Component Technologies from Microsoft

1. COM
2. DCOM
3. COM+

Component Object Model(COM)

Component Object Model (**COM**) defines a binary standard for components to interact with each other through their interface definitions. COM defines both how the object exposes itself and how this exposure work across processes and across networks. Interface based approach allows to ensure dynamic interoperability of binary objects written in different programming languages. Object lifecycle support in COM loads the server objects on demand if they are not loaded already and releases them whenever it finds out there is no user for this object.

COM supports Interface based programming framework where object exposes its functionality through interface. An object can support multiple interfaces. An interface is a pointer to a block of memory in which the first word is a pointer to a table of function addresses(virtual function table or v-table) as in C++ virtual function table. A client uses the interface only through its v-table since the object may actually be nothing more than a proxy which forwards the method calls to actual object.

All COM components provide *IUnknown* **base interface** which has the following methods in the first three entries of the v-table.

QueryInterface	:	returns pointer of given interface (by IID) if the object supports this interface.
AddRef	:	increments the object's reference count.
Release	:	decrements the object's reference count, freeing the object if the count becomes zero.

AddRef() and **Release()** methods are directly related to object lifecycle support of COM. Whenever client gets the interface pointer of the server object, the client counter for object is incremented so that system can track down the number of concurrent users of this particular object. Clients invoke **Release()** method on interface pointers when they do not need it anymore. It has the same functionality as in **destructor** method of an object in C++. Invoking **Release()** method on the interface decrements the client counter for this object in the COM internals and COM removes this object from runtime as soon this count drops zero(no client). When object supports multiple interfaces, client needs to get the interface pointer which is required. **QueryInterface()** method provides the neccessary navigational functionality to obtain the required interface.

Any given function is identified with three elements:

1. the object's class indentification number (CLSID),

2. the interface type(Interface Identification Number-IID) through which the client will call member functions,

3. the offset of the particular member function in the interface.

CLSID and IID are unique 16-byte values and are generated by Globally Unique Identifier generators(such as GUIDGEN). These generators use the current time and the IP address of the machine to generate the number. This technique is defined in OSF's DEC specifications as Universally Unique Idenfier(UUID).

An instance of an COM object is created with *CoCreateInstance()* method by providing CLSID. COM keeps a registry for each CLSID and returns the necessary object's pointer. It starts the object if the object(library-EXE or DLL) is not loaded already. If the server object implemented in DLL format, then this object is defined in the client process' memory(LoadLibrary). If the server object implemented in EXE format, then this object runs in a different process(CreateProcess).

COM supports *Location Transparency* so that an object can be in the client's address space, or in another container. If the server object is in another machine(its entry in the COM registry reveals this), then COM starts a proxy server and this talks to Service Control Manager(SCM) of the other machine and starts the necessary object on that machine and returns the end-point address for communication with the object on the remote machine. Client receives the pointer of the proxy object running on the local machine. COM places its own remote procedure call(RPC) infrastructure code into the *vtable* and then packages each method call into a standard buffer representation(Proxy), which it sends to the component's side, unpacks it, and reissues the original method call on a server object(Stub). While COM internals struggle with these processes, developer never sees a difference in the client/server code.

Versioning in COM is provided by defining additional interfaces for the new server object. Clients has to take care of controlling (*QueryInterface()*) whether server supports particular interfaces or not as objects can support multiple interfaces.

Microsoft recommends developers to use *aggregation* for components which are quite substantial and composed of a lot of interfaces. Aggregation means that an object contains several interfaces and implements them through several objects. This has several advantages in terms of performance and memory usage:

1. If the server object has a lot of services, it can be divided into sub-interfaces and each interface can be handled by a different object. Then a developer defines a wrapper object which contains all interfaces. It takes less time and less memory when COM starts the wrapper object. Whenever one interface is asked, the related object is being brought to memory instead of the whole component.

2. This type of aggregation makes it possible to distribute subcomponents onto different machines so that load sharing can be achieved with no cost.

COM supports persistent object stores for the components which provides **IPersistStorage** interface which defines a protocol to restore/store objects.

Dynamic Interface Invocation

Automation focuses on run-time type checking at the expense of speed and compile-time type checking. Automation replaces code generated by the compiler with code written by developer. Automation Server is a COM component that implements the IDispatch interface. Automation Controller is a COM client that communicates with the Automation Server through its IDispatch interface. Automation Controller uses member functions in the IDispatch interface to indirectly call functions in the Automation server. Automation controller does not need the interface definition.

IDispatch interface supports the following functions:

- GetIDsOfNames : accepts the name of a function and returns its dispatch ID(DISPID) which identifies a function.

- Invoke() : DISPID informs to invoke which function.

- GetTypeInfoCount() : Tells whether type information is available. This function can be used to determine whether to use early-binding or late-binding.

- GetTypeInfo() :

The set of functions implemented by an IDispatch::Invoke() implementation is called a dispatch interface(dispinterface). The implementation part requires to implement a table for function name-to-DISPID and DISPID-to-Function pointer transformations. For small number of functions, a vector can be used while for large number of functions hashtable will be preferable.

Using Dynamic invocation utility degrades the performance of function call for C++ users since they rely on vtbl entries. To overcome this problem, Dual interface concept is proposed. Dual Interface

IUnknown -> Foo will be IUnknown -> IDispatch -> Foo.

this convention allows clients to access Foo's methods through vtbl or through Invoke() interface, therefore they are called dual interfaces. Dual interfaces are preferred way of implementation strategy since it allows C++ clients directly use functions through vbtl while automation clients use dispinterface.

This functionality makes Visual Basic Developer's life very easy while C++ Developer's life turns out to be a suffering. For a C++ Programmer Client:

1. needs to get ProgID

2. gets CLSID by using ProgID(::CLSIDFromProgID())

3. gets a pointer to IDispatch interface(::CoCreateInstance())

4. gets DISPID of function(GetIDsOfNames())

5. prepares parameters in DISPPARAMS structure

6. executes Invoke() method

7. handles return values, exceptions, and possible error codes.

The types of parameters that Invoke() can pass to a function in a dispinterface are limited in number(VARIANTS).

Giving a which function, client wants to call is a problem because each name there are four possible cases:

1. a normal function(DISPATCH_METHOD)
2. a function to put a property (DISPATCH_PROPERTYPUT)
3. a function to put a property by reference (DISPATCH_PROPERTYPUTREF)
4. a function to get a property (DISPATCH_PROPERTYGET)

These functions come from the MIDL definition for providing easy method definitions for accessor methods(get_XX,put_XXX). Therefore, Visual Basic User uses only names in the program and interpreter executes an appropriate method on an interface.

Dispatch parameter structure supports Name-value paired parameter list. For C++ programmer, name values makes no sense, providing parameter values is sufficient.

Results are obtained as a VARIANT array.

Exceptions are obtained as an EXCEPTINFO structure.

- IDispatch takes the name of a function and executes it.

VBScript and JScript clients require IDispatch interface of components. Visual Basic checks whether component supports dual interface or not. If it finds out that it does and there is a type library defining this interface, then it directly uses the custom interface(v-table) instead of the dynamic invocation.

Type Libraries

Type libraries provide type information about components interfaces, methods, properties, argumenmts and structures. A type library is compiled version(binary) of an IDL file that can be accessed programmatically. Automation library provides standard components for creating and reading this binary file. This feature allows Visual Basic clients to access functions through vtbl (which is faster and type safe).

Type libraries may contain help strings for all the components, interfaces, functions inside the library and this makes easier for object browsers (such as Visual Basic) to provide help for those components.

Type libraries(.TLP) can be build from IDL by using **library** keyword in the definition. IDL compiler produces a type library files for this components and all interfaces that are contained by this component.

Visual C++, Visual J++, and Visual Basic are capable of reading type libraries.

ICreateTypeLib-ICreateTypeInfo interfaces are building. ITypeLib and ITypeInfo interfaces are for reading.

Anything defined in the library section of IDL file is added to the type library and they are identified with LIBID (UUID for Type Library). library section also contains coclass declaration for defining interfaces in the type library.

LoadTypeLibExe registers the component’s type library.

- Implementation inheritance: a class inherits its code or implementation from its base class. (COM does not support)

- Interface inheritance: a class inherits the type or interfaces of its base class.

- COM does not support implementation inheritance because implementation inheritance binds one object tightly to the implementation of another object. If the implementation of a base object changes, the derived object breaks and must be changed.

- Experts on writing large applications in C++ strongly recommend building large applications using a foundation of abstract base classes.

- 32 bit HRESULT is used for error codes. Users can defined their return codes by using 16 bit but propagating the error code from inner components to outside is a very complicated task therefore, suggested practice is to use extra out parameters to propagate the error.

- Global Unique Identifier defines a 128-bit(16 byte) value wich is guaranteed to be unique without consulting a general authority. These IDDs are used to distinguish interfaces(IID), components(CLSID).

- WINDOWS registry keeps CLSID->DLL mappings. These mappings go to HKEY_CLASSES_ROOT\CLSID key.

CreateTypeLib2 function in COM library is called to create a type library and obtain a pointer to the ICreateTypeLib2 interface. ICreateTypeLib interface has the following functions:

- **SetGuid()** : sets the globally unique idenfier(GUID) of an element in a type library.
- **SetName()** : sets the name of type library.
- **SetVersion()** : sets the version numbers for the type description.
- **SetDocString()** : sets the help information about the type library.
- **CreateTypeInfo()** : creates a type information for the type library and returns a pointer to ICreateTypeInfo interface.

Containment and Aggregation

As a COM component developer, you might want to decompose your problem space and define several sub component which will cary out the some specific tasks instead of putting all the functionality into one component. There are two ways to obtain this: containment and aggregation.

In containment, a component keeps a pointer to other components and implements all their interfaces by directly calling the same functions or addition of new codes before/after the function call. Containment based approach never let outsider see the contained components.

Aggregation based specialization allows client to access the contained component and invoke the function call directly on them. Unfortunately, a simply returning the pointer of included component is not enough since this reference does not point the outer component's interface. Therefore, a component should be returning the outer components IUnknown interface if it is aggregated in.

For solution of this problem, each aggregatable interface provides two types of IUnknown interface one for delegating and one for non-delegating IUnknown interface. Non-delegating interface definition provides all the base IUnknown functions while delegating interface has an IUnknown pointer to some interface and it forwards all the calls to this interface. If the component is aggregated in another component then this pointer points the outer component's interface. If component is not aggregated in, then this pointer will point to non-delegating interface. Outer compoent makes all function calls on non-delegating interface. Note that the delegating interface should be extended from non-delegating interface also. This will guarantee that a client can not get the pointer to an aggregated component.

This requires two version of CoCreateInstance functions one for non-aggrageted and one for aggregated version.

DelegatingIUnknown interface keeps a pointer to the outer IUnknown interface. If component is not aggregated within another component, this pointer references the NondelegatingIUnknown interface so that it can work without any problem. DelegatingIUnknown interface forwards all the calls to this referenced interface. This will also require some changes in the CreateInstance function of IClassFactory interface and the constructor of the aggregated interface to take care of possible aggregated use.

Registry

Registry is a hierarchy of elements whose each element is called a key. A key can include a set of subkeys, a set of named values, and/or one unnamed value. Subkeys can have other subkeys and values.

COM uses HKEY_CLASSES_ROOT branch of Registry tree.

CLSID Key contains an externalized GUID of registered classes. Under this representation, it might have an InprocServer32 key with an appropriate DLL file name.

ProgID key is used to map a programmer friendly string to a CLSID so that users of these components can access components through ProgID values. The default value of ProgID denotes the program name, component name and version number. This should be a key in HKEY_CLASSES_ROOT tree, including CLSID of this component. ProgID might contain a key named VersionIndepenedentProgID whose value again a key in HKEY_CLASSES_ROOT tree.

HKEY_CLASSES_ROOT\Interface key contains information about each COM interface available on the system that has an interface identifier. For each interface, one IID subkey is required and this subkey might contain the following subkeys:

- **BaseInterface** : The interface from which the current interface is dericed.

- **NumMethods** : The number of methods in the associated interface.
- **ProxyStubClsid32** : The CLSID of the 32-bit proxy/stup DLL that knows how to marshal this interface.
- **TypeLib** : The LIBID associated with the type library in which the interface is described.

HKEY_CLASSES_ROOT\TypeLib defines each type library installed on the system. Every LIBID key in the registry contains several subkeys that specify the version number of the type library, the path to that library, several flags, and an optional path to a directory containing help files with information about the contents of the type library.

Active Template Library(ATL) ATL is composed of C++ template classes that provide support for the most common COM programming procedures as well as dealing with component registaration issues.

Communication between process

- Dynamic Data Exchange (DDE)
- Named pipes
- shared memory
- LPC is local procedure call based on OSF DCE RPC.
- DLL is in-process
- EXE is out-of-process
- EXE is local-server
- Proxy/Stub DLL
- MIDL
  - object : means that this interface is a COM interface.
  - uuid : specifies the IID for thise interface.
  - helpstring
  - pointer_default
- MIDL compiler produces necessary header and DLL files which implements Proxy/Stub pairs.
- Proxy/Stub DLLs should be registered. In the registry:
  HKEY_CLASSES_ROOT\Interface
  GUID of interface
  ProxyStubClsid32 contains the CLSID for the proxy/stub DLL. Note that this should be already registered at HKEY_CLASSES_ROOT\CLSID. In this registration, "InprocServer32" key points

to the location of DLL file related to this class.

- Serving component from an EXE is different from serving components from DLL.
- DEF file tells which function will be in DLL interface. LIBRARY keyword gives the name of DLL file and EXPORTS section lists the name of the functions and their function indices. Note that those functions should be decorated with "extern 'C' " if they are written in C++.
- LoadLibrary(WIN32) loads the DLL.
- DLLs share the same address space as the application they are linked to.
- CoCreateInstance takes CLSID, creates an instance of the corresponding component and returns an interface pointer for this instance of the component.
- Third parameter in the CoCreateInstance call specifies where the component resides:
  - CLSCTX_INPROC_SERVER :
    Component must be implemented in DLLs to run in the client's process.
  - CLSCTX_INPROC_HANDLER :
    An in-proc handler is an in-process component that implements only a part of a component. The other parts of the component are implemented by out-of-process component on a local or remote server.
  - CLSCTX_LOCAL_SERVER :
    Local servers are implemented as EXEs and they have different process that the client has.
  - CLSCTX_REMOTE_SERVER :
    On remote machine and needs DCOM to work.
- A class factory is a component to create other components. It supports IClassFactory interface.
- CoGetClassObject call returns the class factory object related to given CLSID.
- IClassFactory has two functions:
  - CreateInstance: It allows to return the given interface together.
- Advantages of using IClassFactory
  - When client needs to use a creation interface other than IClassFactory such as IClassFactory2(which is the new interface allowing licensing/permissions to IClassFactory interfaces.)
  - When client needs to create a bunch of components all at one time, it is faster than CoCreateInstance since CoCreateInstance obtains and releases the IClassFactory interface for each invocation.

- Class factory component is provided the developer of DLL.

- DllCanUnloadNow():
  is being called by COM to ask the DLL to free it or not. DLLs keep track of their supporting components so that if it drops to zero and COM invokes this method then DLL tells whether it can be removed from memory or not. Note that COM periodically runs CoFreeUnusedLibraries() during its idle time and this call actually invokes DllCanUnloadNow() method on each DLL.

- LockServer():
  If DLL has class factories and those objects might be in use so that LockServer interface allows client to the similar thing to force DLL not to leave the memory especially client needs to use the IClassFactory interface.

- DllRegisterServer():

- DllUnregisterServer():

- DllMain():

- The Service Control Manager (SCM)

  SCM tries to provide a server object when a client request is made. SCM is a part of COM library and responsible for locating server objects. Therefore, SCM keeps a database for class infromation which is extracted from the registry. When a client wishes to create an instance of class determined by it CLSID, the COM Library passes this request to the local SCM and expects that SCM will return a class factory object to the COM Library. SCM takes actions according to the type of registered object server:

  - **In-Process** : It returns the file path of .DLL. Then The COM library loads this DLl (LoadLibrary()) and gets the class factory().

  - **Local out-of-process server**: It starts the registered executable so that this server registers its class factory and the COM library can pick it up from the registry.

  - **Remote out-of-process server**: It contacts the SCM on another machine and forwards this request. The remote SCM starts a server, defines a connection to this class factory object, and returns an RPC connection corresponding the remote class factory. The local SCM creates a proxy for this remote object and returns this object to the COM Library. This proxy object forwards all calls to the real object running on another machine.

- Monikers

  Monikers alllow COM to name objects and activate them in the system, therefore it is also known as intelligent names. The requirement for Monikers becomes clear since CoGetClassObject and CoCreateInstance functions in the COM library is not enough to define a constructor with parameters. A custom activation interface can be defined for this object but Visual Basic client unfortunately can not work with this component. Monikers enable all objects to deal with naming through a single, standard interface (IMoniker). The COM library contains several default moniker implementation and functions to create them and all of them accepts additional arguments:

- **CreateFileMoniker** function creates a file moniker which wraps the path of a file.
- **CreateItemMoniker** function creates an item moniker which is used to identify an object contained in another object.
- **CreatePointerMoniker** function creates a pointer moniker which refers to an active or running object.
- **CreateAntiMoniker** function creates an anti-moniker which is the inverse of another moniker.
- **CreateGenericCompositeMoniker** function creates a composite moniker that can contain other monikers.
- **CreateClassMoniker** function creates a class moniker that is a wrapper for CLSID.
- **CreateURLMoniker** function creates a URL moniker that is a wrapper for Uniform Resource Locator (URL).

BindMoniker function in the COM Library takes an IMoniker interface pointer and binds that moniker to the object it names. This function defines a binding context(IBindCtx) by CreateBindCtx function, calls BinToObject function in IMoniker, and returns the pointer to the interface.

Since each moniker defines a relationship between the name and the object, this name can be displayed and called as display namee. For example, for file moniker, obviously the display name is the full pathname of the wrapped file. For the class moniker, the display name is the 'clsid:'. GetDisplayName function in IMoniker interface can be used to get the display name of a moniker. ParseDisplayName function in IMoniker interface allows to convert a display name to a moniker object.

MyParseDisplayName function in the COM Library converts a strring to a moniker that identifies the object named by given string. MkParseDisplayName function returns a corresponding file moniker object if the string contains a full path name of a file. If the string has 'ProgId:ObjectName' format, ProgId is a registered program identifier, then function gets the CLSID by using CLSIDFromProgID function. Then, function instantiates the moniker (CoGetClassObject), gets the IParseDisplayName interface pointer (QueryInterface), executes ParseDisplayName function so that a moniker identifies this name is defined.

GetObject function in Visual Basic and CoGetObject function in C++ handles this mess.

The class moniker is a system moniker defined in the COM Library. The class moniker wraps the CLSID of a COM class. The class monikers can be used with MkParseDisplayName to refer an object with this CLSID and activate it (BinTToObject). In the registry,
HKEY_CLASSES_ROOT\CLSID\clsid contains the class id of the class moniker object. Once a pointer to the class moniker is obtained, then a BindToObject call on this moniker returns the

required interface pointer. The class moniker offers Visual Basic clients an access to custom activation interface.

- Structured Storage

  Structured Storage service in COM enable an application to save data in a single file in a standardized, structured format. Structured storage format is a hierarchical format and can include some storages and each storage can contain substorages and streams(like files). IStorage and IStream interfaces are defined to support this service in the COM.

Joy of Reference Counting

COM relies on the reference counting technique for garbage collection. In this framework, COM component keeps track of active number of references to each interface and controls its lifecycle.

AddRef() and Release() functions in the base IUnknown interface implement necessary reference counts. COM component developer keeps a counter for each interface and increments this counter whenever AddRef() function is invoked and decrements this counter whenever Release() function is called. Also, in Release() function implementation, it checks whether the counter is dropped to zero or not. If it is zero, then it deallocates the instance from memory. For thread safety InterlockedIncrement() and InterlockedDecrement() functions should be used. This concept also puts several responsibility onto the client of this component. Therefore, there are some rule of thumbs which needs to be followed by the COM clients:

- Functions that return interfaces should always call AddRef on the pointer before returning from function. For example, an implementation of QueryInterface function should do this.

- When client is finished with an interface, client should call Release on that interface.

- Whenever client creates another reference to an interface, client should call AddRef.

In COM, reference counting should be done per interface since:

- easy debugging.

- support on-demand resources.

This complicated rules can be easily violated by the client during development phase. The price is that interfaces never being removed from memory and performance loss. Also, there are some cases that client actually do not need to call AddRef() and Release() pair for copy operations. When it is guaranteed that the life-cycle of the copied refence is nested in the life-cycle of the source of this copy operation.

There are also another set of rules when interface pointers are used as a function parameter. The following rules should be followed:

- Out parameter : function should execute AddRef() before returning such as QueryInterface() implementetation.

- In parameter : Since function can not change the pointer's value then there is no need to call AddRef and Release() functions on the pointer.

- In-Out parameter : Release() call should be made on pointer before assigning a new value to the pointer. AddRef() on this pointer should be called before returning from the function.

Sometimes even very careful developers can miss one of the cases and it might turn out to be a nightmare to debug it and find where this pair of calls are missed. Therefore, a smart pointer concept is proposed as a solution for this problem. A smart pointer is a class that overwrites **operator->** The smart pointer class contains a pointer to another object. Similarly, a smart interface pointer is a smart pointer that contains a pointer to an interface. Smart interface pointers are usefull especially dealing with reference counting problems in COM component development. This wrappers invokes the necessary AddRef() and Release() pair of calls for a client.

Note that figuring out whether client should call AddRef() on the interface pointer coming from a function is completely problematic especially the developer of this component did not stick to these common rules.

- CoBuildVersion() returns the major and the minor version number of COM library.

- CoInitialize() initializes COM Library. This function should be called before calling anything. Similary, when you are done with COM services, client should call CoUnitialize() function. This informs COM that it is time to free any resources related to previous operations.

- CoCreateInstance(
  CLSID requiredCLSID
  IUnknown* outerInterfacePtr
  IID requiredInterfaceID
  void** returnInterfacePtr)

  CLSID is a class idenfier(CLSID in GUID). COM searches HKEY_CLASSES_ROOT\CLSID tree in Registry to find out a match for the requested component.

- QueryInterface(
  IID requiredInterfaceID
  void** returnInterfacePtr)

- Interface pointer is actually a pointer to a pointer to a table of function pointers.

- Two IUnknown pointers can be comparable.

- LockServer(boolean) is provoided only for those clients that require special performance in creating multiple instances of their classes.

- In QueryInterface() of IClassFactory, we return the pointer of this object for IUnknown or IClassFactory request.

- CreateInstance(
  IUnknown *pOuterComponentPtr
  REFID riid,
  void ** returnInterfacePtr)
  First parameter sends an interface pointer to possible outer interface. Component decides whether it allows itself to be aggregared or not. If it does not and this parameter is not NULL, then it should return an error code(CLASS_E_NOAGGREGATION). Recall the second parameter in CoCreateInstance() function call. This parameter comes from this second parameter of CoCreateInstance() function.

- CoCreateInitialize() function first searches the registry and if it finds a valid registered implementation for this class then it loads this object and calls CreateInstance() method on this object to get the required interface from object. Therefore, COM requires that registered CLSID implementations have to support IClassFactory interface.

- EXE components require a WinMain function.

- DLL components require DllGetClassObject() and DllCanUnloadNow() functions.

- DllGetClassObject(
  REFCLSID requiredClassID
  REFID requiredInterfaceID
  void** returnedInterfacePtr)

  DLL checks whether it supports the required Class ID or not. If not, returns an error (CLASS_E_CLASSNOTAVAILABLE).

  Instantiates a class factory object. Aska a IClassFactory interface pointer.

- When custom activation is required such as licensing to restrict the component to be used on properly installed machine, IClassFactory interface is not sufficient. Then, this can be handled in DllGetClassObject() function by calling the proper functions on the interface before creating the IClassFactory object of the component.

- LoadLibrary : load a DLL into caller's address space.

- GetProcAddress : retrieves pointers to the DLL's exported functions

- FreeLibrary : It decrements the internal Windows reference counter for this DLL and unloads it if it drops to zero.

- DllGetClassObject is the first entry point to get the pointer to the IUnknown interface of a component.

- CoCreateInstance() calls LoadLibrary() and GetProcAddress() functions to load the DLL and get he address of exported functions from it.

- DllCanUnloadNow() function is to used to ask DLL implementation whether it is safe to unload this DLL from the memory. This function is being called by COM in CoFreeUnusedLibraries() function. This function is called by CoUnitialize() function which is called by the client before exiting.

- CoCreateInstance() calls CoGetClassObject(CLSID , void** returnedInterfacePtr). This returns the IclassFactory interface pointer of the component. Then on this interface pointer, it is safe to call CreateInstance() function to get the required interface pointer from component.

- CoGetClassObject function can be used directly by clients if there will be several instantiasion of component by the client. Because, CoCreateInstance gets the interface pointer of IClassFactory object, asks the necessary interface to it and releases IClassFactory object for each invocation. This definitely provides a performance improvement for cllient.

- The more detailed version:

  - LoadLibrary() to load DLL into memory

  - GetProcAddress() to get the DllGetClassObject function's address.

  - execute DllGetClassObject() function to obtain a pointer to IClassFactory interface.

  - CreateInstance() function call on IClassFactory interface pointer to obtain the poiniter to the required interface.

- CoGetClassObject() uses Service Control Manager(SCM)(pro:"scum") to locate and load the component.

- Self registering component does not require extra registration file because it supports this functionaliy within itself.

- For DLLs, they have to provide DllRegisterServer() and DllUnregisterServer() function so thta system can call necessary functions to register and unregister them respectively.

- For EXEs, it is responsibility of this process register itself according to command-line parameters(-RegisterServer/-UnregisterServer).

- Since COM is a binary standard, inheritence in source code level is not possible.

- Aggregation allows the internal object's interface to be exposed as interfaces of the external object.

- Aggregation works only among in-process components.(Why?)

- Outer component knows all aggregated interfaces and it delegates QueryInterface call to this aggregated interface.

- To make a component more accessible to developer's using other languages, type libraries are required. Fo instance, Visual Basic and Microsoft Visual J++ in general depend on type libraries.

-

Connectable Objects

Connectable objects allow server objects to talk back to its clients. An outgoing interface(source interface) is an interface defined in the object itself but implemented by the client. A sink object, residing within the client, is the object that implements the client's outgoing interface.

IConnectionPointContainer is the first interface pointer which allows client to obtain the pointer of IConnectionPoint interface. It has the following functions:

- **EnumConnectionPoints()** : enumerates all the outgoing interfaces supported by a connectable object.
- **FindConnectionPoint()** : takes the IID of the desired sink interface and returns a pointer to the relevant IConnectionPoint interface.
- **EnumConnectionPoints()** : retunrns a pointer to IEnumConnectionPoints interface from which client can obtain the information about supported interfaces.

IEnumConnectionPoints

IConnectionPoint interface has the following functions which allows client to query the IConnectionPointContainer interface in which this connection point is managed, the IID of sink interface which is supported by this connection point and information about current connections.

- **GetConnectionInterface**() : returns the IID of the outgoing interface managed by the connection point.

- **GetConnectionPointContainer()** : returns a pointer to the IConnectionPointContainer interface associated with the connection point.

- **EnumConnections()** : returns a pointer to the IEnumConnections interface, enabling a client to enumerate all the connections that exist on a connection point. IEnumConnections object keeps the CONNECTDATA elements. CONNECTDATA is a structure which has a pointer to the sink interface and its associated cookie information.

IEnumConnections

source keyword in MIDL indicates that the defined interface is a source of events and that the interface is called and won't be implemented by the server.

A connection point is an object managed by the connectable object that implements the IConnectionPoint interface.

Advise method is called by the client in oredre to provide the connectable object with a pointer to the sink object. This method must return a unique number to the client that identifies the advisory relationship that has been established. This number is called a "cookie" and is used for terminating the connection between client and server object.

The Unadvise method is used to terminate an advisory relationship that was previously established using the Advice method.

Client gets the pointer to server object, queries the IConnectionPointContainer interface pointer, looks at whether the server supports the registration of certain sink interface which client has, gets the pointer to IConnectionPoint interface and calls Advise on this interface by providing the pointer of its sink object. Keeps the cookie comnig from as a result of Advise method and when it no longer wants to receive those callbacks, client invokes Unadvise method on the IConnectionPoint interface and releases all the interface pointers that client collected from this process. These four round trip is necessary for the first connection set-up and sometimes it might not be desirable especially in remote out-of-process setup because the cost involved.

Distributed Component Object Model(DCOM)

DCOM is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. Microsoft's Distributed Component Object Model (DCOM) currently is avaiable on two operating systems: Window95 and Windows NT4.0. DCOM is language-independent, and objects are described via their interfaces using Microsoft's Object Description Language(ODL). DCOM provides everything COM does and additionally it has the following features:

- DCOM supports *distributed object reference counting* by Pinging protocol. Client machines send a periodic message to indicate that they are still using the object. DCOM considers a connection as broken if more then three ping periods pass without the component receiving a ping message. If the connection is broken, DCOM decrements the reference count and releases the component if the count becomes zero(no client).

- DCOM's preferred transport protocol is the connectionless UDP subset of the TCP/IP protocol suite. DCOM takes advantage of connectionless protocol and merges many low level acknowledge messages with actual data and pinging messages.

- Proxy handles the client requests and serves the client. But proxy can bundle the several request and keeps their results in its cache and serves the client from this cache. So that every client request might not need a network communication.

- DCOM uses Microsoft's RPC protocol (barrowed from OSF's DCE) for remote method invocation.

- Each component can have an Access Control List(ACL). When a client calls a method or creates an instance of a component, DCOM obtains the client's current username associated with the current process. Windows NT authenticates this information. On the server object, DCOM validates this information by looking at component's ACL.

  Application developers can use the same security services to define a finer security restrictions on the object such as interface based or method based security.

DCOM's **limited platform support** is one of the main disadvantage of this technology. Therefore, Microsoft embarked a joint project with SoftwareAG to make DCOM available under Unix and several other operating systems. Once this project is complete, then DCOM will be a proven distributed component framework for heterogenous hardware platforms.

Ditstributed components can be defined in DLL or EXE binaries. We are going to focus on the DLL in this section. DLL is a library description with certain hooks so that COM can extracts the information it needs from this library. When DLL is loaded within client's process, there is no problem with exexcution thread and security restrictions. When DLL needs to be loaded remotely (w.r.t. client's process), it requires an additional structure providing thread of execution as well as the security features. This problem is solved by surrogates: an executable responsible for loading a DLL, providing a thread of execution, and security features.

Default DLL surrogate (dllhost.exe) is already provided with DCOM distribution and it is generally sufficient for most of the DLLs. When a component runs in surrogate, it has to provide an AppId subkey under its CLSID registry entry. The value of AppId subkey is 128-bit GUID and COM looks for a match in HKEY_CLASSES_ROOT\AppId subtree of registry. An AppId registry entry may contain the following subkeys:

1. AccessPermission: sets the access control list (ACL) that determines default access permission.

2. ActivateAtStorage: configures client to activate on same system as persistent storage.

3. AuthenticationLevel: sets the default authentication level.

4. DllSurrogate: names the surrogate process used to load DLLs remotely.

5. LaunchPermission: sets the ACL that determines who can launch the application.

6. LocalService: configures the component to run as a Microsoft Windows NT Service.

7. RemoteServerName: names a remote server.

8. RunAs: sets an application to run only as a given user.

9. ServiceParameters: sets parameters to be passed to LocalService on invocation.

The value of DllSurrogate subkey denotes the surrogate object. If this subkey does not have a value, then COM uses the default DLL surrogate (dllhost.exe). An AppId entry groups the configuration and security options for distributed objects in the registry. Therefore, the same AppId entry can be shared by several objects.

As we know, client starts with CoCreateInstance() function to get the pointer to required interface. Third parameter of CoCreateInstance() function actually tells from where this object should be obtained. Let's assume that CLSCTX_LOCAL_SERVER is used as a third parameter. This definely rules out that the DLL which contains the component will run in the same process. As we recall, COM passes this request to the SCM and expects to get the pointer to the interface. SCM takes the following actions:

1. Checks the AppId key under the required CLSID registry.

2. If AppId key exists, then it checks AppId registry entries with the provided 128-bit GUID.

3. This registry entry directs SCM how to instantiate the server. If there is a DllSurrogate key, then this tells SCM to start this surrogate and obtain the pointer to required class from this component. If DllSurrogate subkey is not used and there is a RemoteServerName key, then this tells SCM to talk to SCM on remote machine to instantiate the server object.

The object can be running on the local machine or a remote machine. An AppId entry can contain DllSurrogate and RemoteServerName subkeys at the same time. Client directs COM by using third argument of the CoCreateInstance function to use which subkey. Lets assume that AppId entry contains both subkeys. If client calls CoCreateInstance with CLSCTX_LOCAL_SERVER, SCM uses DllSurrogate subkey. If client calls CoCreateInstance with CLSCTX_REMOTE_SERVER, then SCM uses RemoteServerName subkey.

You might wonder why there is a DllSurrogate subkey definition at all if the default DLL surrogate can take care of DLLs. The answer is still you might need them especially for DLL codes that

1. Access to global variables, and

2. Use function pointers in implementation of callback functions.

These components are developed under the assumption that DLL and the client code are going to run in the same process. Unfortunately, when we need to run this component remotely, we will run into a lot of unexpected problems. Solution is to write custom surrogates for these components.

Custom surrogate should register itself with CoRegisterSurrogate call and provide a mechanism to terminate itself when there is no client for the components. This can be done through events. Surrogate registers itself and waits for an event and as soon as it detects this event, it terminates. Custom surrogate should supply an implementation for ISurrogate interface, because COM will asks a pointer to ISurrogate interface. ISurrogate interface contains two functions: LoadDllServer and FreeSurrogate. After SCM finds out that it needs to use a DllSurrogate, then it starts this executable and asks a pointer to ISurrogate interface (through DllGetClassObject). SCM calls LoadDllServer function on this interface and it directs all CreateIntance calls to this surrogate. An implementation of ISurrogate interface can use CoCreateInstance function to provide a pointer to the required interface. COM informs surrogate via FreeSurrogate function that there is no client using any component supported by this surrogate. Therefore, in this function, surrogate can create the event (or condition) that it was waiting for termination.

So far we described how surrogates work and load a DLL remotetly. But, we did not tell how the pointer to required interface is transferred from one process to another. Since two processes have different memory layout (maybe on different machines), it is not possible to copy the pointer. This phase is called interface marshaling. As you recall, an interface pointer is a pointer to v-table structure. The same structure should be created in the client process. COM defines a proxy object in the client process for remote interfaces. Similary, a stub is defined in the server side. Proxy behaves like a real interface and forwards all function calls to the stub on the server side. Stub object redirects these calls to the implementation of the interface and returns back the results to the proxy. Then, proxy returns these results to the client like it is running in-process.

COM needs the neccessary code for proxy and stub implementation. Fortunately, MIDL compiler also produces implementation for proxy/stub DLL so that SCM locates the proxy/stub DLL with ProxyStubClsid32 subkey in the CLSID registry of the required class object. After SCM loads the DLL, then it asks a pointer to IPSFactoryBuffer interface via DllGetClassObject function.

IPSFactoryBuffer interface has two methods: CreateProxy and CreateStub. On the client side, CreateProxy function and on the server side CreateStub function is called. CreateProxy function creates an object which implements IRpcProxyBuffer interface as well as the necessary custom inteface. CreateStub function creates an object that implements IRpcStubBuffer interface. CreateStub buffer gets a pointer to the server object and the interface identifier.

IRpcProxyBuffer interface is used by the proxy manager to talk to a proxy. IRpcProxyBuffer interface has two methods: Connect and Disconnect. Connect function accepts a pointer to IRpcChannelBuffer interface that provides the necessary communication channel between proxy and stub. Disconnect function is called to notify the proxy that it is not connected to the stub anymore. It should release the IRpcChannelBuffer interface pointer.

IRpcStubBuffer interface is loaded into the component's address space and used for communication with the stub by the stub manager. IRpcStubBuffer inteface has the following functions:

1. Connect: accepts a pointer to the server interface.

2. IsDllSupported: tells whether the given interface id can be unmarshalled by the stub. COM queries the existing stub objects before creating a new stub to handle the required interface.

3. CountRefs: returns the total number of references the stub is holding on the component's objects. Stub should increment and decrement the number of clients in Connect and Diconnect functions, respectively.

4. DebugServerQueryInterface and DebugServerRelease: allows debuggers to trace the remote invocations on objects.

IRpcChannelBuffer interface introduces an abstraction which transparently (on the same machine or not) handles necessary transmission of marshalled data between processes. IRpcChannelBuffer has the following functions:

1. GetBuffer : asks a memory for marshalling.

2. FreeBuffer : indicates that user is done with the marshalling memory area.

3. SendReceive: sends the message, waits for the response.

4. IsConnected: tells whether there is a connection between proxy and stub.

5. GetDestCtx: describes the barrier between a component and its client:

    1. MSHCTX_LOCAL : marshalling and unmarshalling interfaces have the shared memory.

2. MSHCTX_NOSHAREDMEM : There is no shared memory between marshalling and unmarshalling interfaces.

3. MSHCTX_DIFFERENTMACHINE : marshalling and unmarshalling interfaces are running on different machines.

4. MSHCTX_INPROC

Invoke function of IRpcStubBuffer is used on the stub when the call is arrived.

COM allows three kind of marshalling:

1. Standard marshaling: The code provided by MIDL compiler is called as standard marshalling since it produces necessary proxy/stub DLL for the component defined in the IDL file. Proxy/Stub DLL also contains a necessary DllRegisterServer function for registering itself automatically.

2. Type Library Marshaling: This marshaller is provided for IDispatch interface (Automation) so that a component implementing the IDispatch interface can use this built-in marshaller.

3. Custom marshaling: An IMarshal interface implementation is provided by the developer. COM executes CoMarshalInterface function in the CoRegisterObject function so that CoMarshalInterface function asks object whether it has IMarshal interface support or not. If object returns a pointer to IMarshal interface implementation, then COM knows that object wants to do its own marshaling.

Proxy/Stub DLLs must be registed in related IID entry under ProxyStubClsid32 sub key.

CoMarshalInterface function queries IMarshal interface of the object. If it is not specified, then standard marshaller is used. If it is specified, then custom marshaling will be used. After COM finds that object wants to use custom marshaller, it asks the CLSID of the proxy object via GetUnmarshalClass function. COM asks the size of package marshaller wants to use via GetMarshalSizeMax function and allocates a space from memory. When COM needs to marshal the pointer to an interface, it calls MarshalInterface function to let the marshaller do its job. This buffer is sended back to the client side and this time COM uses UnmarshalInterface function call to let marshaller create a proxy object in the client's process. SCM can get the CLSID of the proxy object from this buffer and instantiates this object. Then it finds that this object has its own custom marshaling interface support. It handed it the buffer back to this marshaler via UnmarshalInterface method and let the proxy open a communication with the stub. If everything is alright, this method returns a pointer to the proxy object so that it can handle all calls related to the custom interface and forwards them to the stub.

DLL surrogate (dllhost.exe) is a default DCOM surrogate that provide security and thread of execution for any DLL. DLL surrogate object can be instructed to load any dll remotely.

Security is handled based on roles in the Windows NT system. Windows NT maintains an access token for each user logged on to the system, containing information identifying the user and the user groups to which the user belongs and information about other priveleges. Each process started by user also keeps the copy of this access token. The operating systems compares the access token of the running process with the security information attached to the system object that process wants to use to decide grand the permission or not.

Each user and user group has an unique security identifier (SID) in the access token. Each system object keeps a security descriptor containing the SIDs of the owner and the primary group to which owner belongs. iSecurity descriptor also contains the discretionary access control list (DACL) and system access control list (SACL). DACL controls the access permissions while SACL lists the object's operations that should generate audit messages. Elements in lists is called access control entry (ACE) and each ACE contains a grant or deny information for a specific user or user group.

COM provides a declarative security support and programmatic security support. Declarative security support allows to define security for per system or per component basis with the help of system tools. Programmatic security support requires implementation of several security interfaces to enforce more restricted security needs or selective security on the interface.

COM security aims to control activation, access, authentication, and identity. Activation control defines who has a permission to launch a component. Access control specifies who has a permission to access the component's objects. Authentication contol ensures that a network transmission is authentic and the data is protected agains unauthorized viewers. Identity control defines the security credentials under which the component executes.

Declarative security can configure all controls and store security information in the registry. Access and authentication control can be done with programmatically while activation and identity control can not since these controls are applied before the execution of the component. COM abstracted the security needs of components with Security Support Provider Interface (SSPI) API so that it is not tightly coupled with any operating system.

Using registry to keep the security information lets COM automatically enforce all the requirements. Even though it sounds like it is vulnarable model to keep all this information one place, COM only allows priveleged users to access the registry for update and run the utility programs related to registry. Declarative security allows to define a default security for all components as well as a security for each individual component. A security description for individual component overwrites the default security settings.

There are seven authentication levels defined by COM:

1. Default (RPC_C_AUTHN_LEVEL_DEFAULT) : connect level authentication.

2. None (RPC_C_AUTHN_LEVEL_NONE) : No authentication.

3. Connect (RPC_C_AUTHN_LEVEL_CONNECT) : Authenticates the client only when the client first connects to the server.

4. Call (RPC_C_AUTHN_LEVEL_CALL) : Authenticates the client before each remote method call.

5. Packet (RPC_C_AUTHN_LEVEL_PKT) : Authenticates that all of the data received is from the expected client.

6. Packet Integrity (RPC_C_AUTHN_LEVEL_PKT_INTEGRITY) : Authenticates all of the data and verifies that it has not been altered when tranferred between the client and the server.

7. Packet Privacy (RPC_C_AUTHN_LEVEL_PKT_PRIVACY) : Authenticates, verifies, and encrypts the arguments passed to every remote call.

Component's view of a client in security context is defined by Impersonation Level. COM supports the following impersonation levels:

1. Anonymous (RPC_C_IMP_LEVEL_ANONYMOUS) : The client is anonymous and the server can not obtain the client's identification information.

2. Identify (RPC_C_IMP_LEVEL_IDENTIFY) : The server can obtain the client's identification information and can impersonate the client for ACL checking but it can not access system objects as the client.

3. Impersonate (RPC_C_IMP_LEVEL_IMPERSONATE) : The server can act in behalf of the client by using client's identity.

4. Delegate (RPC_C_IMP_LEVEL_DELEGATE) : The server can act in behalf of the client to make calls on other servers by using client's identity.

Note that Anonymous is not supported in COM yet and Delegate option can be used in Windows NT 5.0. Default Impersonation Level and Default Authorization Level settings can be done with DCOM Configuration utility and applies to all users of the system.

Security description for each component is stored under AppId entries in the registry. Four subkeys (LaunchPermission, AccessPermission, AuthenticationLevel, and RunAs) are related to security definition of the component. These four subkeys controls all security items.

IAccessControl interface is the part of SSPI API for portable security support of COM.

Programmatic security allows a finer control on the security issues related to a component since the developer has all the control. CoInitalizeSecurity function in COM sets the security descriptions for the current process and it should be run after CoInitiallize() function to make sure that COM does not start a default security description. COM executes this function the first time an interface pointer is marshaled into or out of an apartment in the process. CoInitializeSecurity function allows developer to provide necessary security controls through parameters. The format of the function is:

1. pSecDesc defines the component's access permision. pSecDesc can point to a Win32 security descriptor, a GUID referring to AppId entry or implementation of IAccessControl interface. If GUID to AppId entry in the registry is used, then COM gets access permission information from AppId entry. If IAccessControl interface is used, then COM uses this interface. dwCapabilities flag informs COM about the type of pointer in this parameter.
   1. EOAC_APPID : a GUID referring to AppId,
   2. EOAC_ACCESS_CONTROL : a pointer to an implementation of IAccessControl interface, or
   3. None of the above : a pointer to Win32 security description structure.

2. cAuthSvc specifies the number of registered authentication services. Value 0 means that there is no authentication service and -1 instructs COM to choose.

3. asAuthSvc keeps a pointer to authentication services.

4. dwAuthLevel specifies the default authentication level for outgoing calls.

5. dwImpLevel specifies the default impersonation level for proxies.

Programmatic security is done thorugh providing an implementation of IAccessControl interface. COM calls IsAccessAllowed method to perform access checking at run time so that developer can decide to grant the access or not for a particular trustee (a user, a user group, etc.). Since COM can invoke this function in any thread, the implementation should be thread-safe.

Server can enforce higher levels of security a per-method basis by implementing the IServerSecurity interface which is used to identify the client and impersonate the client's identity. IServerSecurity interface contains the following functions:

- QueryBlanket : determines the client who initiated the current method call. Then, server can decide what to do according to the creditentials of the client.

- ImpersonateClient : temporarily server uses client's creditentials for its operations.

- RevertToSelf : server obtains its own creditentials after impersonating the client. This method should be called after server is done with the impersonating client, otherwise COM will restore the server's creditentials back.

CoGetCallContext function in COM allow to obtain the pointer to the IServerSecurity interface. Similarly, CoQueryClientBlanket function can be used to do QueryBlanket call on the interface, CoImpersonateClient function can be called for ImpersoneClient call, and CoRevertToSelf function can be called for RevertToSelf function call on the implementation of IServerSecurity of an object.

Client side proxy uses IClientSecurity interface for security purposes. IClientSecurity interface contains the following functions:

- QueryBlanket : determines the current security settings for a particular connection to an out-of-process object.

- SetBlanket : alters the current security settings for a particular connection to an out-of-process object.

- CopyProxy : makes a private copy of the proxy. This functionality is usefull especially client wants to change its security level only particular function(s). When SetBlanket is used, the security restrictions will be applied to all method calls in the interface. Therefore, client can create another copy of the proxy and alters necessary security settings for some of functions and uses only this proxy for those function calls.

As in IServerSecurity interface functions, COM provides several functions that directly calls the necessary functions on the proxy's IClientSecurity interface implementation. CoQueryProxyBlanket function calls QueryBlanket, CoSetProxyBlanket function calls SetBlanket, and CoCopyProxy function calls CopyProxy function on the IClientSecurity interface.

DCOM network protocol is called Object RPC (ORPC) since DCOM made some changes on the header description of standard DCE RPC definition. DCOM still uses the authentication, authorization, and message integrity features of the DCE RPC protocol. ORPC defines how calls are made on remote objects and how object references are represented, transmitted, and maintained.

IRemoteActivation interface is an RPC interface implemented by SCM on each machine. It has only one method RemoteActivation that activates the object on remote machine. RemoteActivation method returns the interface pointer identifier (IPID) and object exported identifier (OXID) for the requested object. IPID identifies the interface of an object running in a component and OXID identifies the RPC string binding information needed to connect to the interface specified by an IPID.

SCM runs on a specified well known endpoints for communication such as 135 for UDP and TCP/IP. Function calls are transferred through DCE RPC Protocol Data Units. PDUs contain the in parameters from client to server and out parameters from server to client.

DCOKM uses connectionless UDP in default but it provides reliability because RPC ensures robustness by using a customized mechanism for message ordering and acknowledgement.

DCOM adds an ORPCTHIS structures as first parameter to each request PDU and ORPCTHAT structure as first parameter to each reply PDU.

SCM contains an object implementing IOXIDResolver interface. This interface stores the RPC string bindings to connect with remote objects, sends ping messages to remote object as long as there are client's for this remote object in the local machine, and receives ping messages for object running on the local machine.

OXIDResolver caches the mapping from OXID to the RPC bindings for future uses so that it does not need to do the resolution everytime. OXIDResolvers are used for distributed garbage collection. Each OXIDResolver running on each machine sends only one ping message to machines on which there is a server object that is in use by a local client object. OXIDResolver tries to aggregate the ping messages into one message. For huge number of relationshoip between client-server structure, the size of the message can be very large. Therefore, OXIDResolver also supports delta pinging technique where OXIDs are collected into a ping sets and only a ping messages are sent for those ping sets.

DCOM releases the object pointer whenever objects misses three times ping message. DCOM assumes 120 secs between consequtive ping messages so that each object can stay alive ay least 6 minutes without receiving a ping message. After that, it can be terminated as soos as OXIDResolver decides to collect garbages.

Pinging is necessary if server object needs to keep a state information. But if the server object is stateless such as time server, then the server does not care whether client is alive or not and therefore there is no need for pinging. Component develper can direct DCOM by using MSHFLAGS_NOPING flag in CoGetStandardMarshal function call.

IRemUnkown is a DCOM interface that handles reference counting and interface pointer querying for remote objects. IRemUnknown interface contains the following functions:

- RemQueryInterface: is used for requesting several interface pointers in one call.
- RemAddRef: Increase the reference count of the object referred by IPIDs.
- RemRelease: Decrease the reference count of the object referred by IPIDs.

DCOM defines an IRemUnknown interface implementation for each OXID object in each apartment. The IRemUnkown interface is designed to minimize the network traffic as much as possible. Therefore, RemQueryInterface function can return several interface pointers in one request. DCOM also allows a context definition for each client so that reference counting operation can be done for each client seperately.

DCOM supports interceptors with IChannelHook interface. Interceptors are an extra code which will run after reception of each PDU before this PDU is processed and before the PDU is put into a wire. The logical connection between client and server object is defined as channel, therefore this technique is called channel hooks in DCOM. As you recall, DCOM adds ORPCTHIS for request PDUs and ORPCTHAT for reply PDUs and both structure contain an extension fields. This extension field can be used to send soe minformation from client to server as well as server side to client side by the help of channel hook mechanism. Extension field contains an array of extent fields which is uniquely identified with a GUID. To be able to use this field, we need to provide an implementation of IChannelHook interface. IChannelHook interface contains the following functions:

- ClientGetSize: DCOM calls this function to ask the Channel Hook the size of the data it is planning to insert while DCOM assembling the PDU package so that DCOM can decide the actual size of the PDU package on the wire.
- ClientFillBuffer: DCOM calls this function to inform the Channel Hook object that it is time for putting the data.
- ServerNotify: When DCOM receives a PDU package from the wire and there is a channel hook object, DCOM calls this function to notify the channel hook object before it executes the request PDU package.
- ServerGetSize: If channel hook object is designed to send additional data to the client withing each reply PDU, then DCOM calls this function to learn the size of the data that the channel hook object on the server side would like to add into the reply PDU.
- ServerFillBuffer: DCOM calls this function to inform the channel hook object on the server side that it is time to put the data into the PDU.
- ClientNotify: When DCOM receives the reply PDU from the server and there is a channel hook object in the client side, then DCOM calls this function to inform the channel hook that the reply message is arrived.

An implementation of IChannelHook interface is registered with CoRegisterChannelHook fnction in

DCOM and loaded into the address space of any client and server process by instantiating it via CoCreateInitialize function.

Microsoft Transaction Server (MTS)

MTS defines a set of COM interfaces for developing distributed components and an infrastructure for deploying and administering three-tier solutions. MTS provides a dll surrogate (mtx.exe) along with the MTS Executive (mtxex.exe). Exe-based components are not supported because they cannot be loaded into the address space of MTS. MTS requires the following featres from the component to give its services:

1. IClassFactory interface implementation and CreateInstance method should return a new object everytime (sigletons are not allowed).

2. DllGetClassObject function should be used to expose the class object of the component.

3. Component must support COM required standard reference counting.

4. Components running in the MTS process are not allowed to do aggregation to another component which is out o MTS process.

5. DllRegisterServer function in the implementation of component should register the ProgID, CLSID, Interface, and TypeLib.

6. The interfaces and classes in component must be defined in Type Library.

7. Component should use either standard marshaling or type library marshaling (custom marshaling is excluded).

8. -Oicf flag should be used with MIDL while producing proxy/stub DLL and proxy/stub DLL should be linked with mxil.lib file.

MTS supports an just-in-time activation by late object creation and early object deactivation. MTS tries to utilize the resources of server so that servers can be more scalable than it would be. The real activation object is delayed until client invokes the first function call on the interface. Similarly, the deactivation of object does not wait until the reference counts drops to zero, it can deallocate the object while some client still has the valid reference to this object. In a case a client wants to us ethe same interface again then MTS recerates the copy of the object and client never knows which object is in use. Ofcourse, this magic requires some additional wrappers.

IObjectControl interface aims to provide just-in-time activation with the following functions:

- Activate:

- Deactivate:

- CanBePooled:

MTS queries the IObjectControl interface when it instantiates ythe object. If this interface is supported, then MTS can use Activate function before the instantiation of the object and Deactivate function before the deactivation of the object. CanBePooled function returns whether an object is recyclable. If object can recycled, then this means that this object can be used by another client who would like to use an object to this same interface. MTS tries to address the recycling issue by introducing a distinction between the state of the object and the its code (behaviour). MTS provides several hooks to save teh state information for later reincarnation of the object.

The state information can be kept in:

- Client Side: Client application can keep the state of an object while making object in server stateless. This approach works fine if the object has a small state information so tthat it does not introduce extra overhead for transferring this information to the server object for each function call. But, it is unreasonable if the state information requires a large amount of data.

- Database State Storage: Object can store its state into a database through ODBC API or OLE DB interface. This approach is best suited if the object stores its durable state instead of every temporary state since it will create another bottleneck on the database.

- Shared Property Manager: The Shared Property Manager is defined by MTS as a resource for storing and sharing state among multiple objects in the server process.

MTS defines an implementation of IObjectContext interface for each running object in the server. IObjectContext ineterface contains CreateInstance function and extends from IUnkown interface. GetObjectContext function in MTS gets the pointer to the IObjectContext interface if object defines this interface, then MTS does not need to provide a default one. Under the control of MTS, instantiation of an object should be done with CreateInstance function of IObjectContext interface instead of CoCreateInstance function since an object created from IObjectContext interface can inherit the activity, transaction, and security identities from the creator's context object.

1. SetComplete() : informs that an object finished its work successfully for the client and it can be comitted by MTS and there is no need to keep the state of the object.

2. SetAbort(): informs the failure of the object's process and if this is done under a transaction it also informs MTS that this transaction should abort and object should be deactivated.

3. EnableCommit(): informs MTS that it is safe to commit the transaction and saving is no longer required but object did not quiet finished with its work yet.

4. DisableCommit(): informs MTS that it is not safe to commit the transaction, therefore, MTS should maintain the internal state of the object until further notice.

MTS uses the Microsoft Distributed Transaction Coordinator (MS DTC) which is originally developed for Microsoft SQL Server. MS DTC supports two-phase commit protocol and provides an implementation for four interfaces: ITransaction, ITransactionDispenser, ITransactionOptions, and ITransactionOutcomeEvents.

Component that is going to run in surrogate registers its class id with AppId key. AppId key entry contains the DllSurrogate key and if nothing is specified, system assumes that dllhost.exe is gonna be used for a surrogate component.

An AppId groups the configuration and security options for ditributed COM objects in the registry. Fo class identifier to AppId mapping, a corresponding AppId key word with its 128-bit GUID should be registered as a subkey for class registration. Note that the same AppId can be configured for several objects and this may simplyfy the security and configuration issues.

CoCreateInstance call's third parameter should be chenged to CLSCTX_LOCAL_SERVER.

SCM takes the following actions for this call:

1. checks the AppId key under the required class identifier registry.

2. If AppId key exists, then it checks AppId registry entries with the provided 128-bit GUID.

3. This registry entry directs SCM how to instantiate the server. If there is a RemoteServerName key, then this tells SCM to talk to SCM on remote machine to instantiate the server object.

Note that, client's registry should be configured such as that the component has a propoer AppId and its entry has a proper RemoteServerName key. Client may want to keep its 'dllhost.exe' entry as well as the RemoteServerName entry for the AppId entry. This case, third parameter in CoCreateInstance directs SCM how to handle the request. If CLSCTX_LOCAL_SERVER is used, then SCM starts the DLL on local machine with the surrogate. If CLSCTX_REMOTE_SERVER is used, then SCM looks for RemoteServerName and tells the SCM on the remote machine to start up the related DLL.

Writing the component under the distributed execution requires some attention since there is no shared process anymore. Access to global variables and callback functions implemented using function pointers are likely to cause problems. Therefore, this components may need a custom surrogate to handle this issues. Custom surrogate supports ISurrogate interface. Custom surrogate should register itself with CoRegisterSurrogate call.

Client's request is ended up by caalling the LoadDllServer function of the surrogate object. Surrogate component registers a generic class factory object for the requested class (CoRegisterClassObject) with REGCLS_SURROGATE value as a fourth parameter.

When the client calls CoCreateInstance for instantiation of an object as a local server, the surrogate's IClassFactory::CreateInstance function is called. Surrogate needs to call the object's IClassFactory interface. Therefore, it can use CoCreateInstance function so that COM can take care of this invocation.

FreeSurrogate function is called by COM when there is no client for this DLL. This process is initiated in CoFreeUnusedLibraries function of COM. When FreeSurrogate function is called, surrogate component should unregister class by CoRevokeClassObject function.

Function parameters are simply pushed onto the stack for a function call and then popped off the stack when executinmg the function for in-process components since they share the single address space.

Executable components can run as a Windows NT service or started by user with double click. An executable component has a main or WinMain function as start-up function.

Interface marshalling is to transfer the v-table related to the interface and define a proxy objetc on the client side which forwards all calls from client to a stub and back. MIDL compiler produces necessary proxy/stub DLL and any client can use them after registration.

Custom marshaling is appropriate solution for custom interfaces.

All proxy/stub DLL implements IPSfactoryBuffer interface so that COM put the pointer to to IPSFactoryBuffer interface through DllGetClassObject function. SCM uses ProxyStubClsid32 key to obtain the dll that contains necessary marshaling interface and calls, DllGetClassObject function to obtain a pointer to IPSFactoryBuffer interface.

IPSFactoryBuffer interface has two methods: CreateProxy and CreateStub. On the client side, CreateProxy function and on the server side CreateStub function is called. CreateProxy function creates an object which implements IRpcProxyBuffer interface as well as the necessary custom inteface defined by the developer. CreateStub function creates an object that implements IRpcStubBuffer interface. CreateStub buffer gets a pointer to the server object and the interface identifier.

IRpcProxyBuffer interface is used by the proxy manager to talk to a proxy. IRpcProxyBuffer interface has two methods: Connect and Disconnect. Connect function accepts a pointer to IRpcChannelBuffer interface that provides the necessary communication channel between proxy and stub. Disconnect function is called to notify the proxy that it is not connected to the stub anymore. It should release the IRpcChannelBuffer interface pointer.

IRpcStubBuffer interface is loaded into the component's address space and used for communication with the stub by the stub manager. IRpcStubBuffer inteface has the following functions:

1. Connect: accepts a pointer to the server interface.

2. IsDllSupported: tells whether the given interface id can be unmarshalled by the stub. COM queries the existing stub objects before creating a new stub to handle the required interface.

3. CountRefs: returns the total number of references the stub is holding on the component's objects. Stub should increment and decrement the number of clients in Connect and Diconnect functions, respectively.

4. DebugServerQueryInterface and DebugServerRelease: allows debuggers to trace the remote invocations on objects.

IRpcChannelBuffer interface introduces an abstraction which transparently (on the same machine or not) handles necessary transmission of marshalled data between processes. IRpcChannelBuffer has the following functions:

1. GetBuffer : asks a memory for marshalling.

2. FreeBuffer : indicates that user is done with the marshalling memory area.

3. SendReceive: sends the message, waits for the response.

4. IsConnected: tells whether there is a connection between proxy and stub.

5. GetDestCtx: describes the barrier between a component and its client:

    1. MSHCTX_LOCAL : marshalling and unmarshalling interfaces have the shared memory.

    2. MSHCTX_NOSHAREDMEM : There is no shared memory between marshalling and unmarshalling interfaces.

    3. MSHCTX_DIFFERENTMACHINE : marshalling and unmarshalling interfaces are running on different machines.

    4. MSHCTX_INPROC

Invoke function of IRpcStubBuffer is used on the stub when the call is arrived.

Proxy/Stub DLLs must be registed in related CLSID entry under ProxyStubClsid32 sub key.

CoMarshalInterface function queries IMarshal interface of the object. If it is not specified, then standard marshaller is used.

COM+

COM+ is the follower of COM and DCOM technologies with better development and technology supports. COM+ will provide a framework which will allow developers to build transaction-based distributed applications from COM components that can be managed from a single point of administration. COM+ will include

1. MTS transactions

2. Security administration and infrastructure services

3. Queued Request so that invocatons on COM components can be done through Microsoft's message-queuing server(MSMQ).

4. Publish-and-subscribe Event service.

5. Load balancing service to allow transaction requests to be dynamically spread across multiple servers.

All COM+ components are COM components. In terms of development approach, COM+ eases a lot of compilcated steps when we compared with its ancestors. For C++ development environment, it is worth to mention the following improvements:

- COM+ provides class factories most of the time.

- *coclass* keyword indicates that class should be exposed as a COM+ class. This informs C++ compiler to produce metadata about class. *coclass* defines the public view of an object and its private implementation. *coclass* contains the following definitions

  - **Fields** occupies a space in memory and only accessed from class.

  - **Properties** implemented as accessor functions so that it allows to provide runtime code that can be executed whenever a property is read or written.

- COM+ defines a data type set which enables data exchange through method calls between various languages.

- COM+ can intercept method calls. Whenever a method or property call is seen, compiler/interpreter calls to COM+ object service instead of direct method call on object. The type of interception related to services can be defined with attribute of coclass, cointerface and their methods therefore COM+ programmig is *declarative programming* or *attribute based programming*.

- COM+ allows to define exceptions that might be generated by a method. Necessary conversions are taken care of by the COM+ execution environment whenever language boundaries are crossed.

- COM+ can recognize *constructor* and *destructor* special methods so that it calls constructor whenever it creates a new instance of the object class and calls destructor before it destroys the object instance. So that it takes the burden of invoking AddRef() and Release() methods properly from developers' shoulder(no more reference counting). All method calls are done on object references instead of interface pointers.

- *cointerface* keyword indicates that the interface is COM+ interface. It is a good practice to use interface to define the public behavior and state of coclasses. Method and property definitions can be done in cointerface except that everything must be public and abstract. *cointerface* supports multiple inheritance. But developers no need to navigate through the QueryInterface() to get the pointer of in the interface they want, a simple type casting gets the job done with COM+.

- Overloading is allowed as long as signatures are different.

- COM+ supports several types of event mechanisms:

  interface-based events
  : Sinks implement interfaces to catch events. Sources fire events by calling methods on those interfaces. The problem with this approach, developer has to support all methods defined in the interface.

  method-based events
  : Instead of implementing the whole interface, only the particular method is implemented.

  persistent events
  : The source publishes information about events that can happen and sinks can subscribe to them.

- COM+ defines two different inheritance: *interface inheritance* and *implementation inheritance*. *coclass* indicates with *implements* keyword that it will implement the particular cointerface. *coclass* can inherit from both interface and implementation from another coclass. COM+ does not permit multiple inheritance of coclasses.

- A close related classes can be packaged in two ways:

    1. **logical** Namespace is a collector of classes and possibly other namespaces. A class belongs to only one namespace.

    2. **module** A single deployable unit of code (DLL or EXE).

- COM+ provides a standard registration service that writes information about classes to a machine specific database.

Notes about COM

Interface based programming provide Implementation Hiding so that client invokes methods on the interface instead of an object directly. This encapsulation supports location, platform, programming language transparency, and dynamic binding. Dynamic binding makes upgrading and reusing implementations easier. Note that just only supporting a particular interface while changing the semantic leads the process to catastrophe.

Platform and programming-language transparency requires a definition for common data type set which is a common denominator between data types of different languages. Therefore, COM and CORBA define a common data type set.

COM defines an integration infrastructure in which a component can interact with another within a single address space (in-process) or between processes on a single host (local or out-of-process).

COM binaries are replaceable as long as it supports the previous interfaces.

COM requires a definition of an object class that describes a set of interfaces to be fulfilled by the object. All interfaces should be extended from IUnknown base interface.

COM classes and interfaces are identified with Universally Unique Identifiers (UUID). CLASS ID refers to a particular implementation and supported interfaces are not known until runtime. Client navigates through QueryInterface to get the desired interface pointer because COM does not allow type casting (COM+ does).

Monikers provide a persistent naming mechanism and encapsulate implementation specific schemes based on names that encode an object’s location.

Life cycle is provided with reference counting. Unfortunately, client has to do increment and release operations whenever it copies the interface pointer or releases it, respectively.

Type Libraries allows dynamic method invocation. ITypeLib and ITypeInfo interfaces are provided for this purpose.

When client code is written such a way that it will query the object on the runtime and builds the method invocations by itself. Dynamic method invocation requires to pack parameters and a calling a function. Client almost writes the marshaling code for method invocation on the object. Idispatch interface provides the necessary mechanism. For this functionality, object implementation should provide the Invoke() method call of the interface to resolve request. This might be difficult because it requires extracting method name and parameters and then invoking the related method and returning the result if required. COM handles errors through 32 bit error code (HRESULT).

Two active objects are identical if their IUnkown pointers are identical. If objects are deactivated that means nothing.

COM's persistency is based on monikers.

Reliability and Reuse

Software Reliability [Meyer 1992] can be defined as the combination of correctness (to perform its job according to specification) and robustness (to handle abnormal situations). Reliability is an important ingredient to obtain reusable software components or modules.

When the software reliability is an important issue, it forces developer to write codes such a way that each module contains as much as possible checks before processing anything to make sure that parameters passed to the module are semantically correct and does not fall outside of the intended domain. This technique is known as defensive programming [Meyer 1992] in software engineering. Drawbacks with this approach are that these checks will degrade the software runtime performance as well as make maintenance process extremely complicated because of increased number of controls and code. Eventually, these will degrade the quality of software.

Booch Method

Booch prescribes four steps during Object Oriented Design (OOD) process:

1. Identify classes and objects: Key abstractions in the problem space are possible class and object candidates.

2. Identify the semantics of classes and objects: Try to define the lifecycle of each object from creation to destruction.

3. Identify relationships between classes and objects: What kind of interactions is necessary to fulfill expected task.

4. Implement classes and objects.

While going through these steps, there are several graphical representations to ease the whole development process:

1. Class diagrams and templates: defines classes and inheritance relationships.
2. Object diagrams and timing diagrams: defines message definitions, visibility and thread controls.
3. State-transition diagrams: defines object's lifecycle.
4. Operation templates: defines possible services.
5. Module diagrams and templates: defines the mapping of objects and classes to modules.
6. Process diagrams and templates: defines the mapping of modules to processors in case of multiprocessor configuration.

References

1. Windows Distributed interNet Applications Architecture(WindowsDNA)
   http://www.microsoft.com/dna/default.asp
2. Component Object Model(COM) Home
   http://www.microsoft.com/cominfo/
3. What OLE is really about?by Kraig Brockschmidt
   http://www.microsoft.com/oledev/olecom/aboutole.htm
4. Distributed Component Object Model(DCOM)
   http://www.microsoft.com/oledev/
5. ComponentWare Consortium
   http://www.componentware.com/
6. Distributed Computing Environment(DCE)
   http://www.opengroup.org/dce
7. DCOM and CORBA Side by Side, Step by Step, and Layer by Layer by P. Emerald Chung, Yennun Huang, Shalini Yajnik, Deron Liang, Joanne C. Shih, Chung-Yih Wan, Yi-Min Wang
   http://www.bell-labs.com/~emerald/dcom_corba/Paper.html
8. Notes on OLE and CORBA by Dinesh Katiyar
   ftp://theory.stanford.edu/pub/katiyar/misc/ole_vs_corba.html
9. Computing's Middle Ground -- Three middleware platforms battle it out Information Week, June 16, 1997
   http://www.techweb.com/se/directlink.cgi?IWK19970616S0051
10. Object-Oriented Programming for Structured Procedural Programmers by Emdad H. Khan, Mansoor Al-A'ali, and Moheb R. Girgis

IEEE Computer, Vol. 28, No. 10, October 1995
http://www.computer.org/computer/co1995/rx048abs.htm

11. The Joy Of Reference Counting in Object Magazine, July 1997
http://www.chappellassoc.com/art3.htm

12. COM/DCOM Resources by Yi-Min Wang
http://www.research.att.com/~ymwang/resources/DCOM.htm

13. For Microsoft, COM + MTS = Component Services Architecture
http://www.zdnet.com/pcweek/news/0323/23mcom.html

14. COM+ services add up
http://www.zdnet.com/pcweek/news/0413/13com.html

15. The COM+ Programming Model Makes it Easy to Write Components in Any Language by Mary Kirtland
http://www.microsoft.com/msj/1297/complus2/complus2.htm

16. Object-Oriented Software Development Made Simple with COM+ Runtime by Mary Kirtland
http://www.microsoft.com/msj/1197/complus.htm

17. COM versus CORBA: A Decision Framework, from QUOIN, www.quoininc.com

18. [Meyer 1992] B. Meyer, Applying "Design by Contract", IEEE Computer Magazine, Vol. 25, No. 10, pg:40-51, 1992

Last Updated on **October 28, 1998** by H. Timucin Ozdemir
URL : *http://www.npac.syr.edu/users/timucin/misc/book/com.html*
