MIDL was a key part of pre-DCOM and, in most cases, the standard proxy and stub marshaling code generated by MIDL are all that is needed to ensure that DCOM can communicate with a remote object. However, there are situations in a remoted environment when an object may wish to use its own form of custom marshaling, perhaps to optimize performance, as discussed previously.

Return to Contents

References

- The Component Object Model Specification (MSDN Library, Specifications)
- DCOM Business Overview
- DCOM Technical Overview
- DCOM Solutions in Action (See the "DCOM and Dev Issues" section of the PDC 97 Conference Papers in the MSDN Library)
- Distributed Component Object Model Protocol—DCOM/1.0 (MSDN Library, Specifications)

Return to Contents