**Client activated object**

Client-activated objects (CAO) are server-side objects that are activated upon request from the client. When the client submits a request for a server object using a "new" operator, an activation request message is sent to the remote application. The server then creates an instance of the requested class and returns an ObjRef back to the client application that invoked it. A proxy is then created on the client-side using the ObjRef. The client's method calls will be executed on the proxy. Client-activated objects can store state information between method calls for their specific client, and not across different client objects. Each invocation of "new" returns a proxy to an independent instance of the server type.

When a call is made between objects in the same Application Domain, only a normal local call is required; however, a call across Application Domains requires a remote call. In order to facilitate a remote call, a proxy is introduced by the .NET framework at the client-side. This proxy is an instance of the TransparentProxy class, directly available to the client to communicate with the remote object. Generally, a proxy object is an object that acts in place of some other object. The proxy object ensures that all calls made on the proxy are forwarded to the correct remote object instance. In .NET Remoting, the proxy manages the marshaling process and the other tasks required to make cross-boundary calls. The .NET Remoting infrastructure automatically handles the creation and management of proxies.

Object Marshalling specifies how a remote object is exposed to the client application. It is the process of packaging an object access request in one application domain and passing that request to another domain. The .NET Remoting infrastructure manages the entire marshaling process. There are two methods by which a remote object can be made available to a local client object: Marshal by value, and Marshal by reference.

Marshal by value is analogous to having a copy of the server object at the client, and Marshaling by reference is analogous to having a pointer to the object.

**Server**

Here, the server is created for the client.

**using System;**

**using System.Runtime.Remoting;**

**public class Server{**

**public static void Main(string[] Args){**

**RemotingConfiguration.Configure("server.exe.config",false);//**configuration of server object

**Console.WriteLine("The server is listening. Press Enter to exit....");**

**Console.ReadLine();**

**Console.WriteLine("GC'ing.");**

**GC.Collect();**

**GC.WaitForPendingFinalizers();**

**}**

**}**

Garbage collection (GC) is a new feature in Microsoft .net framework. When we have a class that represents an object in the runtime that allocates a memory space in the heap memory. All the behavior of that objects can be done in the allotted memory in the heap. Once the activities related to that object are get finished then it will be there as unused space in the memory.

**Client activated object**

These are created on the server immediately upon the client's request. An instance of a Client Activated Object is created every time the client instantiates one.

**using System;**

**using System.Runtime.Remoting;**

**public class Client{**

**public static void Main(string[] Args){**

**RemotingConfiguration.Configure("CAOclient.exe.config",false); //**configuration of Client object

**ClientActivatedType CAObject = new ClientActivatedType();**

**Console.WriteLine("Client-activated object: " + CAObject.RemoteMethod());**

**}**

**}**

CAObject created for a client from *ClientActivatedType* class from RemoteType, which has local proxy instant, then the proxy works as per discussion.

**Object configuration**

The server object is configured as here below at port number 8080, activated type set to ClientActivatedType, RemoteType.

<configuration>

<system.runtime.remoting>

<application>

<service>

<activated type="ClientActivatedType, RemoteType"/>

</service>

<channels>

<channel port="8080" ref="http"/>

</channels>

</application>

</system.runtime.remoting>

</configuration>

The Client Object is configurated to the localhost port 8080, and activated type set to “ClientActivatedType, RempteType”

<configuration>

<system.runtime.remoting>

<application>

<client url="http://localhost:8080">

<activated type="ClientActivatedType, RemoteType"/>

</client>

</application>

</system.runtime.remoting>

</configuration>

Here is a remote method that is being requested by the client.

**using System;**

**using System.Security.Principal;**

**public class ClientActivatedType : MarshalByRefObject{**

**public string RemoteMethod(){**

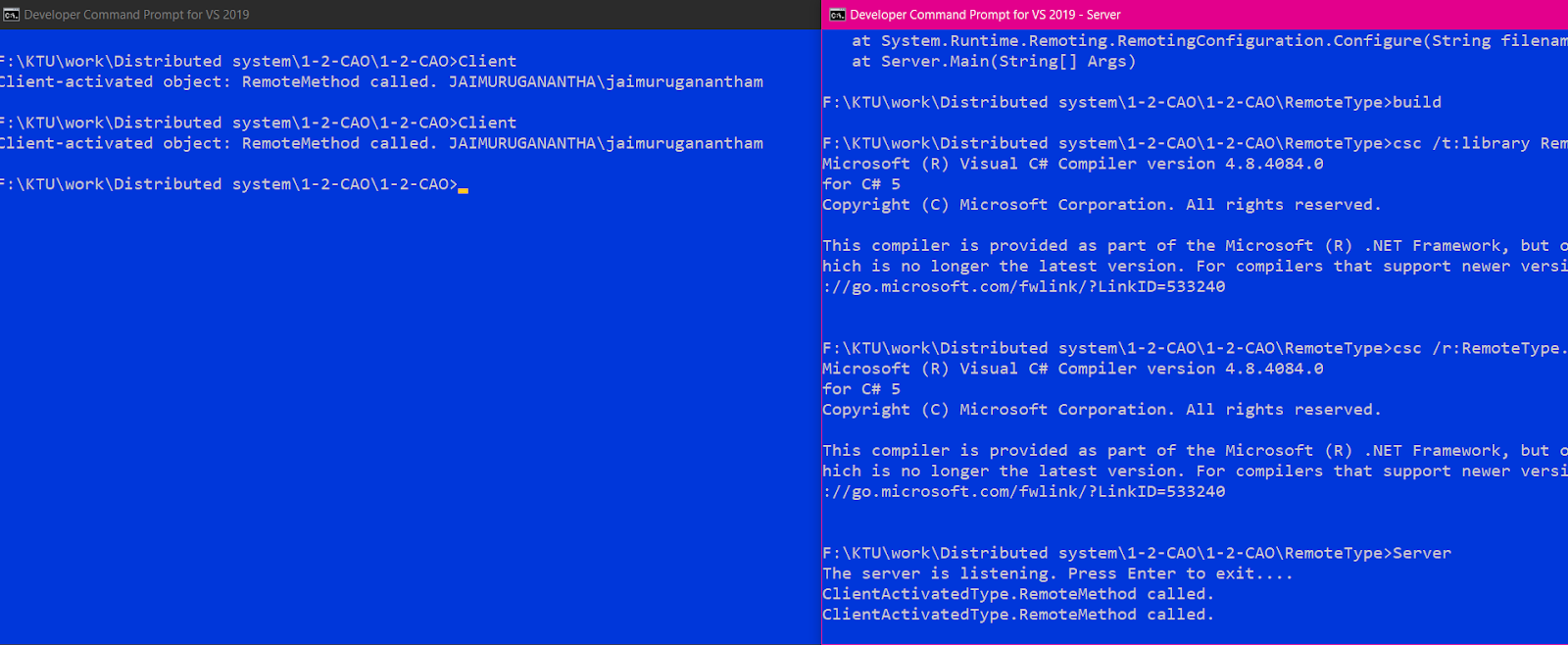
**Console.WriteLine("ClientActivatedType.RemoteMethod called.");**

**return "RemoteMethod called. " + WindowsIdentity.GetCurrent().Name;**

**}**

**}**

**Output of CAO**

****

**Service-oriented architecture**

In this architecture, we are creating services in a server and using them in a client.

To begin with, we are creating a server method to start it.

First of all, we are using System libraries to run

System Libraries are below

>>**using System;**

>>**using System.Runtime.Remoting;**/\*Provides classes and interfaces that allow developers to create and configure distributed applications\*/

>>**using System.Runtime.Remoting.Channels;**/\*Contains classes that support and handle channels and channel sinks, which are used as the transport medium when a client calls a method on a remote object.\*/

>>**using System.Runtime.Remoting.Channels.Tcp;**/\*Contains channels that use the TCP protocol to transport messages and objects to and from remote locations. By default, the TCP channels encode objects and method calls in binary format for transmission, but other encoding and decoding formatter sinks can be specified in the configuration properties of a channel.\*/

>>**using System.Runtime.Remoting.Channels.Http;**/\*Contains channels that use the HTTP protocol to transport messages and objects to and from remote locations. By default, the HTTP channels encode objects and method calls in SOAP format for transmission, but other encoding and decoding formatter sinks can be specified in the configuration properties of a channel.\*/

The code to build the server is here below.

namespace RemotingSamples

{

public class HelloServer : MarshalByRefObject

{

public int callCounter = 0;

public static int Main(string [] args)

{

TcpChannel chan1 = new TcpChannel(8085);

HttpChannel chan2 = new HttpChannel(8086);

ChannelServices.RegisterChannel(chan1);

ChannelServices.RegisterChannel(chan2);

RemotingConfiguration.RegisterWellKnownServiceType

(

typeof(HelloServer),

"SayHello",

WellKnownObjectMode.Singleton

);

/\*RemotingConfiguration.RegisterWellKnownServiceType(

typeof(HelloServer),

"SayHello",

WellKnownObjectMode.SingleCall

);

\*/

System.Console.WriteLine("Press Enter key to exit");

System.Console.ReadLine();

return 0;

}

public HelloServer()

{

Console.WriteLine("HelloServer activated");

}

public String HelloMethod(String name,

out int counter)

{

counter = ++callCounter;

Console.WriteLine(

"Server Hello.HelloMethod : {0} Counter :{1}",

name, callCounter);

return "Hi there " + name;

}

}

}

**Explanation of code**

First, we are creating and registering the channels for TCP and HTTP,

**TcpChannel chan1 = new TcpChannel(8085);**

**HttpChannel chan2 = new HttpChannel(8086);**

**ChannelServices.RegisterChannel(chan1);**

**ChannelServices.RegisterChannel(chan2);**

RemotingConfiguration.RegisterWellKnownServiceType Holds values for an object type registered on the service end as a server-activated type object (single call or singleton).

Single call is a method where objects are created every time new when the client is calling server to get service,

**RemotingConfiguration.RegisterWellKnownServiceType(**

**typeof(HelloServer),**

**"SayHello",**

**WellKnownObjectMode.SingleCall**

**);**

Singleton is a method where objects are not created every time instead it takes as the already created objects.

**RemotingConfiguration.RegisterWellKnownServiceType**

**(**

**typeof(HelloServer),**

**"SayHello",**

**WellKnownObjectMode.Singleton**

**);**

Here we are creating a constructor to inform Server is activated

**public HelloServer()**

**{**

**Console.WriteLine("HelloServer activated");**

**}**

Here we are creating a method to inform object name, and how many times it is created.

**public String HelloMethod(String name,**

**out int counter)**

**{**

**counter = ++callCounter;**

**Console.WriteLine(**

**"Server Hello.HelloMethod : {0} Counter :{1}",**

**name, callCounter);**

**return "Hi there " + name;**

**}**

Now we are going to build a client to use method from server.

using System;

using System.Runtime.Remoting;

using System.Runtime.Remoting.Channels;

using System.Runtime.Remoting.Channels.Tcp;

using System.Runtime.Remoting.Channels.Http;

using System.IO;

namespace RemotingSamples

{

public class Client

{

public static int Main(string [] args)

{

int counter;

TcpChannel chan1 = new TcpChannel();

ChannelServices.RegisterChannel(chan1);

HelloServer obj1 = (HelloServer)Activator.GetObject(

typeof(RemotingSamples.HelloServer),

"tcp://localhost:8085/SayHello");

if (obj1 == null)

{

System.Console.WriteLine(

"Could not locate TCP server");

return 1;

}

HttpChannel chan2 = new HttpChannel();

ChannelServices.RegisterChannel(chan2);

HelloServer obj2 =(HelloServer)Activator.GetObject(

typeof(RemotingSamples.HelloServer),

"http://localhost:8086/SayHello");

if (obj2 == null)

{

System.Console.WriteLine(

"Could not locate HTTP server");

return 1;

}

try

{

Console.WriteLine(

"Client1 TCP HelloMethod {0} Counter {1}",

obj1.HelloMethod("jaimuruganantham", out counter),

counter);

Console.WriteLine(

"Client2 HTTP HelloMethod {0} Counter {1}",

obj2.HelloMethod("Jaimuruganantham", out counter),

counter);

}

catch (IOException ioExcep)

{

Console.WriteLine("Remote IO Error" +

"\nException:\n" + ioExcep.ToString());

return 1;

}

return 0;

}

}

}

We are creating objects like here

**HelloServer obj2 =(HelloServer)Activator.GetObject(**

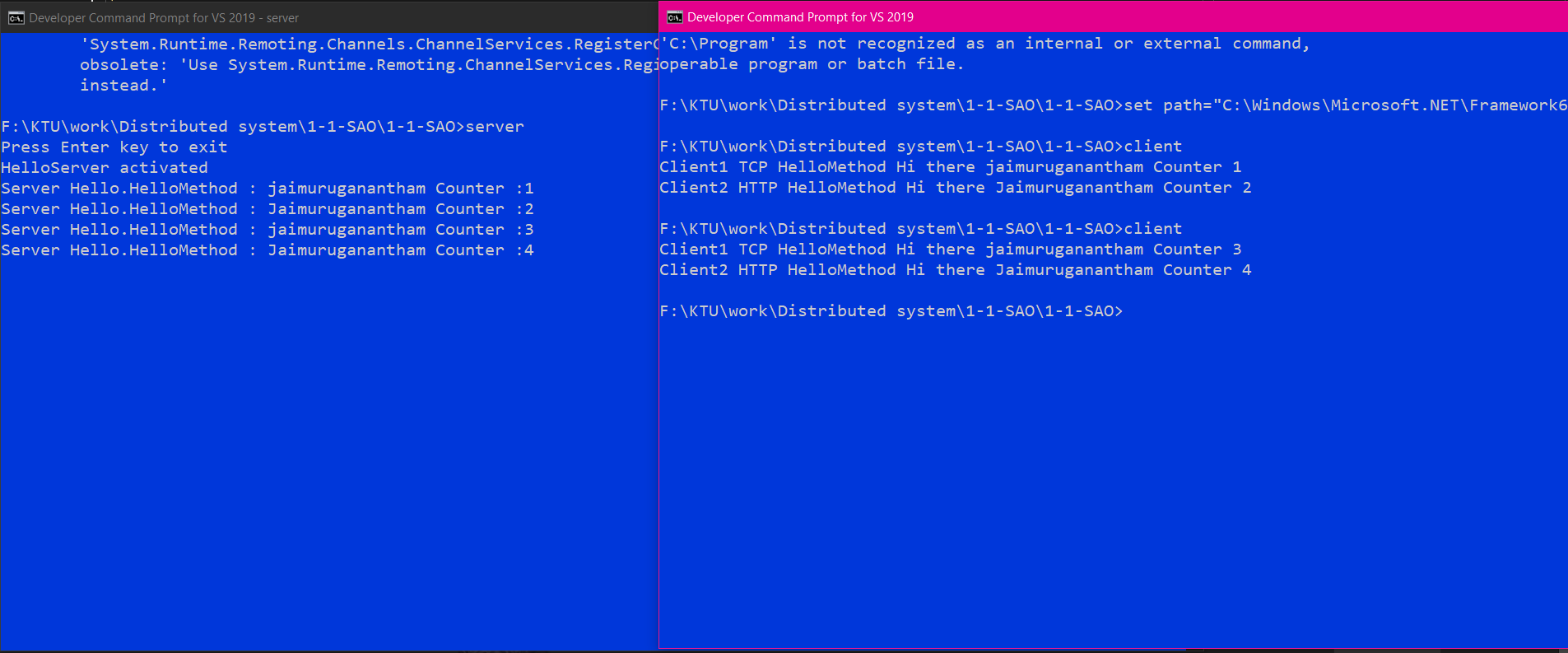
**typeof(RemotingSamples.HelloServer),**

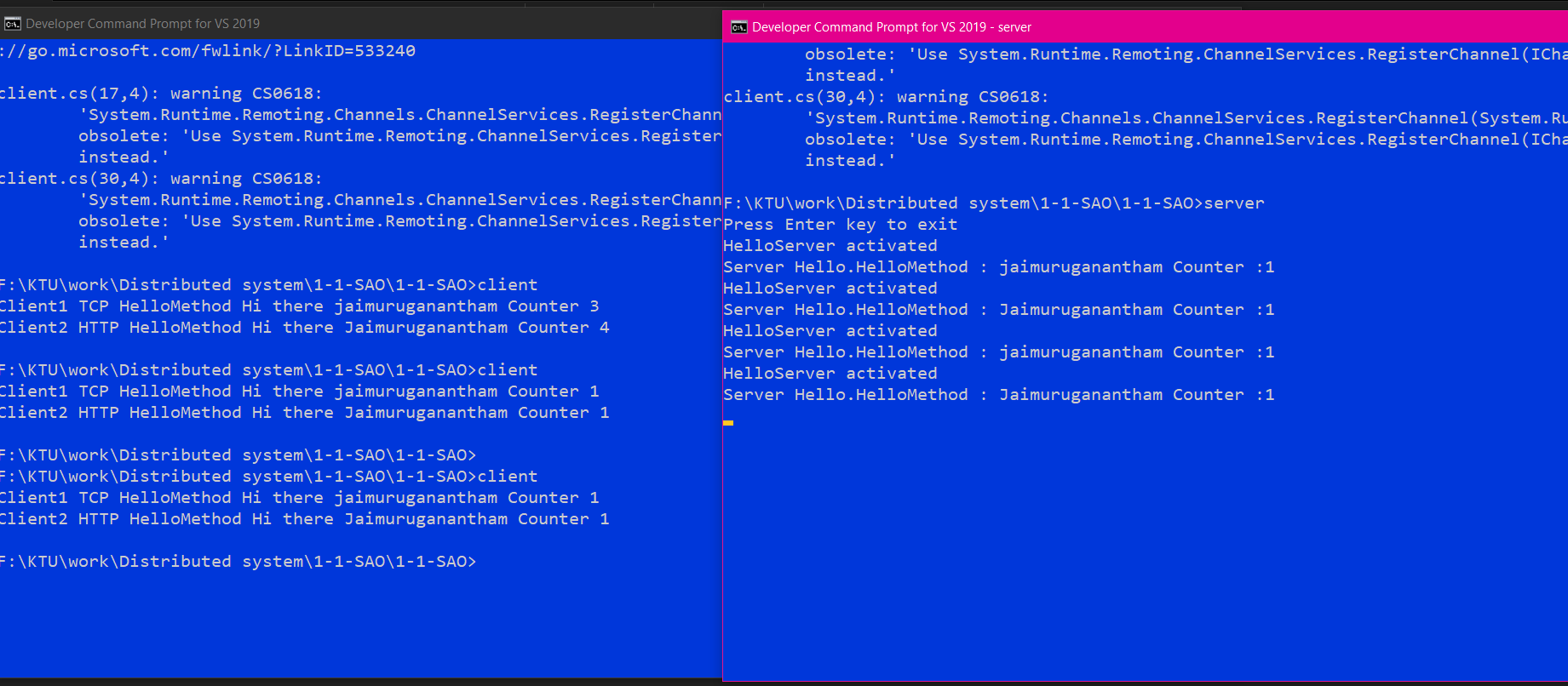
**"http://localhost:8086/SayHello");**

Obj2 created and Activator.GetObject is having parameter type-of and localhost server ID.

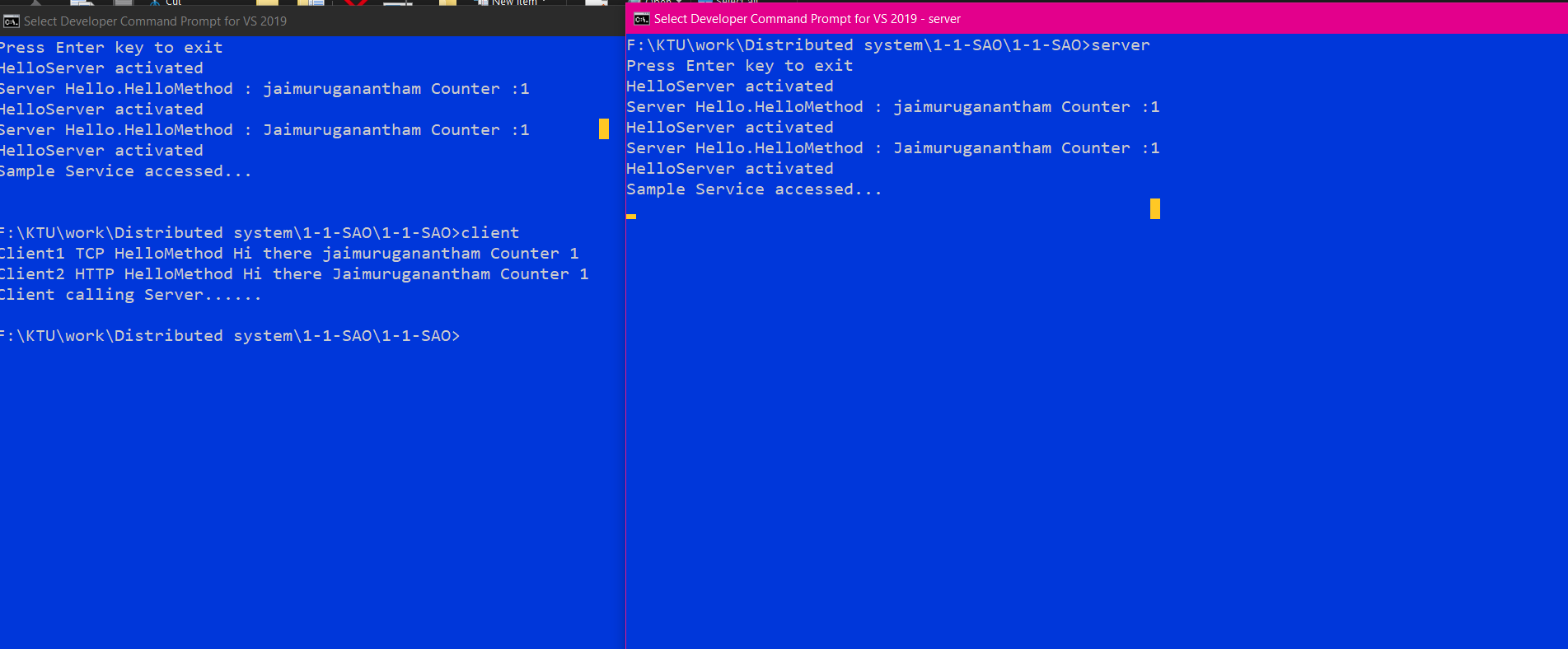
After creating obj2, we are accessing the server method in the client.

Here we are created object with singleton



Here we are created object with single call

Sample service created in Server and accessed in Client



**Object marshaling**

As from the previous explanation, the client is used object marshaling to get currency exchange information from the server,

**Server**

using System;

using System.Runtime.Remoting;

namespace Server

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Configuring remoting");

RemotingConfiguration.Configure("Server.exe.config", true);// **Server configuration file**

Console.WriteLine("Server ready ");

Console.ReadLine();

}

}

}

**Client**

using System;

using System.Runtime.Remoting;

namespace Client

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Configuring remoting");

RemotingConfiguration.Configure("Client.exe.config", true);// **Client configuration file**

Console.WriteLine("Press ENTER to start");

Console.ReadLine();

try

{

/**/currStats object created from Common.ICurrencyStatistics to get currency information**

Common.ICurrencyStatistics currStats;

currStats = (Common.ICurrencyStatistics)Activator.GetObject(

typeof(Common.ICurrencyStatistics),

"tcp://localhost:9001/CurrencyStatistics.rem"

);

**// here r object created from Common.CurrencyRange to get the currency range of a particular currency.**

Common.CurrencyRange r = currStats.MaxMinPeriod(

"USD", "GBP", new DateTime(2021, 1, 1), new DateTime(2021, 7, 1)

);

r.Trace();

}

catch (Exception ex)

{

Console.WriteLine("Exception caught in client: {0}", ex.Message);

}

Console.WriteLine("Press ENTER to terminate client");

Console.ReadLine();

}

}

}

**ICurrencyStatistiscs**

using System;

using System.Collections.Generic;

using System.Text;

namespace Common

{

/// <summary>

/// This interface defines how currency statistics information is transferred

/// between server and client.

/// </summary>

/// <remarks></remarks>

public interface ICurrencyStatistics

{

/// <summary>

/// Get the maximum and minimum exchange rate between two currencies over a specified period.

/// </summary>

/// <param name="currencyFrom">First currency code</param>

/// <param name="currencyTo">Second currency code</param>

/// <param name="dateStart">Start of period</param>

/// <param name="dateEnd">End of period</param>

/// <returns></returns>

CurrencyRange MaxMinPeriod(

string currencyFrom, // **source currency**

string currencyTo// **Destination currency**

DateTime dateStart,// **Starting datte**

DateTime dateEnd//**Ending Date**

);

}

}

**CurrencyRange**

using System;

using System.IO;

namespace Common

{

[Serializable]

public class CurrencyRange

// : MarshalByRefObject

{

// Private storage for this class; all methods accessed by public properties

private string \_currencyFrom;

private string \_currencyTo;

private DateTime \_dateStart;

private DateTime \_dateEnd;

private Decimal \_minRate;

private Decimal \_maxRate;

public string CurrencyFrom

{

get

{

return \_currencyFrom;**// Source currency**

}

set

{

\_currencyFrom = value;**// Source Currency value**

}

}

public string CurrencyTo

{

get

{

return \_currencyTo;**// Destination Currency**

}

set

{

\_currencyTo = value;**// Destination Currency value**

}

}

public DateTime DateStart

{

get

{

return \_dateStart; **// starting date**

}

set

{

\_dateStart = value;// **Starting Date value**

}

}

public DateTime DateEnd

{

get

{

return \_dateEnd;**// Ending date**

}

set

{

\_dateEnd = value;**// Ending date value**

}

}

public Decimal MinRate

{

get

{

return \_minRate;

}

set

{

\_minRate = value;

}

}

public Decimal MaxRate

{

get

{

return \_maxRate;

}

set

{

\_maxRate = value;

}

}

public CurrencyRange()

{

Console.WriteLine("TRACE: Range constructor");

}

**// Trance value of currency are displayed**

public void Trace()

{

Console.WriteLine(

"TRACE: From: {0} To: {1} Start: {2} End {3} Min: {4}, Max: {5} ",

CurrencyFrom, CurrencyTo, DateStart, DateEnd, MinRate, MaxRate

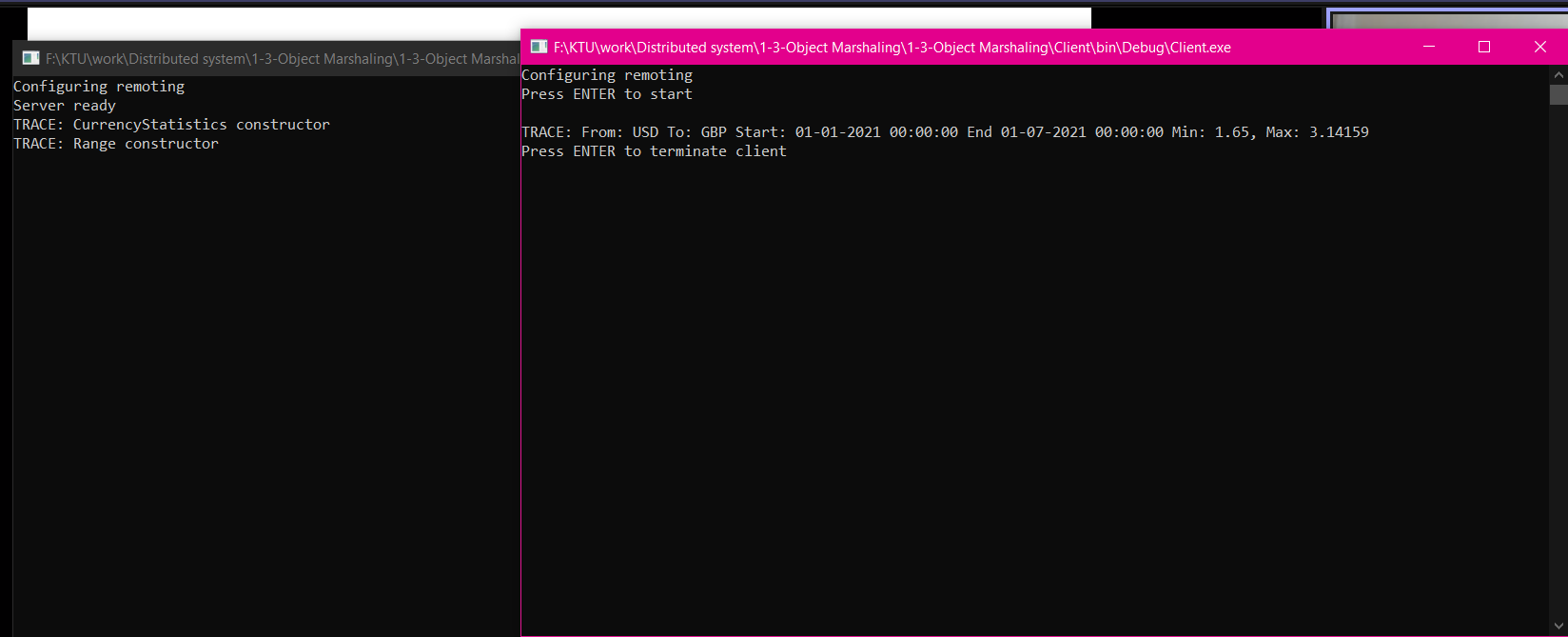
);

}

}

}

**OUTPUT**

****