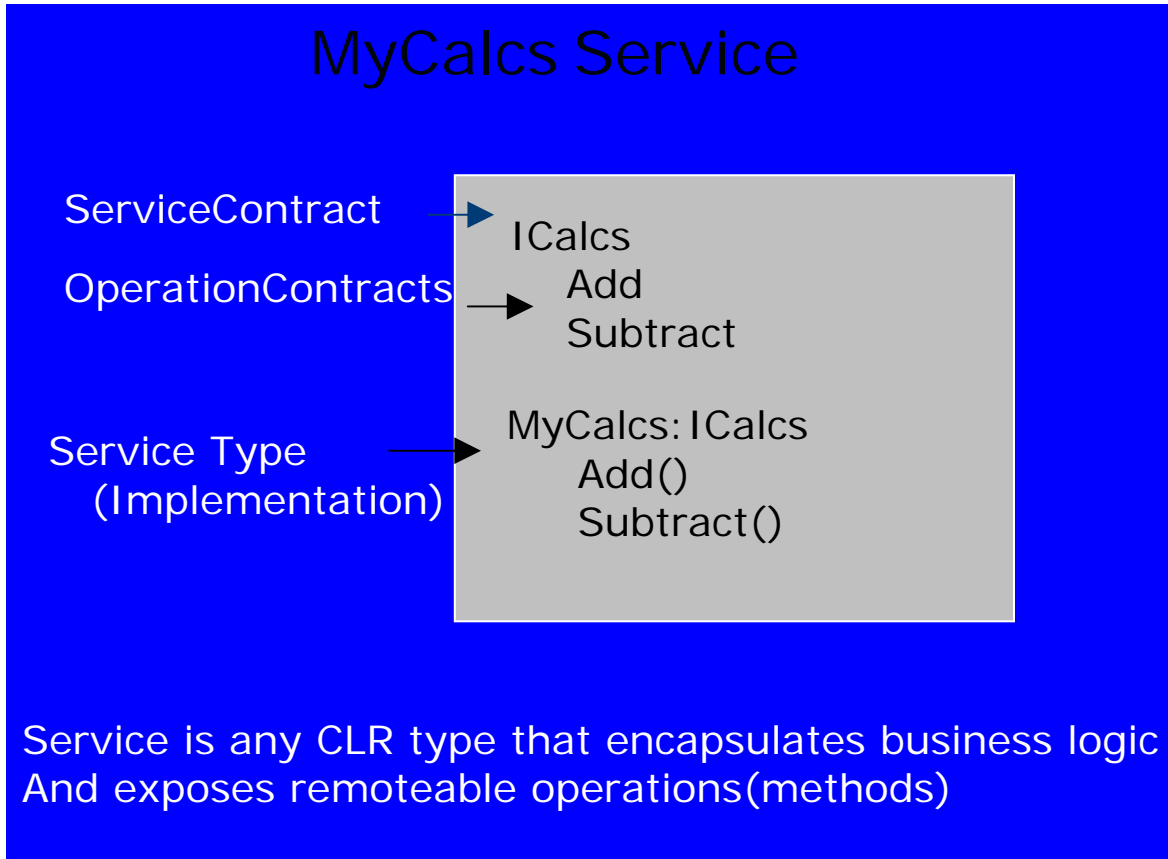


Developing WCF Applications

We are going to implement a simple WCF service called MyCalcs that will expose an Add and Subtract method as shown below.



The first step is to create a new Class Library which we can call CalcsService. Rename the Project and Class to CalcsContract and add a reference and using for System.ServiceModel. A service contract is defined by applying the ServiceContractAttribute to a class or interface. In this example add an interface called IMyCalcs with two methods called Add and Subtract. The service contract is then the behavior that our clients program against. The result should look like:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ServiceModel;
namespace CalcsService
{
    [ServiceContract()]
    public interface IMyCalcs
    {
        [OperationContract]
        int Add(int value1, int value2);
        [OperationContract]
        int Subtract(int value1, int value2);
    }
}
```

```
}
```

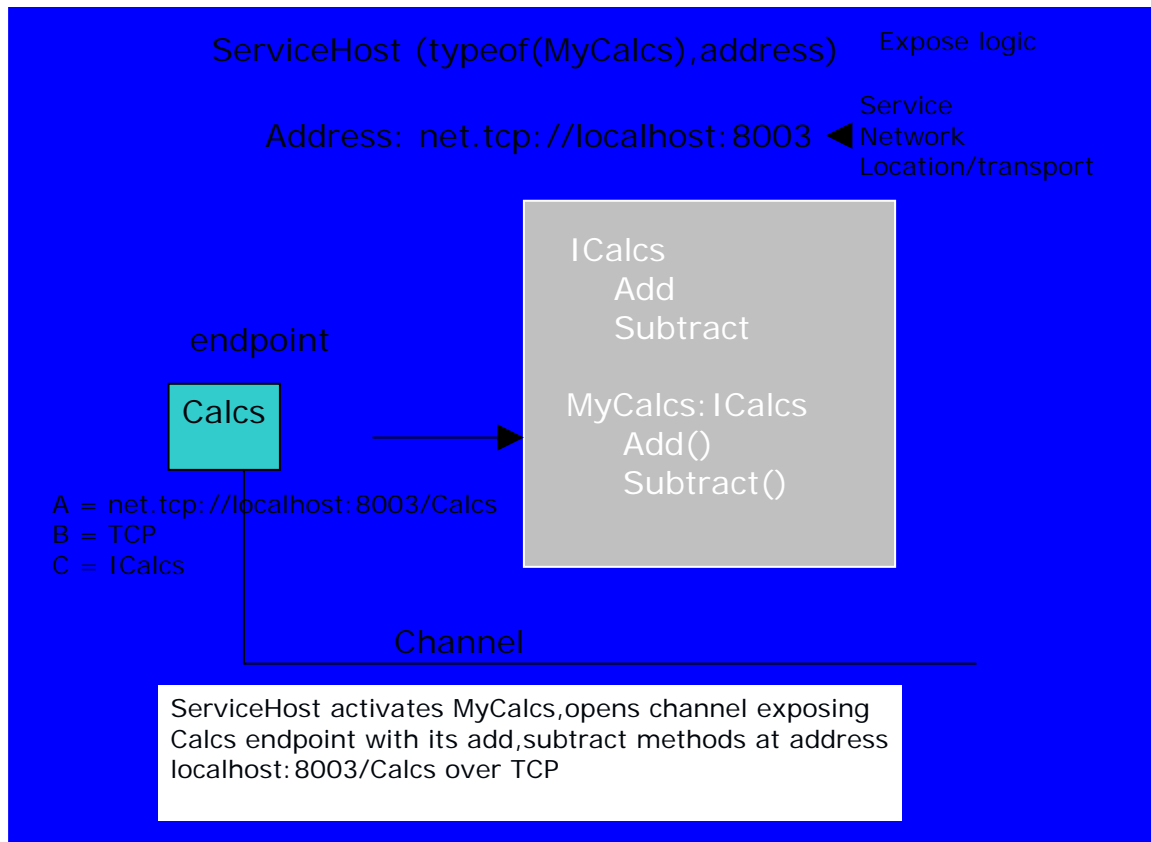
```
}
```

Like working with any Interface we must now define an implementation of the IMyCalcs. Note we are free to change this implementation at anytime as long as the contract remains intact, otherwise client code can be broken. Actually WCF will let you add and remove parms but this is not real good practice. Now add a new Class Library to the solution called CalcsImplementation. Add a project Reference for CalcsContract (Add reference.Projects Tab). Define a class called MyCalcs that implements the add and subtract methods of IMyCalcs. Seperating the Contract and implementation is a matter of style and flexibility but there is no reason they cannot be in the same project.The code should look like:

```
namespace CalcsService
{
    public class MyCalcs : IMyCalcs
    {
        public int Add(int Value1, int Value2)
        {
            return Value1 + Value2;
        }

        public int Subtract(int Value1, int Value2)
        {
            return Value2 - Value1;
        }
    }
}
```

We have now defined our MyCalcs WCF service that will expose 2 methods for remote consumption. Web services can be activated on the fly by IIS but alas the same is not true for WCF services that use transports other than HTTP. In this case we want to use TCP binary as our service will run exclusively inside the firewall. Thus MyCalcs is really a remoteable object . To do this we must provide a Host program for the service , intially we will chose a Console app which will serve our debugging and testing needs. The following diagram shows the key elements:



The Host must use the ServiceHost to implement our MyCalcs service type so its operations are exposed through one or more EndPoints. I will first show how to do this programmatically as I believe it is important to understand the players in this process. The serviceHost will use ABC's to define endpoints where A is the address(URI) that the service is available on the network, B is the Binding and C is the Contract. We already know about the C as it is ICalcs. Binding has many options that are beyond the scope of this document and we will only specify that our transport is TCP. Our address is as shown above.

The ServiceHost is instantiated as follow:

```
ServiceHost host = new ServiceHost(typeof(MyCalcs), new
Uri("net.tcp://localhost:8003"));
```

We then add a service endpoint using the above ABCs

```
host.AddServiceEndpoint(typeof(CalcsService.IMyCalcs), new NetTcpBinding(),
"Calcs");
```

and then open the channel for the service to the enterprise exposing its methods:

```
host.Open();
```

Note: We specified a base address on the ServiceHost so the address to reach our Calcs business logic is net.tcp://localhost:8002/Calcs

The code should look like:

```

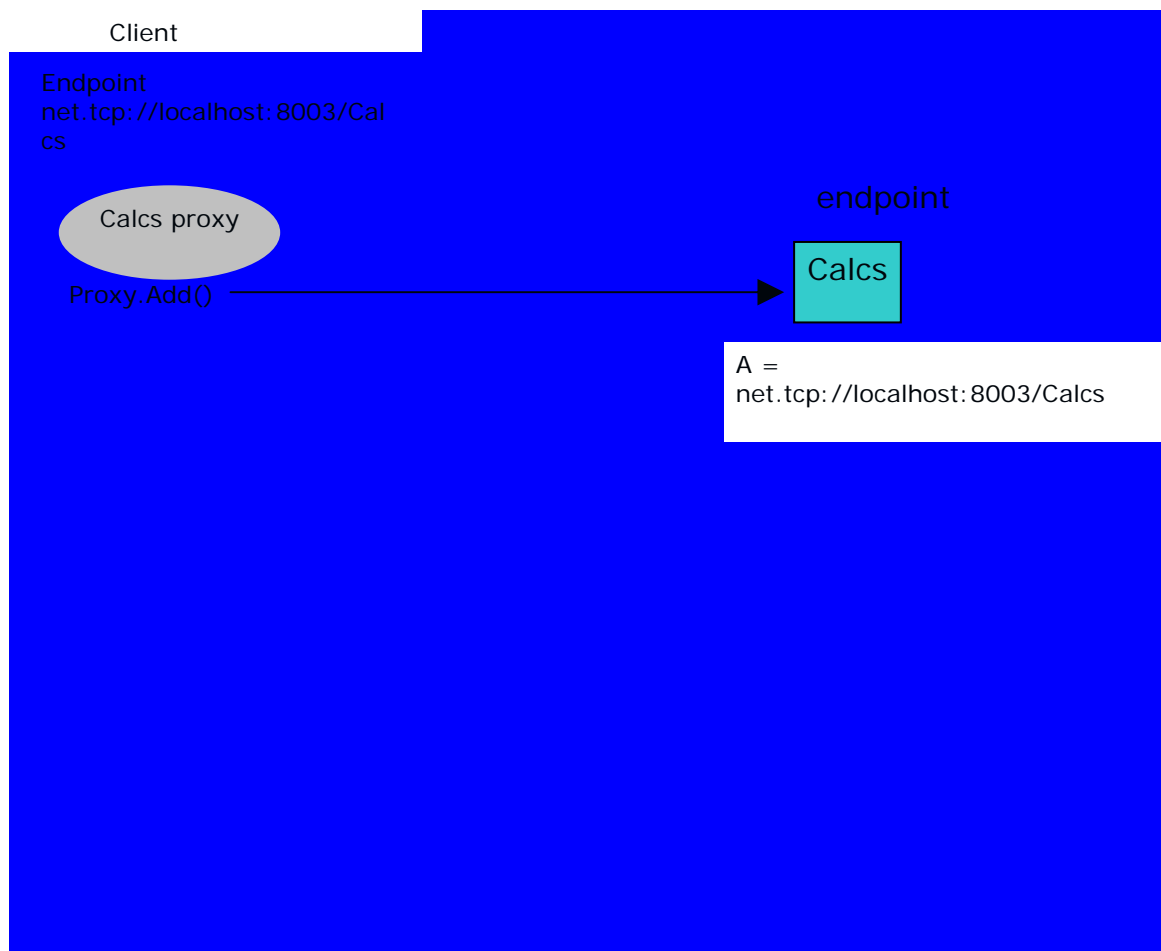
using (ServiceHost host = new ServiceHost(typeof(CalcsService.MyCalcs), new
Uri("net.tcp://localhost:8002")))
{
    host.AddServiceEndpoint(typeof(CalcsService.IMyCalcs), new
NetTcpBinding(), "Calcs");
    host.Open();    //open the channel

    Console.WriteLine("Press <ENTER> to terminate the service host");
    Console.ReadLine();    //keep the service running
}

```

Now we define the Client that will use the Calcs Clients do not directly call methods of the service but invoke them through a proxy.

Client diagram:



The programmatic way to do this is through `ChannelFactory<T>` which creates a proxy for service contract T. One must first provide Endpoint information to the `ChannelFactory` as follows:

```

EndpointAddress ep = new EndpointAddress( "net.tcp://localhost:8003/Calcs" );

```

Then a proxy is created

```
IMyCalcs proxy = ChannelFactory<IMyCalcs>.CreateChannel(new NetTcpBinding(), ep);
```

But we also must have a definition for the service contract **IMyCalcs** add a reference to the CalcsService assembly that contains the interface and then add the following:

```
using CalcsService;
```

Note: This is an advantage of having the service contract and implementation being in separate projects. Architectureally it is preferred but as usual a little more work.

Operations (method) can then be called on the proxy as in:

```
int test = proxy.Add(1, 2);
```

The code should look like:

```
static void Main(string[] args)
{
    EndpointAddress ep = new
        EndpointAddress("net.tcp://localhost:8003/Calcs");
    IMyCalcs proxy = ChannelFactory<IMyCalcs>.CreateChannel(new
        NetTcpBinding(), ep);
    using (proxy as IDisposable)
    {
        int test = proxy.Add(1, 2);
        Console.WriteLine(test);
    }

    Console.WriteLine("Press <ENTER> to terminate Client.");
    Console.ReadLine();
}
```

Often you will need to specify the identity to the endpoint and this can be done:

```
EndpointIdentity endpointIdentity =
    EndpointIdentity.CreateUpnIdentity(WindowsIdentity.GetCurrent().Name);
```

```
EndpointAddress ep = new EndpointAddress(new
    Uri("net.tcp://cd423:8001/Logger"), endpointIdentity);
```

Declarative with App.Config

The code shown above is not overly complex but for more flexibility the same service Model properties can be delivered through config settings. The first thing to-do is add a config file to the ConsoleHost project then add the following tag:

```
<system.serviceModel>
```

Then as there can be multiple services defined in the project we add the services tag

```
<services>
```

For each service we need a <service> tag whose name is the service type (MyCalcs) . This corresponds to the `ServiceHost` host = `new ServiceHost(typeof(MyCalcs), new`

`Uri("net.tcp://localhost:8003"))` C# code for the programmatic solution above. So our tag looks like:

```
<service name=" CalcsService.MyCalcs">
```

This name must match the

Now we define the endpoint using the same ABC's as above:

```
<endpoint address=" net.tcp://localhost:8003/Calcs" binding="netTcpBinding"
contract="MyCalcsContracts.IMyCalcs" />
```

Figure 1.3

Our App.config now looks like:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="CalcsService.MyCalcs">
        <endpoint address="net.tcp://localhost:8003/Calcs"
binding="netTcpBinding" contract="CalcsService.IMyCalcs"></endpoint>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

We specified the full Uri for the address in the endpoint tag but we could have written these differently and more flexibly by adding a <host> section as follows:

```
:
<host>
  <baseAddresses>
    <add baseAddress="net.tcp://localhost:8003/" />
  </baseAddresses>
</host>
```

Then the <endpoint> tag looks like:

```
<endpoint address="Calcs" binding="netTcpBinding"
contract="CalcsService.IMyCalcs"></endpoint>
```

And the config file now looks like:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="CalcsService.MyCalcs">
        <endpoint address="Calcs" binding="netTcpBinding"
contract="CalcsService.IMyCalcs"></endpoint>
      <host>
        <baseAddresses>
          <add baseAddress="net.tcp://localhost:8003/" />
        </baseAddresses>
      </host>
    </service>
  </services>
</system.serviceModel>
</configuration>
```

The code in Program.cs that hosts the service can then be changed to

```
ServiceHost host = new ServiceHost(typeof(CalcsService.MyCalcs));
host.Open();
```

We have moved the configuration into the config file which is always good design and the same client can then be used to access the service.

Service Reference

Visual Studio with .NET 3.0 extensions has an Add Service Reference that can be used to make this process even simpler. But first we must add some functionality to the Host so that we publish metadata describing the methods in our service using a system supplied service contract called IMetadataExchange. This will enable the Add Service reference to create the client proxy for us without writing any code ourselves.

The first thing we do expose a metadata endpoint which will use the following ABCs:

- A MEX (metadata exchange) ..this is appended to host Uri so absolute Uri is net.tcp://localhost:8003/MEX
- B mexTcpBinding
- c IMetadataExchange

The endpoint will then look like:

```
<endpoint binding="mexTcpBinding" bindingConfiguration="" address="MEX"
          contract="IMetadataExchange" />
```

However this not quite enough as the service must be told to make use of this endpoint through a **behavior**, these modify the way messages are processed as they flow through the channel stack. Thus our service will process requests for metadata from clients using this system supplied endpoint. So we add the following behavior definition as a child of <system.servicemodel>

```
<behaviors>
  <serviceBehaviors>
    <behavior name="MyBehavior">
      <serviceMetadata />
    </behavior>
  </serviceBehaviors>
</behaviors>
```

We also have to tell the service to use the behavior by add the behaviorConfiguration="MyBehavior" attribute to the <service> tag so it looks like:

```
<service behaviorConfiguration="MyBehavior"
name="MyCalcsImplementation.MyCalcs">
```

The config file should now look like:

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior name="MyBehavior">
          <serviceMetadata />
        </behavior>
      </serviceBehaviors>
    </behaviors>
    <services>
      <service behaviorConfiguration="MyBehavior"
name="CalcsService.MyCalcs">
        <endpoint address="Calcs" binding="netTcpBinding"
contract="CalcsService.IMyCalcs"></endpoint>
        <endpoint address="MEX" binding="mexTcpBinding"
contract="IMetadataExchange"></endpoint>
      <host>
        <baseAddresses>
          <add baseAddress="net.tcp://localhost:8003/" />
        </baseAddresses>
```

```

        </host>
    </service>
</services>
</system.serviceModel>

```

This also shows how to add new endpoints to the service thus exposing new functionality. Then whenever a client wants to consume our service we only need to use the Add Service Reference to generate the proxy for use. Go to the Client project and from Solution Explorer right-click on the Client project node and select Add Service Reference. The dialog presented requires you to provide a valid base address to the service. Supply the base address for the MyCalcs, net.tcp://localhost:8003, and as I prefer change the reference name to something like SuperCalc . Now the client code can be simplified to

```

static void Main(string[] args)
{
    SuperCalc.MyCalcsClient proxy = new SuperCalc.MyCalcsClient();
    int test = proxy.Add(1, 2);
    Console.WriteLine(test);
    Console.WriteLine("Press <ENTER> to terminate Client.");
    Console.ReadLine();
}

```

Configuring service endpoints using the Service Configuration Editor

The generation of the config file is pretty straightforward as long as one understands the process but there is also a tool to guide you this process if you need the assistance called the Service Configuration Editor. Just right-click on the app.config file. Select Edit WCF Configuration. You'll see the Service Configuration Editor interface, the first step is to click the "Create a New Service". This wizard will lead us through defining the Service Type, Contract, Binding and Address for our service. The first page ask for the service type which in our case is CalcsService.MyCalcs. This can be typed or the browse can be used get the type from the service assembly(..\calcsservice\CalcsImplementation\bin\). Click Next will ask us for a service contract which is filled in for us (CalcsService.IMyCalcs). Clicking Next will take us to the Communication mode Page and here we specify TCP as we are operating within the firewall. Next then takes us to the address Page where we remove the default and type in "Calcs". If we click Save at this point the config file will look like:

```

<configuration>
  <system.serviceModel>
    <services>
      <service name="CalcsService.MyCalcs">
        <endpoint address="Calcs" binding="netTcpBinding"
bindingConfiguration=""
        contract="CalcsService.IMyCalcs" />
      </service>
    </services>
  </system.serviceModel>
</configuration>

```

We have a single relative endpoint , much like Figure 1.3 above and now we want to add the metadata support that we did manually. Recall that this involved :

1. Add a behavior
2. add endpoint
3. Modify the service to use the behavior.

To accomplish (1) right click Service Behaviors under Advanced then click Add and select serviceMetadata from the Available elements list. Change the name to MyBehavior.

The config now contains a Behaviors section and looks like :

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior name="MyBehavior">
        <serviceMetadata />
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <services>
    <service name="CalcsService.MyCalcs">
      <endpoint address="Calcs" binding="netTcpBinding"
bindingConfiguration=""
        contract="CalcsService.IMyCalcs" />
    </service>
  </services>
</system.serviceModel>
```

To accomplish (2) Right click Endpoints and select add Service Endpoint then fill in Address = MEX, binding=mexTcpBinding (from dropdown) and type in IMetadataExchange for the Contract. Give it a name of mex. The following endpoint tag is added to the config:

```
<endpoint address="MEX" binding="mexTcpBinding" bindingConfiguration=""
  name="Mex" contract="IMetadataExchange" />
```

For (3) click on the Service name (CalcsService.MyCalcs) and use the dropdown to select the BehaviorConfiguration which is MyBehavior.

There is one thing still missing and that is the Base address as all these addresses are relative. So click Host and then New and enter net.tcp://localhost:8003/ into the input box. The config file should now look like:

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior name="MyBehavior">
        <serviceMetadata />
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <services>
    <service behaviorConfiguration="MyBehavior"
name="CalcsService.MyCalcs">
      <endpoint address="Calcs" binding="netTcpBinding"
name="basicTCP" contract="CalcsService.IMyCalcs" />
      <endpoint address="MEX" binding="mexTcpBinding" bindingConfiguration=""
```

```

        name="Mex" contract="IMetadataExchange" />
    <host>
        <baseAddresses>
            <add baseAddress="net.tcp://localhost:8003/" />
        </baseAddresses>
    </host>
</service>
</services>
</system.serviceModel>

```

Is this really any easier than manual definition? I guess it is up to the user as the manual method is pretty simple as long as you keep in mind that the config must supply the answers to the ABCs, plus the service type, the metadataExchange endpoint and a host address.

I have shown the basics of developing a simple WCF service that runs on my local machine. If this remained the case then we should use named pipes as the transport but the advantage of TCP is that I can now just change the localhost part of my address to 'machine name' and I can then call it from anywhere in the enterprise. In fact I can start my ConsoleHost on my machine, set a breakpoint in the Add method then call Calcs.Add from another machine and the breakpoint fires.

Windows Services as Host

Most of the Infinite Energy, Inc. services will run inside of the firewall and that is why TCP has been used as the transport and not HTTP as performance is much better. However as has been shown these services must be hosted and running when called. A console application is good for testing but not for production. The best option would be Windows Activation Services but they are not available till IIS 7.0 so that leaves Windows Services as the best option. These do have the advantage of being easily managed by the Control Panel. Administrative Tools. Services. Hopefully the solution will be to put related services into individual container services such as a Utility services might contain PKI encryption, FTP etc. In any event we will now look at how to build the service host. We now add another project to our Solution, this time a Windows service template is chosen. This will generate the code to handle most of the framework for our service. A service is either started automatically when the server boots or manually from the services console. In either case it runs till the machine stops it is manually stopped or it throws. There is a really good article on writing services at [writing services](#). I will cover the highlights, so start by renaming the Service1.cs file to CalcsWinSrvHost. We need the references added for the service Type and the implementation as we did in the Console case, so add the CalcsContract and CalcsImplementation project references and system.servicemodel. Copy the app.config from the Console project., then add the following to lines to the OnStart virtual method:

```

ServiceHost host = new ServiceHost(typeof(MyCalcs));
host.Open();

```

At this point it is recommended to add code like the following so events are recorded in the system event log about the startup:

```

string baseAddresses = "";
foreach (Uri address in m_serviceHost.BaseAddresses)
{

```

```

        baseAddresses += " " + address.AbsoluteUri;
    }
    string s = String.Format("{0} listening at {1}", this.ServiceName,
baseAddresses);
    EventLog.WriteEntry(s, EventLogEntryType.Information);

```

Our service is now ready for action but the Windows service manager must be told that it exists and some startup properties such as account to run under, although these can be modified later from the Services manager console. To do this we add an installer by right clicking on the CalcsWinService.cs file in designer mode and selecting the Add Installer option. A new file is added to the project called ProjectInstaller.cs. The designer for ProjectInstaller.cs can be used to set properties like Name and description of the service. Set the ServiceName (this is name that will appear in list of Windows Services) to MyCalcs and description to "My calculator".

Then the service is installed by running the following from the Visual Studio Command Line:

```
InstallUtil CalcsWinSrvHost.exe
```

Or run it from : C:\WINDOWS\Microsoft.NET\Framework\vXXX

Our service should now appear in the services manager console list where it can be started and stopped. **Debugging a service is not as hard as one might think as just attach the Dev studio debugger to the running service process.**

A really valuable tool in debugging server problems is **NetStat -a** which will display all TCP connections and listening ports. This is especially true when the service starts but the Host is not active since the service appears in the list of running processes. This can be the result of errors like placing the Host in a using that goes out of scope.

Hosting in IIS 6.0

Our project that we have built can be hosted in a console Application or a Windows Service for consumption inside the firewall. Web services will expose the same functionality to the internet and we use IIS 6.0 as our host to accomplish this as it contains the HTTP activation services that are needed to host our application. Start by right clicking on our HelloIndigo solution and selecting new Web site then WCF service. This will add the .svc file that we need and the settings in web.config. It also generates an service.cs file in the App_Code directory that we can delete as we are using an existing contract. We then have to modify the service.svc file to point to our existing contract so it looks like:

```
<% @ServiceHost Language=C# Service="HelloIndigo.HelloIndigoService" %>
```

Now add a project reference for HelloIndigo to the web site so the refernce to HelloIndigo.HelloIndigoService can be resolved.

So now when the service model will activate a new ServiceHost instance associated with the HelloIndigoService type. The serviceModel settings that were place in the web.config file must be modified to also reflect our existing contract and type. We also want to expose service metradata so add the mex endpoint. The services section of web.config should now look like:

```

<services>
  <service name="HelloIndigo.HelloIndigoService"
behaviorConfiguration="returnFaults">
    <endpoint contract="HelloIndigo.IHelloIndigoService"
binding="basicHttpBinding"/>
  </service>
</services>

```

We also need to add a line to the behavior section to expose the service metaData:

```

<serviceMetadata httpGetEnabled="true"/>

```

You can then add a Web Reference to our client application and call the methods on the service:

```

localhost.HelloIndigoService proxy = new
Client.localhost.HelloIndigoService();
string test = proxy.HelloIndigo();

```

Note:

Soap11 is supported by basic basicHttpBinding binding and Soap 1.2 by WSHttBinding
For this to work the .SVC extension must be registered with IIS 6 , to do this see the article [IIS6 svc mapping](#)

Complex Types as Parameters

WCF encourages the use of the new DataContract serializer in lieu of the ASMX xmlSerializer which you can still use. As a result we will modify the existing Contract , something that we normally never do as we are breaking clients modifying the contract. However this is sample code and saves us work by modifying the existing project. We add two new Operation to IMyCalcs contract:

```

[OperationContract]
[Fullname MyOperation1();
[OperationContract]
IList<Fullname> MyOperation2();

```

So we are adding a methods that either return a FullName CLR object to the caller or a List which is of course the interesting one. To use the new DataContract Serializer add the following code to the contract:

```

[DataContract]
public class Fullname
{
    string firstName;
    string lastName;

    [DataMember]
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    [DataMember]
    public string LastName

```

```
{
    get { return lastName; }
    set { lastName = value; }
}
```

We could have just as easily put the [DataMember] attribute around the fields. Now add code to the implementation as follows:

```

private Fullname _name;
//return a Fullname object
public Fullname MyOperation1()
{
    _name = new Fullname();
    _name.FirstName = "john";
    _name.LastName = "doe";
    return _name;
}
//return a list of Fullname objects
public IList<Fullname> MyOperation2()
{
    IList<Fullname> list = new List<Fullname>();
    _name = new Fullname();
    _name.FirstName = "john";
    _name.LastName = "doe";
    list.Add(_name);
    _name = new Fullname();
    _name.FirstName = "bill";
    _name.LastName = "smith";

    list.Add(_name);
    return list;
}

```

Our interface has changed so must regenerate the Service Reference that the client is using but deleting the old one and Adding a new one. Now we can write code in the client that looks like:

```

Supercalc.MyCalcsClient proxy = new Supercalc.MyCalcsClient();
Fullname name = proxy.MyOperation1();
Fullname[] names = proxy.MyOperation2();

```

Note:

The IList has been serialized as array.

WCFTrace

This is a valuable tool that allows us to see all the WCF internal operations and is sometimes valuable in trouble shooting. To enable it add the following to your config file:

```
<system.diagnostics >
  <sharedListeners>
    <add name="sharedListener"
          type="System.Diagnostics.XmlWriterTraceListener"
          initializeData="c:\logs\servertrace.svclog" />
  </sharedListeners>
  <sources>
    <source name="System.ServiceModel" switchValue="Verbose,
ActivityTracing" >
      <listeners>
        <add name="sharedListener" />
      </listeners>
    </source>
    <source name="System.ServiceModel.MessageLogging"
switchValue="Verbose">
      <listeners>
        <add name="sharedListener" />
      </listeners>
    </source>
  </sources>
</system.diagnostics>
```

`initializeData` points to the log location and it can be analyzed with the **Microsoft Service Trace Viewer**

Instancing and Concurrency

When a receiving application receives a message, that message gets dispatched to an object's instance method. WCF is all about sending and receiving messages. There are many settings that have an impact on how a WCF application will scale. The settings for instancing modes and concurrency mode impact the scalability of a WCF receiving application. Instance management refers to a set of techniques used by Windows® Communication Foundation to bind a set of messages to a service instance. **The instancing settings**

Are:

Single: every received message is dispatched to the same object (a singleton)

PerCall (default): every received message is dispatched to a newly created object

PerSession: messages received within a session (usually a single sender) are dispatched to the same object

Concurrency

As WCF is a asynchronous messaging platform. It makes extensive use of asynchronous I/O, and as a result, each received message may be dispatched to a receiving object by different threads. This feature allows WCF to use the CPU efficiently, and as a result, allows WCF applications to scale. `ConcurrencyMode` controls the threading behavior of your service class.

Single (default): only one thread may access the receiver object at a time

Multiple: multiple threads may access the receiver object concurrently

Reentrant: only one thread may access the receiver object at a time, but callbacks may re-enter that object on another thread (Useful when performing asynchronous I/O)

When do we need to worry about synchronization?

Let's first consider the default settings of

`InstanceContextMode=InstanceContextMode.PerCall` and `ConcurrencyMode=Single`.

These settings dictate that a new receiver object will be created for every received message that is properly formatted, and that a single thread will invoke the instance method on the receiver object.:

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerCall,
                 ConcurrencyMode=ConcurrencyMode.Single)]
sealed class Receiver : ISomeContract {
    // member variable for race condition
    Int32 someNumber;

    internal Receiver() {
        Console.WriteLine("a receiver has been created");
    }
}
```

```

}

static void Main(string[] args){
    // create the address and binding
    Uri address = new Uri("net.tcp://localhost:4000/ISomeContract");
    NetTcpBinding binding = new NetTcpBinding(SecurityMode.None, false);

    // create the service host and add an endpoint
    Console.WriteLine("Creating ServiceHost");
    ServiceHost svc = new ServiceHost(typeof(Receiver));
    svc.AddServiceEndpoint(typeof(ISomeContract), binding, address);

    // show the concurrency mode
    ServiceBehaviorAttribute annotation =
svc.Description.Behaviors.Find<ServiceBehaviorAttribute>();
    Console.WriteLine("important behaviors:");
    Console.WriteLine("\tconcurrency mode = {0}", annotation.ConcurrencyMode);
    Console.WriteLine("\tinstance context mode = {0}",
annotation.InstanceContextMode);
    // open the service host (Start listening, among other things)
    Console.WriteLine("opening ServiceHost");
    svc.Open();
    Console.WriteLine("the Receiver is ready\n");

    // wait before exiting process
    Console.ReadLine();
}

// method can only be used with Concurrency Mode = PerCall
public void SomeOperation(Int32 number) {
    // set the field value to the parameter and wait
    someNumber = number;

    Random rand = new Random();
    Thread.Sleep(rand.Next(1000));
    // show the current value of the field and parameter
    // with percall, they will always be the same
    Console.WriteLine("parameter = {0}, somenumber = {1}, thread = {2}\n",
        number.ToString(),
        someNumber.ToString(),
        Thread.CurrentThread.ManagedThreadId);
}
}

```

If a sending application sends 5 messages to this receiving application, the SomeOperation method is invoked 5 times synchronously by the same thread. The following is the console output:

```
C:\WINDOWS\system32\cmd.exe
Creating ServiceHost
important behaviors:
    concurrency mode = Single
    instance context mode = PerCall
opening ServiceHost
the Receiver is ready

a receiver has been created
Main - parameter = 0, somenumber = 0, thread = 4

a receiver has been created
Main - parameter = 1, somenumber = 1, thread = 4

a receiver has been created
Main - parameter = 2, somenumber = 2, thread = 4

a receiver has been created
Main - parameter = 3, somenumber = 3, thread = 4

a receiver has been created
Main - parameter = 4, somenumber = 4, thread = 4

a receiver has been created
Main - parameter = 5, somenumber = 5, thread = 4
```

Since one and only one thread can interact with the receiver object at a time, and each message is dispatched to a new receiver object, there is no need write thread synchronization code in our receiver type definition.

If we change the InstanceContextMode to Single, we see the following output:

```
C:\WINDOWS\system32\cmd.exe
Creating ServiceHost
a receiver has been created
important behaviors:
    concurrency mode = Single
    instance context mode = Single
opening ServiceHost
the Receiver is ready

Main - parameter = 0, somenumber = 0, thread = 4
Main - parameter = 1, somenumber = 1, thread = 4
Main - parameter = 2, somenumber = 2, thread = 4
Main - parameter = 3, somenumber = 3, thread = 4
Main - parameter = 4, somenumber = 4, thread = 4
Main - parameter = 5, somenumber = 5, thread = 4
_
```

Here we have a single instance of the receiver type, but since the `ConcurrencyMode` is still `Single`, there is no need to write thread synchronization code in the `SomeOperation` method.

If, however, we change the `ConcurrencyMode` to `Multiple`, we see the following:

```
C:\WINDOWS\system32\cmd.exe
Creating ServiceHost
a receiver has been created
important behaviors:
    concurrency mode = Multiple
    instance context mode = Single
opening ServiceHost
the Receiver is ready

Main - parameter = 0, somenumber = 0, thread = 4
Main - parameter = 1, somenumber = 1, thread = 4
Main - parameter = 3, somenumber = 3, thread = 5
Main - parameter = 2, somenumber = 5, thread = 4
Main - parameter = 4, somenumber = 5, thread = 5
Main - parameter = 5, somenumber = 5, thread = 3
```

Clearly, we now have a race condition, and as a result, we have to write synchronization code in our `SomeOperation` method. While this can be quite a chore, allows us to service received messages more efficiently than if we allow only one thread at a time to work with the receiver object.

Ideally, your service classes should specify `Multiple` for this option. Note that this will require you to manage synchronization yourself if your class contains instance level variables.

To summarize, here are some basic scenarios for what happens when 100 clients simultaneously hit a service method:

Scenario 1: `InstanceContextMode.Single+ConcurrencyMode.Single`

Result: 100 sequential invocations of the service method on one thread

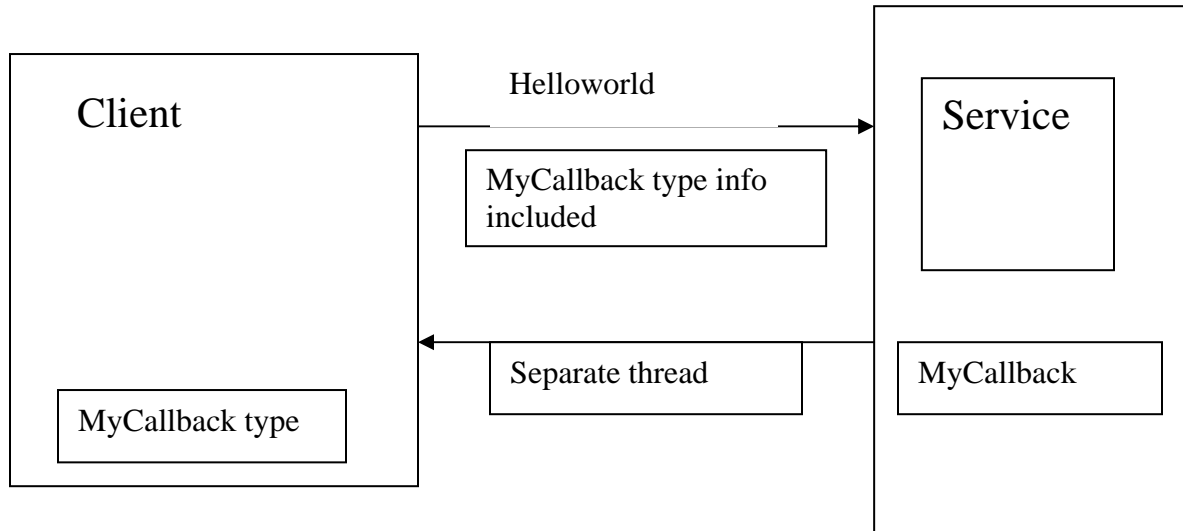
Scenario 2: `InstanceContextMode.Single+ConcurrencyMode.Multiple`

Result: N concurrent invocations of the service method on N threads, where N is determined by the service throttle.

Scenario 3: `InstanceContextMode.PerCall+Any ConcurrencyMode`

Result: N concurrent invocations of the method on N service instances, where N is determined by the service throttle

Callbacks



Sometimes it is necessary for a service to be able to callback into methods on the Client. The first thing we do is create an interface with service operations representing the callback operation by adding to HelloIndigoService.cs.

```
public interface IHelloIndigoServiceCallback
{
    [OperationContract]
    void HelloIndigoCallback(string message);
}
```

Note: Callback contracts do not require a ServiceContractAttribute.

Now associate this callback contract with the service contract so that its metadata will be included in the service description and WSDL document by adding the CallbackContract attribute:

```
[ServiceContract(Name = "HelloIndigoContract", Namespace =
"http://www.InfinteEnergy.com", CallbackContract =
typeof(IHelloIndigoServiceCallback))]
```

The service type must also be marked as reentrant so it will be able to return from the Callback:

```
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant)]
public class HelloIndigoService : IHelloIndigoService
```

When clients create a proxy for a service contract that supports callbacks, they are required to implement the callback contract and supply a channel to receive callbacks. The client proxy will send information about the callback channel with each call to the service. You can access this through the

OperationContractGetCallbackChannel<T>() method. So we add code to HelloIndigo:

```
IHelloIndigoServiceCallback callback =  
  
OperationContext.Current.GetCallbackChannel<IHelloIndigoServiceCallback>();  
callback.HelloIndigoCallback(message);
```

Now make sure the endpoint in app.config is set to support duplex communication as it is with TCP

```
<endpoint address="HelloIndigoService" binding="netTcpBinding"  
contract="HelloIndigo.IHelloIndigoService" />
```

On the client we first add a new class that implements the callback contract:

```
using System;  
using System.Threading;  
class CallbackType : Client.localhost.HelloIndigoContractCallback  
{  
    public void HelloIndigoCallback(string message)  
    {  
        Console.WriteLine("HelloIndigoCallback on thread {0}",  
            Thread.CurrentThread.GetHashCode());  
    }  
}
```

The Client then must provide an implementation for the callback contract by adding to our main and modifying the proxy creation to include the callback instance:

```
CallbackType cb = new CallbackType();  
InstanceContext context = new InstanceContext(cb);  
using (HelloIndigoContractClient proxy = new  
HelloIndigoContractClient(context))
```

Note:

The callback is executed on a separate thread from initial app thread.

MessageContracts and Streaming

Many times a method has no idea how large the return buffer will be as is the case when one is downloading a file from an FTP site. WCF supports two modes of transferring messages. The normal mode is Buffered where it holds the whole message in memory till the operation is complete and read by the client. However with streams the receiver can start processing the message before it is completely delivered, this can really improve the scalability. Thus we do not have to allocate a huge buffer to download the file as the following shows. However where Streams are used they must be the only input or output parameter and this can present a problem if one must pass parameters to the method and that is where MessageContracts come in. When you define a service contract, you traditionally describe a set of operations that may include a parameter list and a return value. Each operation is ultimately linked to a request, and/or a response message. A message contract gives you more granular control over the actual structure of the message. You can use message contracts as the only parameter and as the return type for any operation, in lieu of a parameter lists or return . For my FTP example I want to be able to pass three parms to my FTP download method in my service that contain the FTP site name, the FTP user and password. The return must include not only a stream but the length of the stream. The first thing we do is define messageContract for the input parms:

```
[MessageContract]
public class FTPRequestParms
{
    [MessageBodyMember]
    public string FTPLocation;
    [MessageBodyMember]
    public string FTPUser;
    [MessageBodyMember]
    public string FTPPassword;

    public FTPRequestParms() { }
}
```

These parms will be transmitted as part of the Message Body. Then the return value is represented by the following class:

```
[MessageContract]
public class MyFileInfo
{
    [MessageHeader]
    public string FileName;
    [MessageHeader]
    public long FileSize;
    [MessageBodyMember]
    public Stream Stream;
}
```

The **service contract** is then modified to look like:

```
[OperationContract]
MyFileInfo FTPDownload(FTPRequestParms parms);
```

The **service method** code will now look :

```
//get the response stream from the .Net FTP provider, Read it in "chunks"
//then write each "chunk" to the output FTPFileInfo.Stream
using (WebResponse response = ftpRead.GetResponse())
{
    Stream ftpStream = response.GetResponseStream();
    //read "chunks" of the FTP stream and write to the output
stream
    int bufferSize = 2048;
    byte[] buffer = new byte[bufferSize];
    int readCount = ftpStream.Read(buffer, 0, bufferSize);
    while (readCount > 0)
    {
        FTPFileInfo.Stream.Write(buffer, 0, readCount);
        readCount = ftpStream.Read(buffer, 0, bufferSize);
    }
    ftpStream.Close();
}
```

The **client code** looks like:

```
//get the file from the FTP site using the given credentials, the File is
returned in the FTPFileStream
IEIUUtilities.FTPDownload(CurrentFTPDirectory + FileName,
ConfigurationManager.AppSettings["FTPPassword"],
ConfigurationManager.AppSettings["FTPUserName"], out FTPFileStream);
//stream the data into a local stream
int bufferSize = 2048;
byte[] buffer = new byte[bufferSize];
int readCount = FTPFileStream.Read(buffer, 0, bufferSize);
while (readCount > 0)
{
    FileContentsStream.Write(buffer, 0, readCount);
    readCount = FTPFileStream.Read(buffer, 0, bufferSize);
}
byte[] Received = new byte[FileContentsStream.Length];
FileContentsStream.Seek(0, SeekOrigin.Begin);
//get the buffer out of the stream
FileContentsStream.Read(Received, 0, (int)FileContentsStream.Length);
```

Certainly there is a bit of work involved however this code should handle any size file and not have to worry about buffer sizes.