UNIT V: Component Object Model

By
C Narendra

Component Object Model (COM)

COM is a Microsoft platform for software component introduced by Microsoft in 1993.

The Component Object Model (COM) is a software architecture that allows applications to be built from binary software components.

COM is the underlying architecture that forms the foundation for higher-level software services, like those provided by OLE.

OLE services span various aspects of commonly needed system functionality, including compound documents, custom controls, inter application scripting, data transfer, and other software interactions.

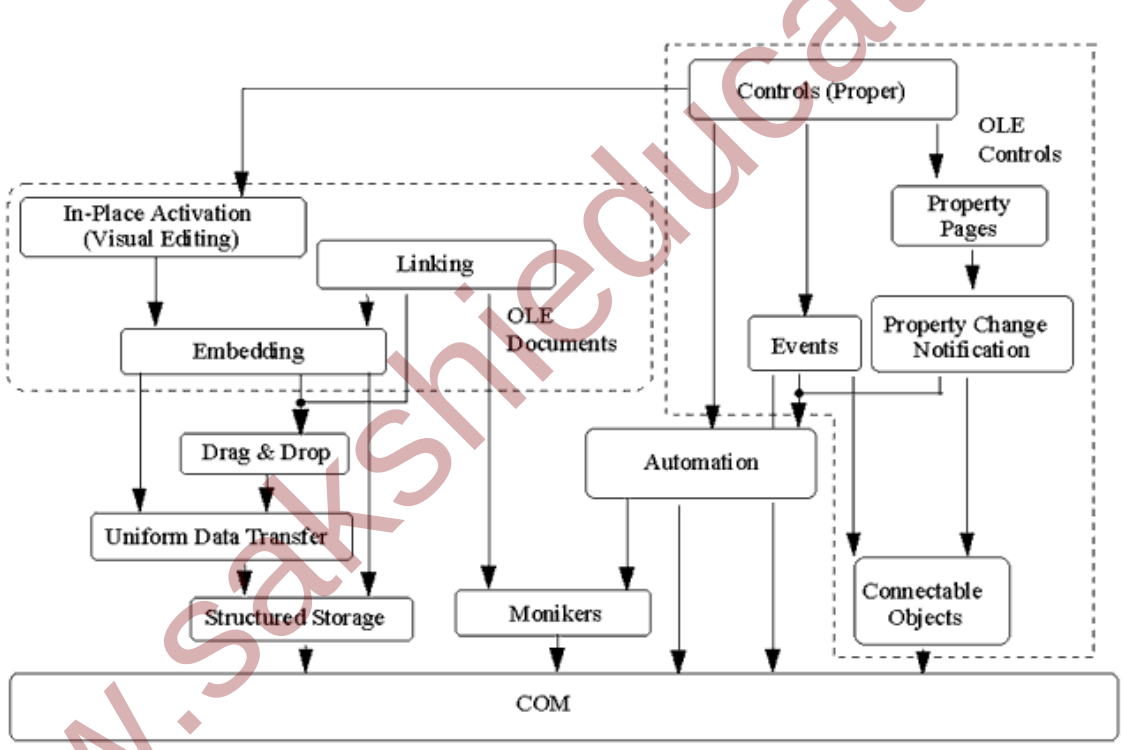


Fig: OLE technologies build on one another, with COM as the foundation.

Casting between different interfaces of an object is achieved through the Query Interface() function. The preferred method of inheritance within COM is the creation of sub-objects (called aggregation) to which method calls are delegated.

Although it has been implemented on several platforms, COM is primarily used with Microsoft Windows. COM is expected to be replaced to at least some extent by the Microsoft .NET framework, and support for Web Services through the Windows Communication Foundation (WCF).

Networked DCOM uses binary proprietary formats, while WCF uses XML-based SOAP messaging. COM also competes with CORBA and Java Beans as component software systems.COM Fundamentals

The Component Object Model defines several fundamental concepts that provide the model's structural underpinnings. These include:

- A binary standard for function calling between components.
- A provision for strongly-typed groupings of functions into interfaces.
- A base interface providing:

- A way for components to dynamically discover the interfaces implemented by other components.
- Reference counting to allow components to track their own lifetime and delete themselves when appropriate.

- A mechanism to identify components and their interfaces uniquely, worldwide.
- A "component loader" to set up component interactions and, additionally (in the cross-process and cross-network cases), to help manage component interactions.

Advantages of COM

1. COM promotes component-based software development.
2. COM promotes code reusability.
3. COM promotes Object-oriented programming (OOP).
4. COM comprises the necessary mechanisms for COM components to communicate with each other.
5. COM helps to access components loaded in different machines on the network.

1. COM promotes component-based software development :

Before component-based development came, software programs have been coded using procedural programming paradigm, which supports linear form of program execution.

But component-based program development comes with a number of advantages, such as the ability to use pre-packaged components and tools from third party vendors into an application and support for code reusability in other parts of the same application.

2. COM promotes code reusability:

Standard classes are normally reused in the same application but not to be used easily in other applications. But COM components are designed to separate themselves from single applications and hence can be accessed and used by several different applications without any hassle.

3. COM promotes Object-oriented programming (OOP) :

Encapsulation which allows the implementation details of an object to be hidden. Encapsulation helps to hide how an object has implemented a method internally. This is most important characteristic.

Polymorphism, which is the ability to exhibit multiple behaviors.

Inheritance, which allows for the reuse of existing classes in order to design new and more specialized classes.

This ultimately helps to incorporate more vigorously implemented or advanced implementation into an object at later time without affecting the client which uses it.

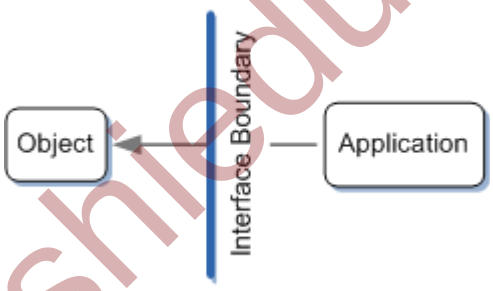
4. COM comprises the necessary mechanisms for COM components to communicate with each other

In the normal case, two components coded using two different programming languages can not communicate with each other. But COM can make it possible for different language components, which adhere to the COM specification, to interact with each other and hence COM is language-independent.

5. COM helps to access components loaded in different machines on the network

COM abstracts away the nitty-gritty of clients to locate COM components anywhere in the network. Thus COM provides location transparency and COM components are location independent.

COM Interfaces



An interface defines a set of methods that an object can support, without dictating anything about the implementation. The interface marks a clear boundary between code that calls a method and the code that implements the method. In computer science terms, the caller is decoupled from the implementation.

COM uses the word interface in a sense different from that typically used in Visual C++ programming. A C++ interface refers to all of the functions that a class supports and that clients of an object can call to interact with it.

A COM interface refers to a predefined group of related functions that a COM class implements, but a specific interface does not necessarily represent all the functions that the class supports.

Components, especially server-side ones, are bound to have their own interfaces.

An interface is simply a list of methods a COM component implements and makes them available to consumers. COM interface are immutable. That is, once a COM component has been released with an interface, this interface must never be manipulated.

COM interfaces are the mechanisms by which a client contacts, connects and use components. If the client component is using a different language from the one a server component is coded, there will be language problem for interactions.

In order to remove this barrier, COM interfaces provide an universal way so that components coded in different languages can communicate with each other. Thus every interface constitutes a binding contract between a COM object (no interface is attached with this) and a COM component with an interface.

COM specification makes life easier by allowing a component to publish more than one interface at a time. New interfaces can be incorporated to support new features of the components while keeping the original interface intact.

We have to increment a version number for the new interface. A class module then would need to be implemented for both old and new interfaces in order to support both old clients that rely on the old interface and new clients that can take advantage of the new interface. Thus the immutability factor of COM interfaces can be overcome.

The steps in creating a COM interface are as follows:

- Decide how you want to provide marshaling support for your interface; either with type-library “driven marshaling or with a proxy/stub DLL.
- Even in-process interfaces must be marshaled if they are to be used across apartment boundaries. It is a good idea to build marshaling support into every COM interface.
- Describe the interface or interfaces in an interface definition (IDL) file. In addition, you can specify certain local aspects of your interface in an application configuration file (ACF).
- If you are using type-library“driven marshaling, add a library statement that references the interfaces for which you want to generate type information.
- Use the MIDL compiler to generate a type library file and header file, or C-language proxy/stub files, interface identifier file, DLL data file and header file. See MIDL Compilation for more information.
- Depending on the marshaling method you chose, write a module definition (DEF) file, compile and link all the MIDL-generated files into a single proxy DLL, and register the interface in the system registry, or register the type library. See Loading and Registering a Type Library and Building and Registering a Proxy DLL for more information.

Interface Definition Language (IDL)

IDL is just a declaration language, not a programming language.

The syntax for IDL is almost similar to C++ language. But IDL goes beyond what C++ can offer by being able to define functions that extend process boundaries.

IDL provides many vital extensions that allow attributes such as type libraries, co-classes, interfaces, and method parameters to be specified quite elegantly. But if there is no language problem, it is not mandatory to use IDL for defining COM interfaces.

The interface definition begins with an object attribute. The object attribute is used to identify a COM interface. Then a **UUID** (universally unique identifier), which provides an unique identifying number (a string of hexadecimal digits) for each interface and this number is being generated by an algorithm, which takes into account the unique value from the network card on the programmer's PC and the current system date and time, follows that.

The letter **'I'** in the interface name helps to identify that this is an interface. Following *IFindSum* is a colon and then the interface name IUnknown, the base interface and every COM object has to have this interface. IUnknown has three methods that a client can invoke: *QueryInterface, AddRef, & Release.*

The *QueryInterface* method is the mechanism, that a client uses to discover and navigate the interfaces of a component dynamically. It is the most significant method of all COM interfaces available for a component since this method allows run-time checking of all of these interfaces.

When the QueryInterface method provides an interface pointer to a client, the QueryInterface calls the method AddRef. The only way to access a COM object is through a pointer to an interface.

An interface pointer is actually a pointer to a pointer in a virtual table that is implemented in memory. That is, for every class that contains public methods, a virtual table will be created and placed in memory at run time.

Virtual tables are generated for each class and not for objects of that class. Each virtual table contains an array of function pointers whose elements contain the address of each specific function that an object can implement.

At the top of every virtual table array will be three fundamental methods that make up this IUnknown interface.

The methods AddRef and Release manage reference counting. This reference count is an internal count of the number of clients using a particular COM object .

I t is possible for several clients to invoke the services available with a COM object at the same time. When a client begins a session with an object, it calls the AddRef method, which increments one to the existing count.

Once the session is over, the corresponding Release method gets invoked to decrement one from the existing count.

The next line down is used to locate the definition for the base interface IUnknown The next line identifies the interface method "SumofNumbers". This interface method is designed to return an **HRESULT** value.

An HRESULT is used with COM to indicate whether a method call is successfully accomplished or not. Almost all COM interfaces return a special 32-bit code, called as

HRESULT, from their methods to return status information. It is actually the method COM utilizes to return errors to the caller of the method.

The reason for this mechanism is due to the inability of COM to transmit error messages back to the client if there is anything wrong on the server side or on the way to the server. As components are being written using different programming languages and each language follows its own exception mechanisms, COM can not correctly pass on the error messages and hence HRESULT.

Following the HRESULT are three parameters for the interface method SumOfNumbers.

The [in] in the first two parameters specifies that these parameters are input values to be passed to the interface method and the [out, retval] in the last parameter indicates that this parameter returns a value that will be ultimately passed to back to the client.

Windows Registry and Components Environment

COM components may be loaded in different address spaces in the same machine or machine connected with the network. Before an object can be created, the COM runtime must first locate the COM components. The COM runtime is able to locate COM components through the use of Windows registry.

COM components may be out-of-process or in-process. An out-of-process component can effectively be a separate program containing objects. This type runs in its own process space and will thus be largely independent from any client programs.

These components are often large applications themselves, and provide access to their objects to make it easy for other programs to create macros or otherwise make use of existing functionality. This can be very beneficial, because the component developer can choose his own threading models. The drawback here is that the developer has to write the code for threading. This work can be handled by automatically COM+, the latest one from the COM family.

There is also a performance issue. Because the client application is communicating with objects which are in an entirely different process there is quite a bit of overhead.

COM steps in between the client and the server and handles all the communication between them. Mainly out-of-process components are useful if we are trying to make use of the objects in a pre-existing application.

An in-process component is one where the objects run inside the client's process. Each client gets its own private copy of the component and hence a copy of all the objects in that component. An in-process component is often called COM DLLs or ActiveX DLLs.

In-process component does not have a process of its own and it always runs within the context of another process. Often this other process is the client application itself but may be run within the context of a different process entirely.

In this case, we can get performance boost as the component is loaded in the client application itself. There is almost no overhead when client interacts with an object that is running in the same process. The major advantages of having in-process components are stability and increased manageability.

Creating a COM object

A client looking for an object has to make a call to an API function called CoCreateInstance with the CLSID for the class, which comprises the particular object.

The prefix Co in this function is a typical naming convention for COM runtime API functions. CoCreateInstance in turn as a second step calls another API function called CoGetClassObject. Basically the function CoCreateInstance asks the SCM to search the Windows registry location, which is HKEY_CLASSES_ROOT\CLSID.

The function of SCM is to locate the requested class object by using the specified CLSID.

The first place the SCM looks to locate the class object is within its own internal database. If the class is not located inside its database, the SCM turns to the system registry as a third step. As soon as the class object is found, the SCM immediately returns to COM an interface pointer referred to as IClassFactory.

Every COM class must have a factory object associated with it. The main purpose of these class objects is to implement the IClassFactory interface.

Once the client gets the IClassFactory pointer, two things happen. One, by obtaining a pointer to the IClassFactory, CoCreateInstance can call IClassFactory::Create Instance in order to create the requested COM object. Once the object is created, a second step occurs: the interface pointer to the IClassFactory gets released.

The interface called IClassFactory's job is to talk to other class objects. Further, CoGetClassObject is a COM class object whose sole purpose is to create a new instance of a different class. Because this COM class creates class objects by implementing the IClassFactory interface, CoGetClassObject is referred to as a class factory.

The following rules apply to all interfaces implemented on a COM object:

- They must have a unique interface identifier (IID).
- They must be immutable. Once they are created and published, no part of their definition may change.
- All interface methods must return an **HRESULT** value so that the portions of the system that handle remote processing can report RPC errors.
- All string parameters in interface methods must be Unicode.
- Your data types must be remotable. If you cannot convert a data type to a remotable type, you will have to create your own marshaling and unmarshaling

routines. Also, **LPVOID**, or **void ***, has no meaning on a remote computer. Use a pointer to **IUnknown**, if necessary.

LockServer

When an object is created it resides in memory. It is the client's job to notify the object when it is no longer needed.

The Release method told above can decrement the reference count for the interface. If there are several interfaces for an object, it is possible that the object is still being accessed somewhere else. The idea is that the reference count has to reach zero before the object is to be released from the memory.

Releasing objects from memory usually improves performance. Some special clients constantly create and destroy objects, which normally affects the application's performance.

The LockServer method allows components to remain in memory by incrementing the reference count by one. Because LockServer keeps a count of one in the reference counter, the object will stay in memory even if it is not being accessed currently.

Having the component reside in memory has the advantage of allowing the object to be called without having to go through the process of recreating it. But on the other hand, if objects are not destroyed when they are not in use, there will be reduction of system memory resources.

COM Data Types

The following table shows data types used in COM and their corresponding .NET Framework built-in value types or classes. Any type not explicitly identified in this table is converted to an **Int32** system type.

COM value type	COM reference type	System type
bool	**bool ***	System.Int32
char, **small**	**char ***, **small ***	System.SByte
short	**short ***	System.Int16
long, **int**	**long ***, **int ***	System.Int32
Hyper	**hyper ***	System.Int64
unsigned char, **byte**	**unsigned char ***, **byte ***	System.Byte
wchar_t, **unsigned short**	**wchar_t ***, **unsigned short ***	System.UInt16
unsigned long,	**unsigned long ***, **unsigned**	System.UInt32

COM value type	COM reference type	System type
unsigned int	**int ***	
unsigned hyper	**unsigned hyper ***	System.UInt64
float	**float ***	System.Single
double	**double ***	System.Double
VARIANT_BOOL	**VARIANT_BOOL ***	System.Boolean
void *	**void ****	System.IntPtr
HRESULT	**HRESULT ***	**System.Int16** or **System.IntPtr**
SCODE	**SCODE ***	System.Int32
BSTR	**BSTR ***	System.String
LPSTR or **[string, …] char ***	**LPSTR ***	System.String
LPWSTR or **[string, …] wchar_t ***	**LPWSTR ***	System.String
VARIANT	**VARIANT ***	System.Object
DECIMAL	**DECIMAL ***	System.Decimal
DATE	**DATE ***	System.DateTime
GUID	**GUID ***	System.Guid
CURRENCY	**CURRENCY ***	System.Decimal
IUnknown *	**IUnknown ****	System.Object
IDispatch *	**IDispatch ****	System.Object
SAFEARRAY(type)	**SAFEARRAY**(type) *	type[]

The following table lists COM value and reference types that convert to corresponding element types. For example, a COM coclass automatically maps to a managed class with the same name.

COM value type	COM reference type	Element type
Typedef BaseType MyType	**ByRef BaseType**	**BaseType**
MyStruct	**ByRef VALUETYPE**<MyStruct>	**valuetype**<MyStruct>
MyEnum	**ByRef VALUETYPE**<MyEnum>	**valuetype**<MyEnum>
MyInterface *	**ByRef CLASS** <MyInterface>	**Class** <MyInterface>
MyCoClass	**ByRef CLASS** <_Class>	**Class** <_Class>

Proxy

A proxy resides in the address space of the calling process and acts as a surrogate for the remote object. From the perspective of the calling object, the proxy is the object.

Typically, the proxy's role is to package the interface parameters for calls to methods in its object interfaces. The proxy packages the parameters into a message buffer and passes the buffer onto the channel, which handles the transport between processes.

The proxy is implemented as an aggregate, or composite, object. It contains a system-provided, manager piece called the proxy manager and one or more interface-specific components called interface proxies.
The number of interface proxies equals the number of object interfaces that have been exposed to that particular client. To the client complying with the component object model, the proxy appears to be the real object.

Each interface proxy is a component object that implements the marshaling code for one of the object's interfaces.

The proxy represents the object for which it provides marshaling code. Each proxy also implements the IRpcProxyBuffer interface.

Although the object interface represented by the proxy is public, the **IRpcProxyBuffer** implementation is private and is used internally within the proxy.

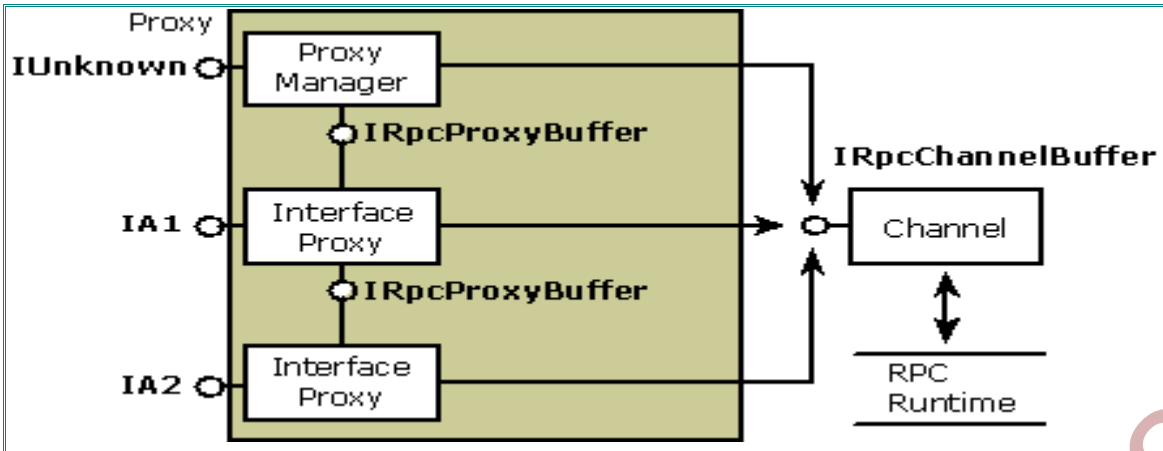
The proxy manager keeps track of the interface proxies and also contains the public implementation of the controlling IUnknown interface for the aggregate.

Each interface proxy can exist in a separate DLL that is loaded when the interface it supports is materialized to the client.

Structure of the Proxy

Each interface proxy implements **IRpcProxyBuffer** for internal communication between the aggregate pieces.

When the proxy is ready to pass its marshaled parameters across the process boundary, it calls methods in the IRpcChannelBuffer interface, which is implemented by the channel. The channel in turn forwards the call to the RPC run-time library so that it can reach its destination in the object.



Stub

The stub, like the proxy, is made up of one or more interface pieces and a manager.

Each interface stub provides code to unmarshal the parameters and code that calls one of the object's supported interfaces.

Each stub also provides an interface for internal communication. The stub manager keeps track of the available interface stubs.

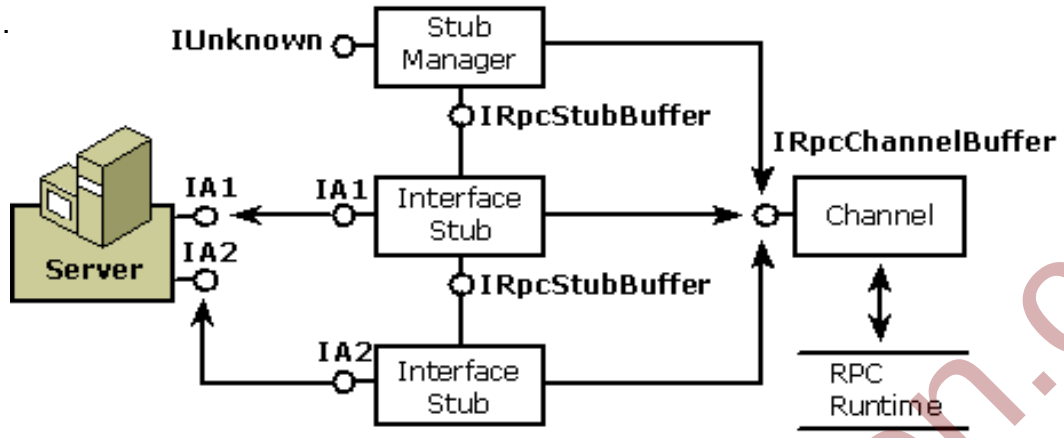
There are, however, the following differences between the stub and the proxy:

- The most important difference is that the stub represents the client in the object's address space.
- The stub is not implemented as an aggregate object because there is no requirement that the client be viewed as a single unit; each piece in the stub is a separate component.
- The interface stubs are private rather than public.
- The interface stubs implement IRpcStubBuffer, not IRpcProxyBuffer.
- Instead of packaging parameters to be marshaled, the stub unpackages them after they have been marshaled and then packages the reply.

Structure of the Stub

Each interface stub is connected to an interface on the object. The channel dispatches incoming messages to the appropriate interface stub.

All the components talk to the channel through IRpcChannelBuffer, the interface that provides access to the RPC run-time library



Marshalling

COM handles all of the details described in this section for you. This section is provided for those few programmers who need these details and for those interested in the underlying information.

Marshaling is the process of packaging and unpackaging parameters so a remote procedure call can take place.

Different parameter types are marshaled in different ways. For example, marshaling an integer parameter involves simply copying the value into the message buffer. Marshaling an array, however, is a more complex process.

Array members are copied in a specific order so that the other side can reconstruct the array exactly. When a pointer is marshaled, the data that the pointer is pointing to is copied following rules and conventions for dealing with nested pointers in structures.

Unique functions exist to handle the marshaling of each parameter type.

With standard marshaling, the proxies and stubs are systemwide resources for the interface and they interact with the channel through a standard protocol.

Standard marshaling can be used both by standard COM-defined interfaces and by custom interfaces, as follows:

- In the case of most COM interfaces, the proxies and stubs for standard marshaling are in-process component objects which are loaded from a systemwide DLL provided by COM in ole32.dll.

- In the case of custom interfaces, the proxies and stubs for standard marshaling are generated by the interface designer, typically with MIDL.

  These proxies and stubs are statically configured in the registry, so any potential client can use the custom interface across process boundaries.

  These proxies and stubs are loaded from a DLL that is located via the system registry, using the interface ID (IID) for the custom interface they marshal.

- An alternative to using MIDL to generate proxies and stubs for custom interfaces, a type library can be generated instead and the system provided, type-library–driven marshaling engine will marshal the interface.

Standard marshaling, an interface (standard or custom) can use custom marshaling. With custom marshaling, an object dynamically implements the proxies at run time for each interface that it supports. For any given interface, the object can select COM-provided standard marshaling or custom marshaling.

Custom marshaling is inherently unique to the object that implements it. It uses proxies implemented by the object and provided to the system on request at run time. Objects that implement custom marshaling must implement the Imarshal interface, whereas objects that support standard marshaling do not.

If you decide to write a custom interface, you must provide marshaling support for it. Typically, you will provide a standard marshaling DLL for the interface you design. You can use the tools contained in the Platform SDK CD to create the proxy/stub code and the proxy/stub DLL. Alternatively, you can use these tools to create a type library which COM will use to do data-driven marshaling.

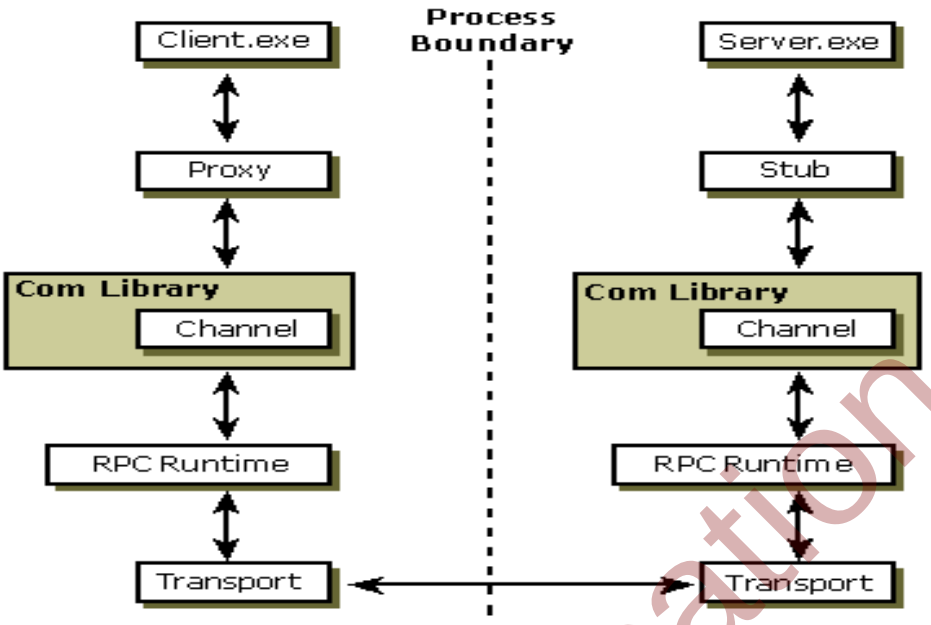
Components of Inter process communications

On the client side of the process boundary, the client's method call goes through the proxy and then onto the channel, which is part of the COM library.

The channel sends the buffer containing the marshaled parameters to the RPC run-time library, which transmits it across the process boundary.

The RPC run time and the COM libraries exist on both sides of the process. The distinction between the channel and the RPC run time is a characteristic of this implementation and is not part of the programming model or the conceptual model for COM client/server objects.

COM servers see only the proxy or stub and, indirectly, the channel. Future implementations may use different layers below the channel or no layers.



Interface Pointers

An instance of an interface implementation is actually a pointer to an array of pointers to methods that is, a function table that refers to an implementation of all of the methods specified in the interface.

Objects with multiple interfaces can provide pointers to more than one function table. Any code that has a pointer through which it can access the array can call the methods in that interface.

Each interface the immutable contract of a functional group of methods is referred to at run time with a globally unique interface identifier (IID). This IID, which is a specific instance of a globally unique identifier (GUID) supported by COM, allows a client to ask an object precisely whether it supports the semantics of the interface.

To summarize, it is important to understand what a COM interface is, and is not:

- **A COM interface is not the same as a C++ class**

1. If you are a C++ programmer, you can define your implementation of an interface as a class, but this falls under the heading of implementation details, which COM does not specify.
2. Different object classes may implement an interface differently yet be used interchangeably in binary form, as long as the behavior conforms to the interface definition.

- **A COM interface is not an object**

  It is simply a related group of functions and is the binary standard through which clients and objects communicate. As long as it can provide pointers to interface methods, the object can be implemented in any language with any internal state representation.

- **COM interfaces are strongly typed**

  Every interface has its own interface identifier (a GUID), which eliminates the possibility of duplication that could occur with any other naming scheme.

- **COM interfaces are immutable**

  You cannot define a new version of an old interface and give it the same identifier. Adding or removing methods of an interface or changing semantics creates a new interface, not a new version of an old interface. Therefore, a new interface cannot conflict with an old interface.

Object Creation and Destruction

Because exceptions are excluded from the kernel's restricted form of C++, you cannot implement "normal" C++ constructors and destructors without jeopardy.

Constructors and destructors are typed to return no value (such as an error code). Normally, if they encounter a problem, they raise an exception. But because exceptions aren't supported in the kernel's C++ runtime, there is no way for you to know when an allocation or deallocation error has occurred.

The macros also define the primary constructor and a destructor for a class. These macro-created constructors are guaranteed not to fail because they do not themselves perform any allocations. Instead, the runtime system defers the actual allocation of objects until their initialization (usually in the init member function).

Comparison of COM and CORBA

Three of the most popular distributed object paradigms are Microsoft's

Distributed Component Object Model (DCOM)

Common Object Request Broker Architecture (CORBA)

and JavaSoft's **Java/Remote Method Invocation (Java/RMI)**.

CORBA relies on a protocol called the **Internet Inter-ORB Protocol (IIOP)** for remoting objects. Everything in the CORBA architecture depends on an **Object Request Broker (ORB)**. The ORB acts as a central Object Bus over which each CORBA object interacts transparently with other CORBA objects located either locally or remotely.

Each CORBA server object has an interface and exposes a set of methods. To request a service, a CORBA client acquires an object reference to a CORBA server object. The client can now make method calls on the object reference as if the CORBA server object resided in the client's address space.

The ORB is responsible for finding a CORBA object's implementation, preparing it to receive requests, communicate requests to it and carry the reply back to the client.

A CORBA object interacts with the ORB either through the ORB interface or through an **Object Adapter** - either a **Basic Object Adapter (BOA)** or a **Portable Object Adapter (POA)**.

DCOM which is often called 'COM on the wire', supports remoting objects by running on a protocol called the **Object Remote Procedure Call (ORPC)**. This ORPC layer is built on top of DCE's RPC and interacts with COM's run-time services. A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime.

Java/RMI relies on a protocol called the **Java Remote Method Protocol (JRMP)**. Java relies heavily on Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream.

Since Java Object Serialization is specific to Java, both the Java/RMI server object and the client object have to be written in Java. Each Java/RMI Server object defines an interface which can be used to access the server object outside of the current Java Virtual Machine(JVM) and on another machine's JVM.

Introduction to .NET

The .NET Framework introduces a completely new model for the programming and deployment of applications. .NET is Microsoft's vision of "software as a service", a development environment in which you can build, create, and deploy your applications

and the next generation of components, the ability to use the Web rather than your own computer for various services.

Microsoft introduced great technologies like COM, DCOM, COM+ etc. to enable reuse of Software. Although these technologies are very powerful to reuse Software, they required a huge learning curve.

The .NET Framework was born:

Microsoft changed all complex tasks with the new .NET Framework. That was a huge advantage for all developers. Most of the Win32 API was now accessible through a very simple Object Model. Most of the features and functions of C++ were added to Visual Basic. A new Programming Language C# was introduced, which offered flexibility and productivity. ASP.NET also called ASP+ replaced ASP.

It provides the easiest and most scalable way to build, deploy and run web services. ASP.NET server controls enable an HTML-like style of declarative programming that let you build great pages with far less code than with classic ASP. VB, C++ and C# Code can be used in other languages.

The .NET Compilation Stages:

- The Code written in .NET isn't compiled directly to the executable, instead .NET uses two steps to compile the code.
- First, the code is compiled to an Intermediate Language called Microsoft Intermediate Language (MSIL).
- Second, the compiled code will be recompiled with the Common Language Runtime (CLR), which converts the code to the machine code.
- The basic Idea of this two stages was to make the code language independence.

The major Components (Layers) of the .NET framework:

The top layer includes user and program interfaces. Windows Forms are a new way to create standard Win32 desktop applications, based on the Windows Foundation Classes (WFC) produced for J++.

Web Forms provide a powerful, forms-based UI for the web. Web Services, which are

perhaps the most revolutionary, provide a mechanism for programs to communicate over the Internet using SOAP. Web Services provide an analog of COM and DCOM for object brokering and interfacing, but based on Internet technologies so that allowance is made for integration even with non-Microsoft platforms.

Web Forms and Web Services, comprise the Internet interface portion of .NET, and are implemented through a section of the .NET Framework referred to as ASP.NET. The middle layer includes the next generation of standard system services such as ADO.NET and XML. These services are brought under the control of the framework, making them universally available and standardizing their usage across languages.

The last layer includes system-level capability that a developer would need.

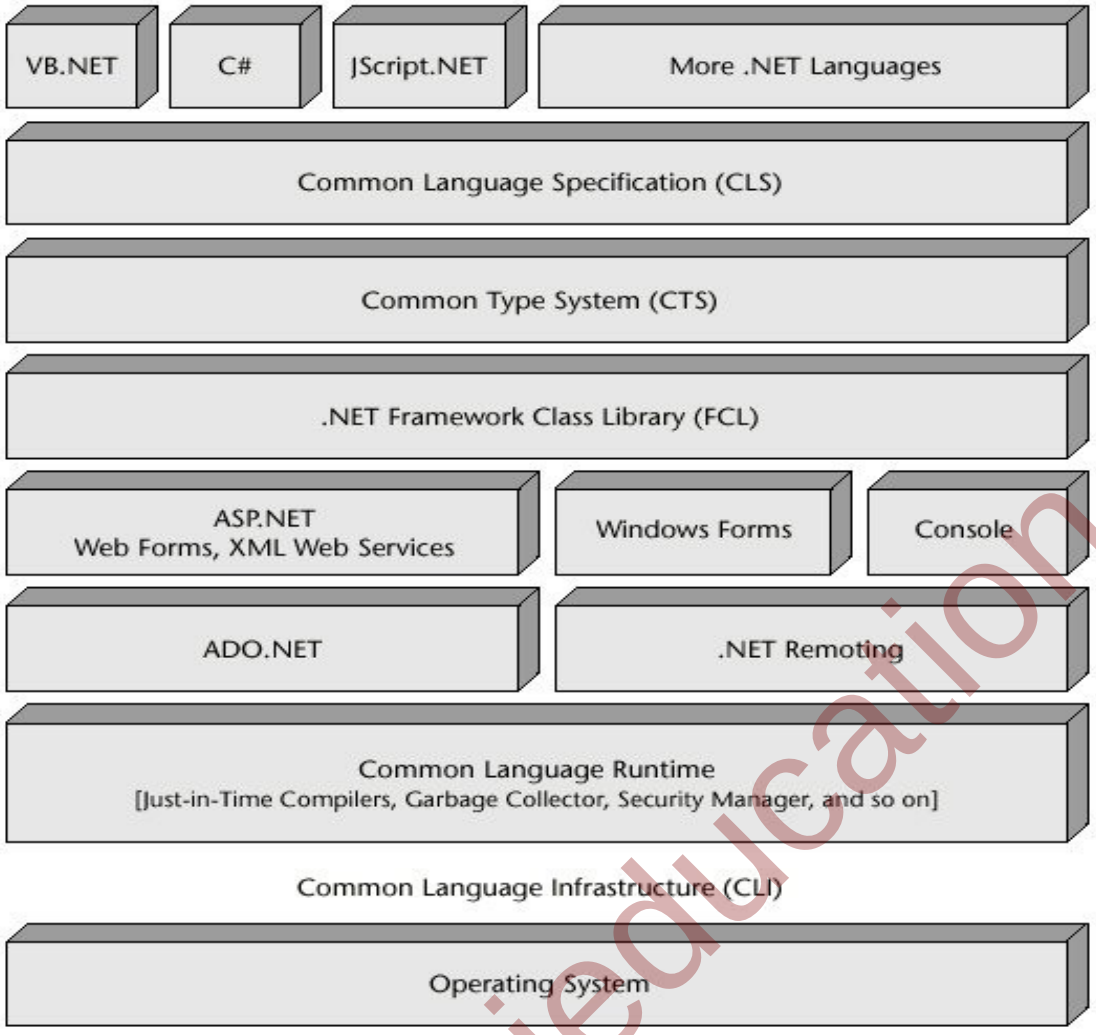
Overview of .Net Architecture

.NET as a new standard that will allow software to run anywhere, at any time, on any platform, and on devices large and small.

NET is tiered, modular, and hierarchal. Each tier of the .NET Framework is a layer of abstraction. .NET languages are the top tier and the most abstracted level.

The common language runtime is the bottom tier, the least abstracted, and closest to the native environment. This is important since the common language runtime works closely with the operating environment to manage .NET applications.

The .NET Framework is partitioned into modules, each with its own distinct responsibility. Finally, since higher tiers request services only from the lower tiers, .NET is hierarchal



.NET Pros

- It offers multiple language support.
- It has a rich set of libraries, a la JVM.
- It's open-standard friendly (e.g., HTTP and XML) -- it may even become a standard itself.
- Its code is compiled natively, regardless of language or deployment (Web or desktop).

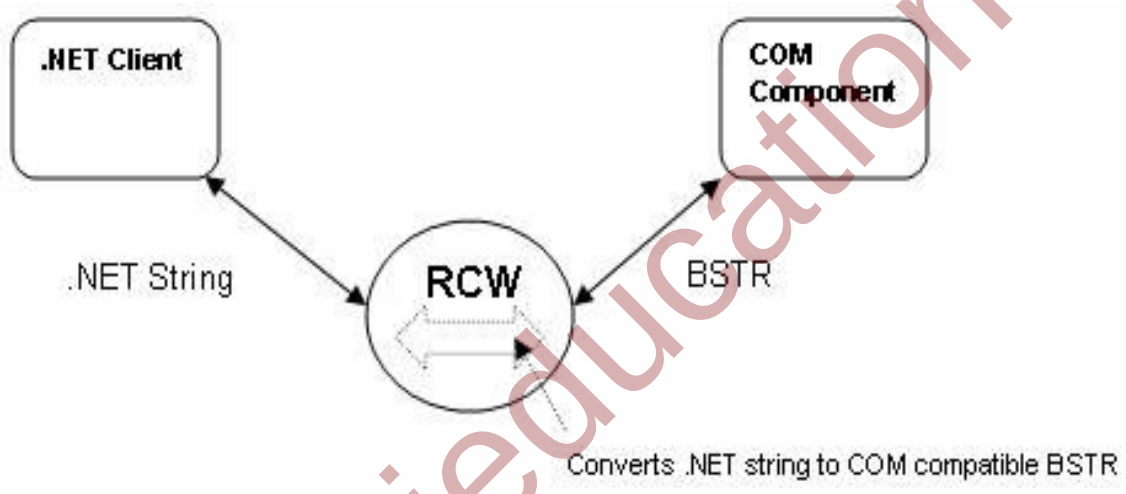
.NET Cons

- It's yet another platform to consider, which generally means rewriting and learning new tricks.
- Microsoft tends to have good ideas, but mediocre implementation.
- Currently, it's only available on Windows.

- Microsoft claims C#, IL, and CLR/CLS will be submitted to ECMA, but there's still no clear view on what will be standardized from the platform.

.NET Marshalling

Thus .NET runtime automatically generates code to translate calls between managed code and unmanaged code. While transferring calls between these two codes, .NET handles the data type conversion also. This technique of automatically binding with the server data type to the client data type is known as marshalling. Marshaling occurs between managed heap and unmanaged heap.



Sample diagram for marshaling

Logically the marshalling can be classified into 2 types.

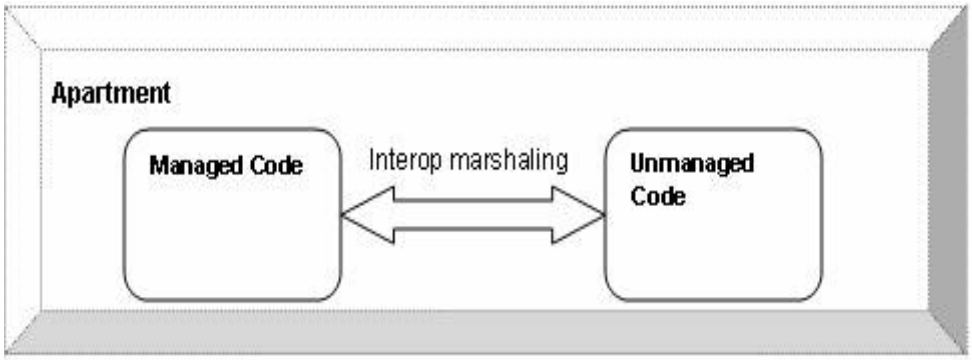
1. Interop marshalling
2. COM marshalling

If a call occurs between managed code and unmanaged code with in the same apartment, Interop marshaller will play the role. It marshals data between managed code and unmanaged code.

In some scenarios COM component may be running in different apartment threads. In those cases i.e., calling between managed code and unmanaged code in different apartments or process, both Interop marshaler and COM marshaler are involved.

Interop marshaler

When the server object is created in the same apartment of client, all data marshaling is handled by Interop marshaling.

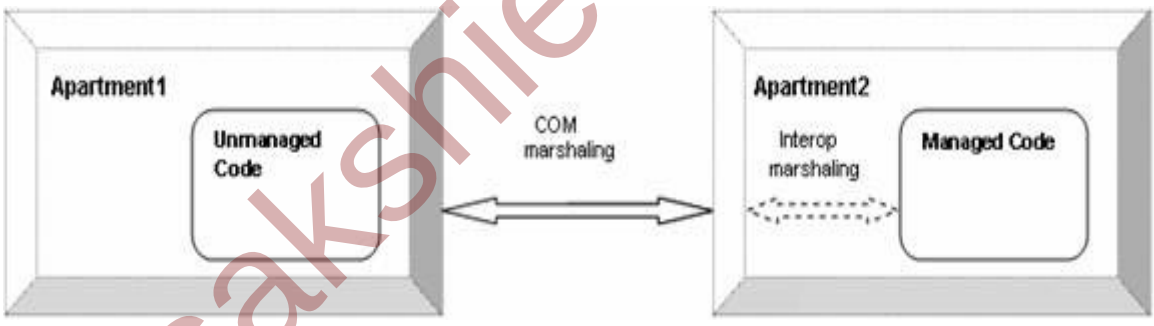


Sample diagram for same apartment marshalling

COM marshaler

COM marshaling involved whenever the calls between managed code and unmanaged code are in different apartments.

This kind of different apartment communication will impact the performance. The apartment settings of the managed client can be changed by changing the STAThreadAttribute / MTAThreadAttribute / Thread.ApartmentState property. Both the codes can run in a same apartment, by making the managed code's thread to STA. (If the COM component is set as MTA, then cross marshaling will occurs.)



Sample diagram for cross apartment marshalling

In the above scenario, the call with in different apartments will occur by COM marshaling and the call between managed and unmanaged code will occur by Interop marshaling.

Remoting in .Net

Introduction:

Distributed computing is an integral part of almost every software development. Before .Net Remoting, DCOM was the most used method of developing distributed application on Microsoft platform. Because of object oriented architecture, .Net Remoting replaces DCOM and .Net framework replaces COM.

Benefits of Distributed Application Development:

Fault Tolerance: Fault tolerance means that a system should be resilient when failures within the system occur.

Scalability: Scalability is the ability of a system to handle increased load with only an incremental change in performance.

Administration: .NET remoting is an architecture which enables communication between different application domains or processes using different transportation protocols, serialization formats, object lifetime schemes, and modes of object creation.

Remote means any object which executes outside the application domain. The two processes can exist on the same computer or on two computers connected by a LAN or the Internet. This is called marshalling (This is the process of passing parameters from one context to another.),

There are two basic ways to marshal an object:

- Marshal by value: the server creates a copy of the object passes the copy to the client. Marshal by reference: the client creates a proxy for the object and then uses the proxy to access the object.

Comparison between .NET Remoting and Web services:

S.No	ASP.Net WebService	.NET Remoting
1	Easy to develop and deploy	Involves complex programming
2	Accessed only over HTTP	Can be accessed over any of the protocol like HTTP,SMTP,TCP etc.,
3	Gives extensibility by allowing us to intercept the SOAP messages during the serialization and deserialization stages.	Highly extensible by allowing us to customize the different components of the .NET remoting framework.
4	Webservices support only the data types defined in the XSD type system, their by limiting the number of objects that can be serialized.	By Using binary communication, .NET Remoting can provide support for rich type system

For performance comparison between .Net Remoting and ASP.Net Web Services

Architecture:

Channels are Transport protocols for passing the messages between Remote objects. A channel is an object that makes communication between a client and a remote object, across app domain boundaries. The .NET Framework implements two default channel classes, as follows:

- HttpChannel: Implements a channel that uses the HTTP protocol.
- TcpChannel: Implements a channel that uses the TCP protocol (Transmission Control Protocol).

- Channel take stream of data and creates package for a transport protocol and sends to other machine.

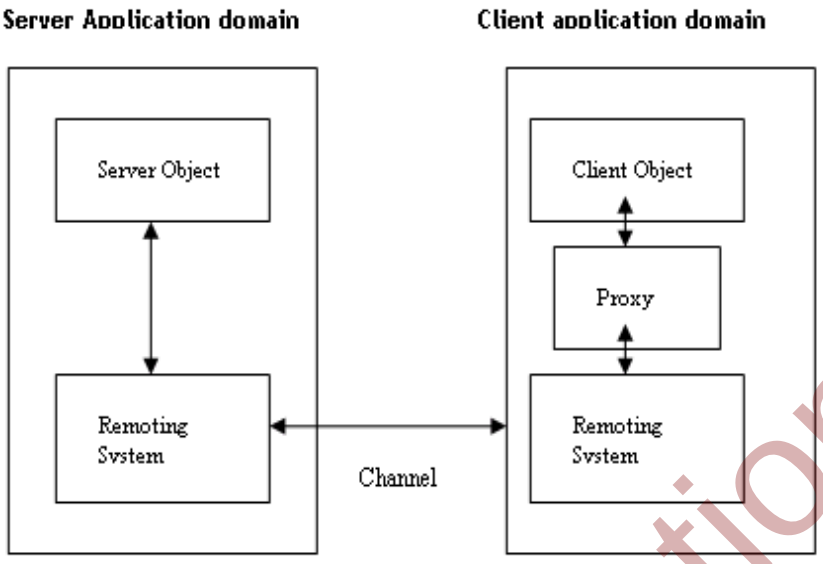


Fig. 1

A simple architecture of .NET remoting

Remoting system creates a proxy for the server object and a reference to the proxy will be returned to the client. When client calls a method, Remoting system sends request thro the channel to the server. Then client receives the response sent by the server process thro the proxy.

Example:

Let us see a simple example which demonstrates .Net Remoting. In This example the Remoting object will send us the maximum of the two integer numbers sent.

Creating Remote Server and the Service classes on Machine 1:
Please note for Remoting support your service (Remote object) should be derived from MarshalByRefObject.

```
////////////////////////////////////////////////////////////////////////////////////////////////////

using System;
using System.Runtime.Remoting.Channels; //To support and handle Channel and
channel sinks
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http; //For HTTP channel
using System.IO;

namespace ServerApp
{

public class RemotingServer
```

```
{
public RemotingServer()
{
//
// TODO: Add constructor logic here
//
}
}
//Service class
public class Service: MarshalByRefObject
{
public void WriteMessage (int num1,int num2)
{
Console.WriteLine (Math.Max(num1,num2));
}
}
//Server Class
public class Server
{
public static void Main ()
{
HttpChannel channel = new HttpChannel(8001); //Create a new channel
ChannelServices.RegisterChannel (channel); //Register channel
RemotingConfiguration.RegisterWellKnownServiceType(typeof(Service),"Service",WellK
nownObjectMode.Singleton);
Console.WriteLine ("Server ON at port number:8001");
Console.WriteLine ("Please press enter to stop the server.");
Console.ReadLine ();
}
}

}
```

//

Save the above file as ServerApp.cs. Create an executable by using Visual Studio.Net command prompt by,

csc /r:system.runtime.remoting.dll /r:system.dll ServerApp.cs

A ServerApp.Exe will be generated in the Class folder.

Run the ServerApp.Exe will give below message on the console

Server ON at port number:8001

Please press enter to stop the server.

In order to check whether the HTTP channel is binded to the port, type http://localhost:8001/Service?WSDL in the browser.
You should see a XML file describing the Service class.

Please note before running above URL on the browser your server (ServerApp.Exe should be running) should be ON.

Creating Proxy and the Client application on Machine 2

SoapSuds.exe is a utility which can be used for creating a proxy dll.

Type below command on Visual studio.Net command prompt.

soapsuds -url:http://< Machine Name where service is running>:8001/Service?WSDL -oa:Server.dll

This will generates a proxy dll by name Server.dll. This will be used to access remote object.

Client Code:

```
////////////////////////////////////////////////////////////////////////////////////////////////////
using System;
using System.Runtime.Remoting.Channels; //To support and handle Channel and channel sinks
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http; //For HTTP channel
using System.IO;
using ServerApp;

namespace RemotingApp
{

public class ClientApp
{
public ClientApp()
{

}

public static void Main (string[] args)
{
HttpChannel channel = new HttpChannel (8002); //Create a new channel
ChannelServices.RegisterChannel (channel); //Register the channel
//Create Service class object
Service svc = (Service) Activator.GetObject (typeof (Service),"http://<Machine name where Service running>:8001/Service"); //Localhost can be replaced by
//Pass Message
svc.WriteMessage (10,20);
}
}
}
////////////////////////////////////////////////////////////////////////////////////////////////////
```

Save the above file as ClientApp.cs. Create an executable by using Visual Studio.Net

command prompt by,
csc /r:system.runtime.remoting.dll /r:system.dll ClientrApp.cs
A ClientApp.Exe will be generated in the Class folder. Run ClientApp.Exe , we can see the result on Running ServerApp.EXE command prompt.
In the same way we can implement it for TCP channel also.