WCF Routing Service - Part II: Context-based Routing & Protocol Bridging

Samir NIGAM, 7 Jun 2014 CPOL

★★★★★ 4.95 (14 votes)

This article describes context-based routing and protocol bridging using WCF RoutingService.

Download the sample code - 264.9 KB

Table of Contents

All Posts

- WCF Routing Service - Part IV: Service Versioning & Multicasting
- WCF Routing Service - Part III: Failover & Load Balancing
- WCF Routing Service - Part II: Context-based Routing & Protocol Bridging
- WCF Routing Service - Part I: Basic Concept, Simple Routing Service & Content-based Routing

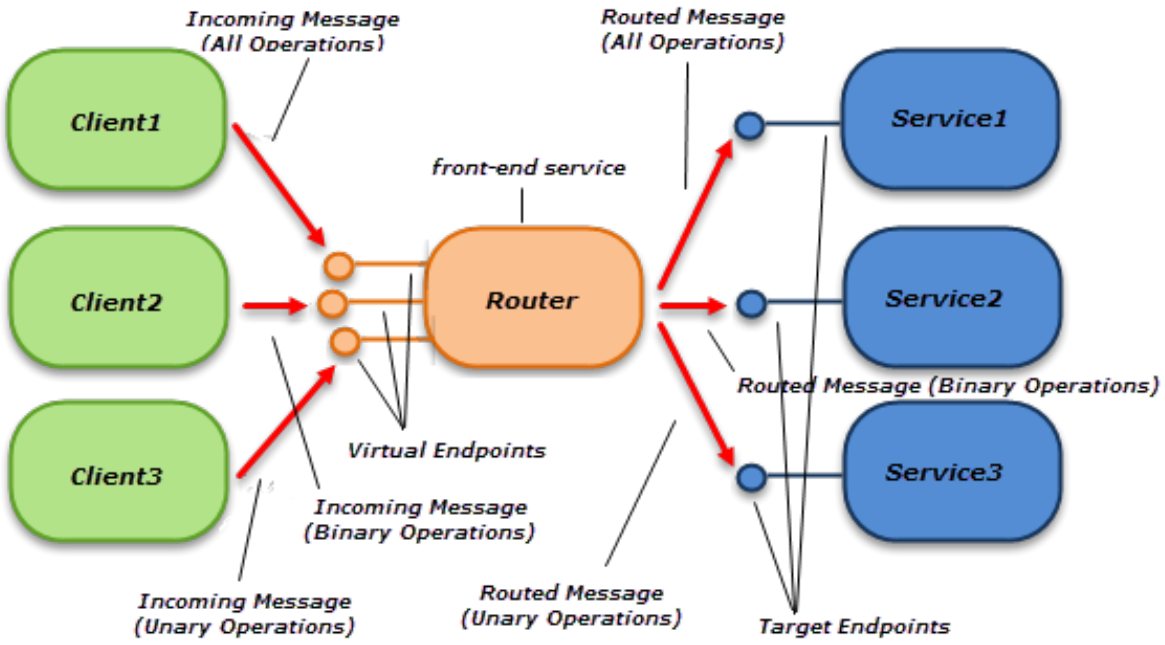
Introduction

In the first part of the series, I've introduced **WCF Routing Service** concept, then described its virtual endpoint(s), target service(s), message filter(s) and filter table configurations respectively. Then I've demonstrated a simple **Routing Service**example and finally explained content-based routing using **WCF** `RoutingService`. In this part of the series, I'm going to demonstrate context-based routing and protocol bridging using **WCF** `RoutingService`.

Context Based Routing

In Context-based routing, the incoming messages are routed through an intermediary based on the channel on which messages arrive not by inspecting the content of the incoming messages.

Let's consider an example to understand Context-based routing. Suppose that there are three types of clients of our `ComplexNumberCalculator` service (say **Client1**,**Client2** & **Client3**). **Client1** can perform all complex number operations while**Client2** & **Client3** have only access of **Binary** and **Unary** operations of complex number respectively. So there would be three endpoints in the intermediary for '**all**',' **binary**' and '**unary**' complex number operations respectively. **Client1** will use one endpoint (say '**all**'), **Client2** will use another endpoint (say '**binary**') and **Client3**will use the last endpoint (say '**unary**'). **Client1** will send the messages to the endpoint named '**all**' that must be routed by the intermediary to a service handling all operations of complex number, **Client2** will send the messages to the endpoint named '**binary**' that must be routed by the intermediary to a service handling binary operations of complex number and **Client3** will send the messages to the endpoint named '**unary**' that must be routed by the intermediary to a service handling unary operations of complex number.



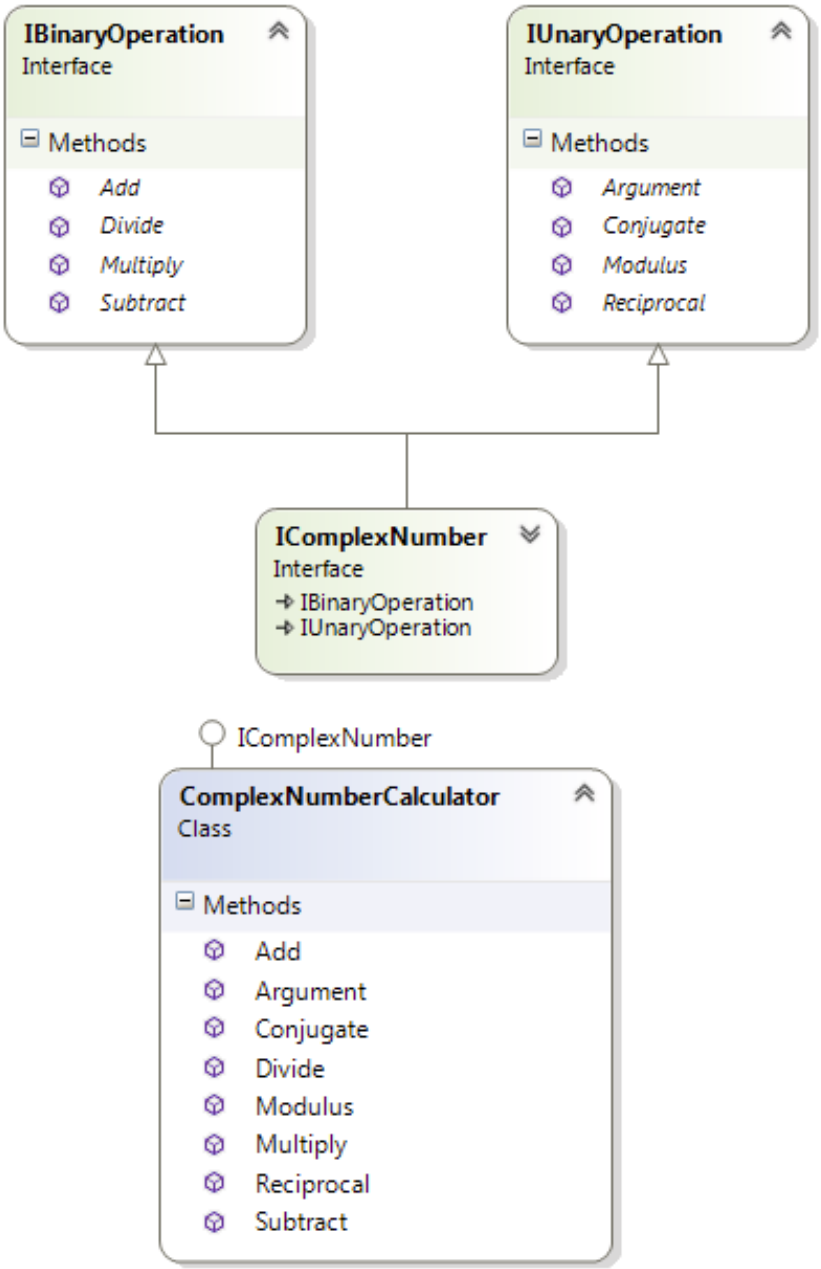
Our Demo Service

In order to implement the above scenario using context-based routing, I've made some changes in our `ComplexNumberCalculator` service. I've defined two new service contracts `IBinaryOperation` & `IUnaryOperation` for binary and unary operations of complex number respectively. Then I've defined the empty`IComplexNumber` service contract by implementing these two service contracts. Actually I've moved all complex number operations by grouping into binary and unary operations from `IComplexNumber` service contract to `IBinaryOperation`& `IUnaryOperation` service contracts respectively. Please see the service contracts definitions below-

```
[ServiceContract]
public interface IBinaryOperation
{
    [OperationContract]
    Complex Add(Complex x, Complex y);

    [OperationContract]
```

```
    Complex Subtract(Complex x, Complex y);

    [OperationContract]
    Complex Multiply(Complex x, Complex y);

    [OperationContract]
    Complex Divide(Complex x, Complex y);
}

[ServiceContract]
public interface IUnaryOperation
{
    [OperationContract]
    double Modulus(Complex x);

    [OperationContract]
    double Argument(Complex x);

    [OperationContract]
    Complex Conjugate(Complex x);

    [OperationContract]
    Complex Reciprocal(Complex x);
}

[ServiceContract]
public interface IComplexNumber : IBinaryOperation, IUnaryOperation
{
}
```

Below is the class diagram of the above design-



Next I've configured `ComplexNumberCalculator` service with three endpoints in our console host application (one for each client type) along with a standard **mex**endpoint-

```
<services>
    <service name="CalculatorService.ComplexNumberCalculator">
       <endpoint address="" binding="basicHttpBinding" contract="CalculatorService.IComplexNumber" />
       <endpoint address="binary" binding="basicHttpBinding"
contract="CalculatorService.IBinaryOperation" />
       <endpoint address="unary" binding="basicHttpBinding"
contract="CalculatorService.IUnaryOperation" />
       <endpoint address="mex" kind="mexEndpoint" />
       <host>
          <baseAddresses>
             <add baseAddress="http://localhost:8081/ComplexNumberService" />
          </baseAddresses>
       </host>
</service>
```

Instead of creating the three client applications one for each client type, I've simulated the same in a single

client application by configuring following three client side endpoints-

```
<client>
        <endpoint address="http://localhost:8080/RoutingService/Router" binding="basicHttpBinding"
                  contract="IComplexNumber" name="BasicHttpBinding_IComplexNumber" />
        <endpoint address="http://localhost:8080/RoutingService/Router/binary"
binding="basicHttpBinding"
                  contract="IBinaryOperation" name="BasicHttpBinding_IBinaryOperationr" />
        <endpoint address="http://localhost:8080/RoutingService/Router/unary"
binding="basicHttpBinding"
                  contract="IUnaryOperation" name="BasicHttpBinding_IUnaryOperation" />
</client>
```

Below is the client application code-

```
var cfAll = new ChannelFactory<IComplexNumber>("BasicHttpBinding_IComplexNumber");
var cfBinary = new ChannelFactory<IBinaryOperation>("BasicHttpBinding_IBinaryOperationr");
var cfUnary = new ChannelFactory<IUnaryOperation>("BasicHttpBinding_IUnaryOperation");

var channelAll = cfAll.CreateChannel();
var channelBinary = cfBinary.CreateChannel();
var channelUnary = cfUnary.CreateChannel();

var z1 = new Complex();
var z2 = new Complex();

z1.Real = 3D;
z1.Imaginary = 4D;

z2.Real = 1D;
z2.Imaginary = -2D;

Console.WriteLine("*** Context Based Routing ***\n");
Console.WriteLine("\nPlease hit any key to run OR enter 'exit' to exit.");
string command = Console.ReadLine();

   while (command != "exit")
   {
      Console.WriteLine("Please hit any key to start Client1: ");
      Console.ReadLine();
      ComplexNumberArithmetics(channelAll, z1, z2);

      Console.WriteLine("Please hit any key to start Client2: ");
      Console.ReadLine();
      ComplexNumberBinaryArithmetics(channelBinary, z1, z2);

      Console.WriteLine("Please hit any key to start Client3: ");
      Console.ReadLine();
      ComplexNumberUnaryArithmetics(channelUnary, z1);

      Console.WriteLine("\nPlease hit any key to re-run OR enter 'exit' to exit.");
      command = Console.ReadLine();
   }

((IClientChannel)channelAll).Close();
((IClientChannel)channelBinary).Close();
((IClientChannel)channelUnary).Close();
```

In the above, the method `ComplexNumberArithmetics` performs all operations of complex number, the method `ComplexNumberBinaryArithmeticss` performs only binary operations of complex number and the method`ComplexNumberUnaryArithmetics` performs only unary operations of complex number (see the code of these methods in the sample).

That's it.

Context Based Routing using EndpointName filterType

In this section I'll configure our `RoutingService` for the context-based routing using the `EndpointName filterType`. First I've configured `RoutingService`with following three virtual endpoints one for each client type-

```
<services>
   <service name="System.ServiceModel.Routing.RoutingService">
      <endpoint address="" binding="basicHttpBinding"
                       contract="System.ServiceModel.Routing.IRequestReplyRouter" name="All" />
      <endpoint address="binary" binding="basicHttpBinding"
                      contract="System.ServiceModel.Routing.IRequestReplyRouter" name="Binary" />
      <endpoint address="unary" binding="basicHttpBinding"
                      contract="System.ServiceModel.Routing.IRequestReplyRouter" name="Unary" />
      <host>
         <baseAddresses>
            <add baseAddress="http://localhost:8080/RoutingService/Router" />
         </baseAddresses>
      </host>
   </service>
</services>
```

Next I've defined three endpoints for each target service-

```
<client>
   <endpoint address="http://localhost:8081/ComplexNumberService1" binding="basicHttpBinding"
                   contract="*" name="AllOperation" />
   <endpoint address="http://localhost:8081/ComplexNumberService2" binding="basicHttpBinding"
                   contract="*" name="BinaryOperation" />
   <endpoint address="http://localhost:8081/ComplexNumberService3" binding="basicHttpBinding"
                   contract="*" name="UnaryOperation" />
</client>
```

Next I've enabled the `RoutingBehavior` followed by specifying the name of the filter table. I've done this by defining the **default behavior**-

```
<behaviors>
   <serviceBehaviors>
      <behavior name="">
         <routing filterTableName="RoutingTable" />
      </behavior>
   </serviceBehaviors>
</behaviors>
```

Then I've defined following filters for each virtual endpoint of the `RoutingService`by setting `filterData` attribute to the name of the virtual endpoint using the`filterType`: '`EndpointName`'-

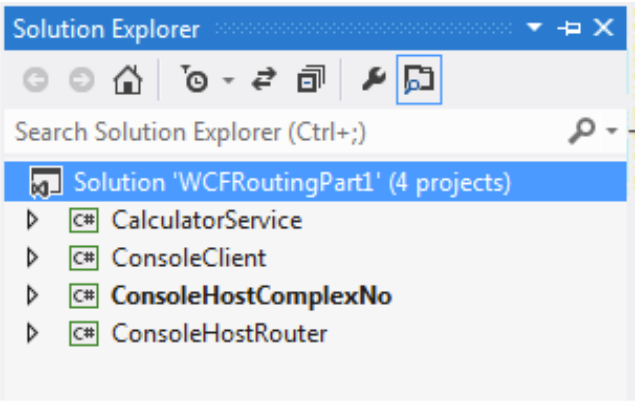
```
<filters>
          <filter name="AllOperationFilter" filterType="EndpointName" filterData="All" />
          <filter name="BinaryOperationFilter" filterType="EndpointName" filterData="Binary" />
          <filter name="UnaryOperationFilter" filterType="EndpointName" filterData="Unary" />
</filters>
```

Finally I've mapped each filter to the respective target service endpoint in the filter table: `RoutigTable` as

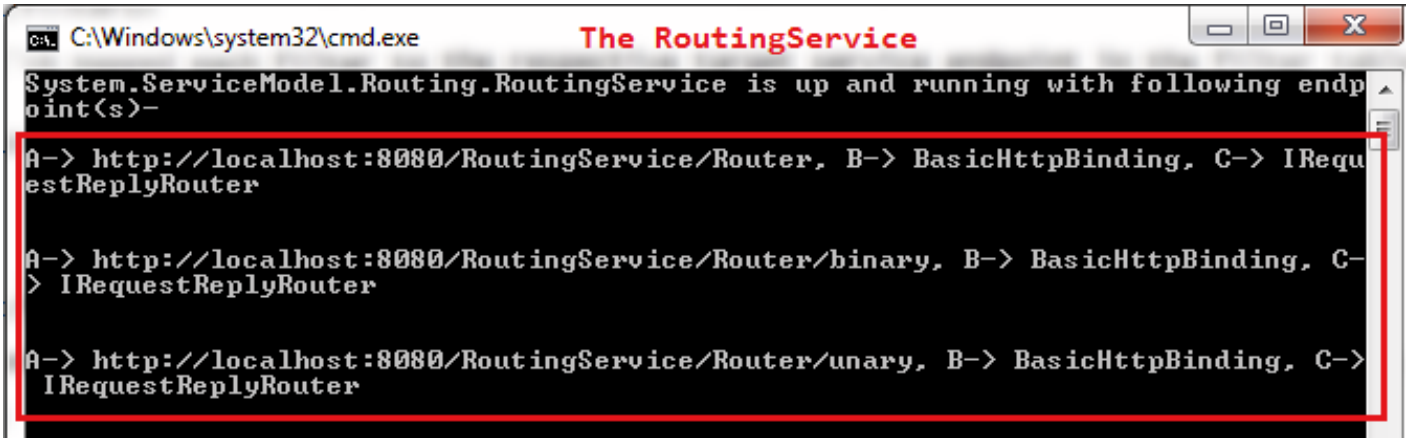
below-

```
<filterTables>
          <filterTable name="RoutingTable">
            <add filterName="AllOperationFilter" endpointName="AllOperation" />
            <add filterName="BinaryOperationFilter" endpointName="BinaryOperation" />
            <add filterName="UnaryOperationFilter" endpointName="UnaryOperation" />
          </filterTable>
</filterTables>
```

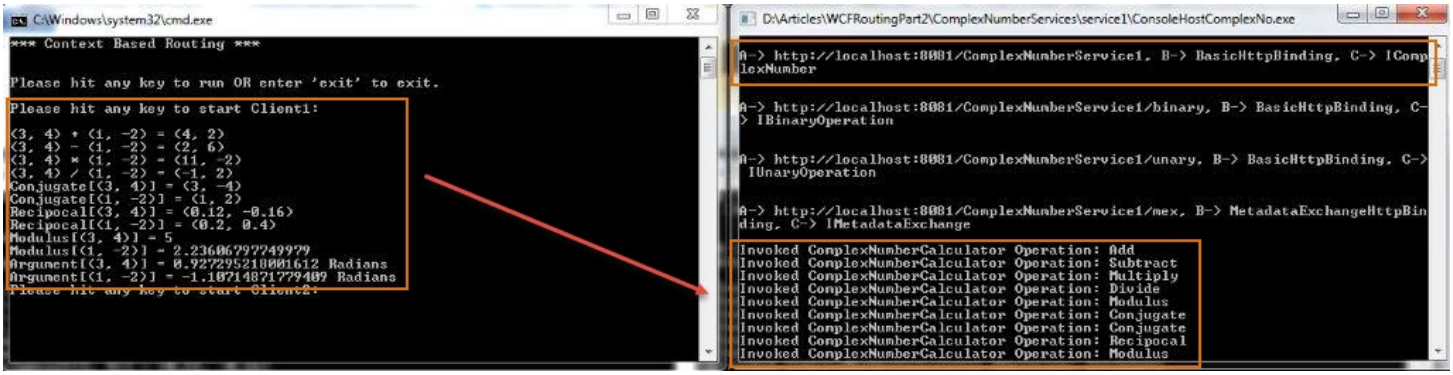
Before running our demo project, just have a quick look of sample project provided with this post-



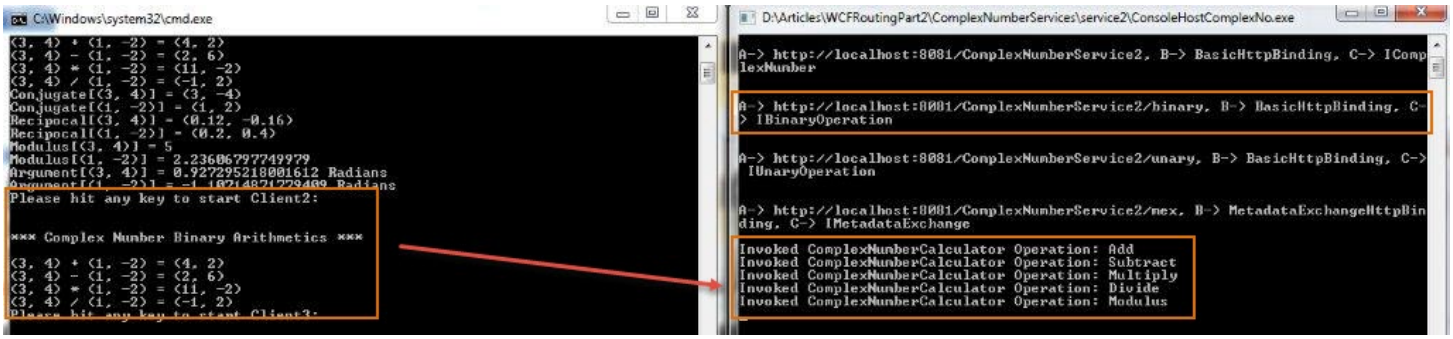
Now set **ConsoleClient** & **ConsoleHostRouter** projects as **Start Up projects** and hit **Ctrl+F5 keys** in order to run the projects. Next run the**WCFRoutingPart2\ComplexNumberServices\StartAllServices.cmd** file (see the sample code) from the **Visual Studio Developer Command Prompt** (in**Administrator** mode) in order to start `ComplexNumberService1`,`ComplexNumberService2` and `ComplexNumberService3` services. Now see the`RoutingService` with three virtual endpoints-



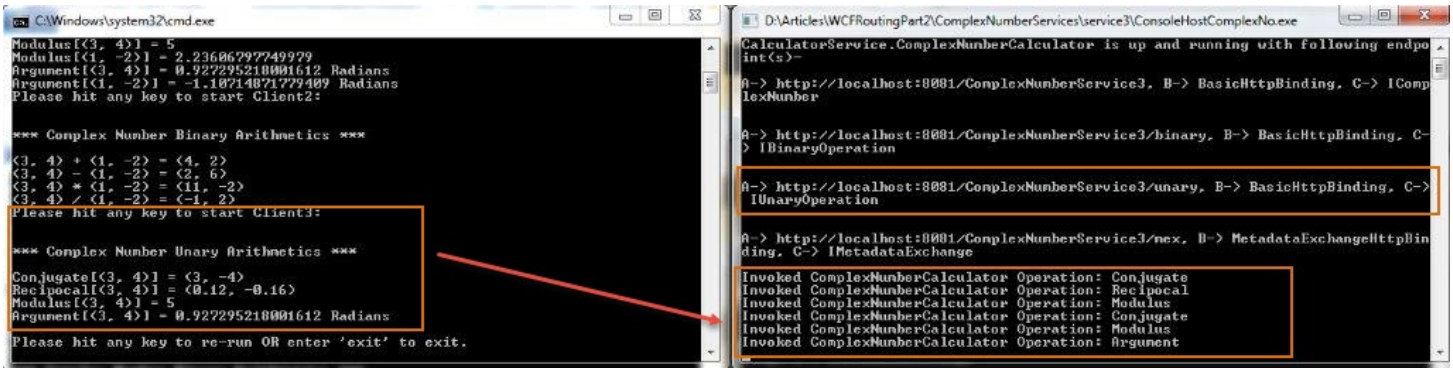
Next press any key on the console client window in order to simulate our **Client1**, you can verify that all complex number operations are routed to the`ComplexNumberService1` service by the intermediary `RoutingService`.



Next press any key on the console client window in order to simulate our **Client2**, you can verify that complex number's **Binary Operations** are routed to the`ComplexNumberService2` service by the intermediary `RoutingService`.



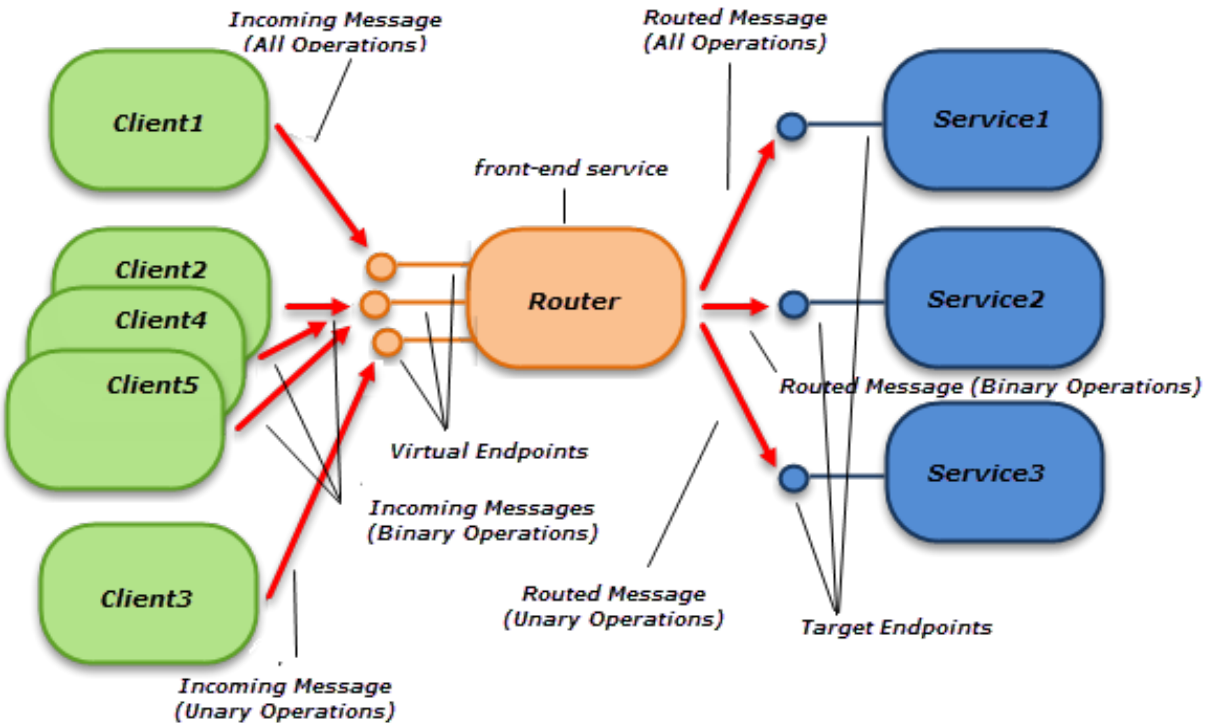
Finally press any key on the console client window in order to simulate our **Client3**, you can verify that complex number's **Unary Operations** are routed to the`ComplexNumberService3` service by the intermediary `RoutingService`.



So from this example you have seen that the routing is made based on the channel on which messages arrived. You can also implement context-based routing by inspecting the addresses of the incoming messages using `EndpointAddress` or`EndpointAddressPrefix filterTypes`.

Context Based Routing using EndpointAddressPrefix filterType

In this section I'll configure our `RoutingService` for the context-based routing using the `EndpointAddressPrefix` filterType. Let's consider two more client types say **Client4** and **Client5** to perform complex number's **Binary Operations**.



So I've configured `RoutingService` with two more virtual endpoints, one for**Client4** and other for **Client5**, like below-

```
<services>
   <service name="System.ServiceModel.Routing.RoutingService">
      <endpoint address="all" binding="basicHttpBinding"
                        contract="System.ServiceModel.Routing.IRequestReplyRouter" name="All" />
      <endpoint address="binary" binding="basicHttpBinding"
                       contract="System.ServiceModel.Routing.IRequestReplyRouter" name="Binary" />
      <endpoint address="unary" binding="basicHttpBinding"
                       contract="System.ServiceModel.Routing.IRequestReplyRouter" name="Unary" />

      <endpoint address="binary/another" binding="basicHttpBinding"
                      contract="System.ServiceModel.Routing.IRequestReplyRouter" name="BinaryAnother"
/>
      <endpoint address="binary/another/extra" binding="basicHttpBinding"
                      contract="System.ServiceModel.Routing.IRequestReplyRouter"
name="BinaryAnotherExtra" />
      <host>
         <baseAddresses>
            <add baseAddress="http://localhost:8080/RoutingService/Router" />
         </baseAddresses>
      </host>
   </service>
</services>
```

Then I've updated previously defined filters using the `PrefixEndpointAddressfilterType` by setting `filterData` attribute to the address prefixes for each complex number operations like below-

```
<filters>
          <filter name="AllOperationFilter" filterType="PrefixEndpointAddress"
filterData="http://localhost:8080/RoutingService/Router/all" />
          <filter name="BinaryOperationFilter" filterType="PrefixEndpointAddress"
filterData="http://localhost:8080/RoutingService/Router/binary" />
          <filter name="UnaryOperationFilter" filterType="PrefixEndpointAddress"
filterData="http://localhost:8080/RoutingService/Router/unary" />
</filters>
```

Please note that I've defined only one filter for the complex number **Binary Operations** as we're using `PrefixEndpointAddress filterType`. So all incoming messages on the virtual endpoints- **Binary**, **BinaryAnother** and**BinaryAnotherExtra** would be handled by this filter and messages would be routed to service `ComplexNumberService2` with endpoint name **BinaryOperation**.

Next I've configured client application with two more endpoints for the`RoutingService` virtual endpoints as described above like below-

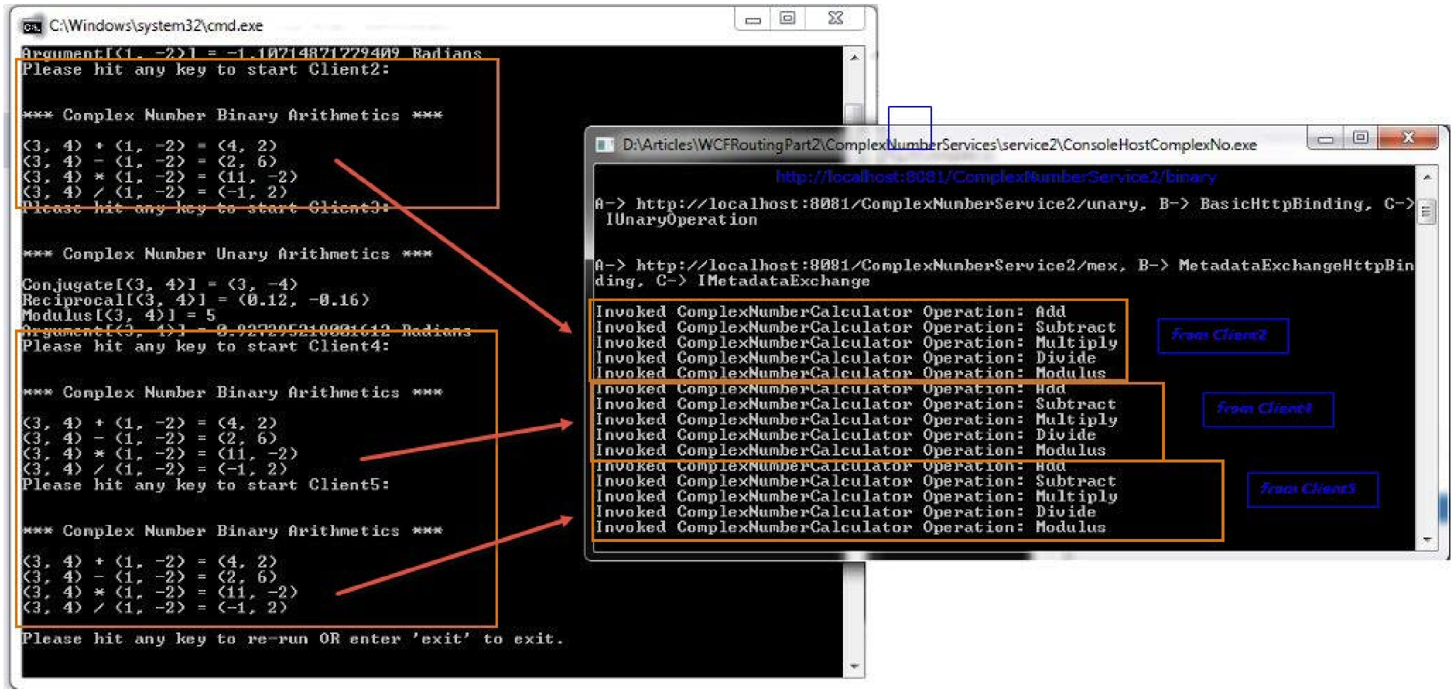
```
<client>
        <endpoint address="http://localhost:8080/RoutingService/Router/all" binding="basicHttpBinding"
                  contract="IComplexNumber" name="BasicHttpBinding_IComplexNumber" />
        <endpoint address="http://localhost:8080/RoutingService/Router/binary"
binding="basicHttpBinding"
                  contract="IBinaryOperation" name="BasicHttpBinding_IBinaryOperationr" />
        <endpoint address="http://localhost:8080/RoutingService/Router/unary"
binding="basicHttpBinding"
                  contract="IUnaryOperation" name="BasicHttpBinding_IUnaryOperation" />

        <endpoint address="http://localhost:8080/RoutingService/Router/binary/another"
binding="basicHttpBinding"
                  contract="IBinaryOperation" name="BasicHttpBinding_IBinaryOperationr_Another" />
        <endpoint address="http://localhost:8080/RoutingService/Router/binary/another/extra"
binding="basicHttpBinding"
                  contract="IBinaryOperation" name="BasicHttpBinding_IBinaryOperationr_Another_Extra"
/>
</client>
```

Finally I've made some changes in the client side application to incorporat **Client4**and **Client5**-

```
...

var cfBinaryAnother = new ChannelFactory<IBinaryOperation>
("BasicHttpBinding_IBinaryOperationr_Another");
var cfBinaryAnotherExtra = new ChannelFactory<IBinaryOperation>
("BasicHttpBinding_IBinaryOperationr_Another_Extra");

...

var channelBinaryAnother = cfBinaryAnother.CreateChannel();
var channelBinaryAnotherExtra = cfBinaryAnotherExtra.CreateChannel();

...

Console.WriteLine("Please hit any key to start Client4: ");
Console.ReadLine();
ComplexNumberBinaryArithmetics(channelBinaryAnother, z1, z2);

Console.WriteLine("Please hit any key to start Client5: ");
Console.ReadLine();
ComplexNumberBinaryArithmetics(channelBinaryAnotherExtra, z1, z2);

...

((IClientChannel)channelBinaryAnother).Close();
((IClientChannel)channelBinaryAnotherExtra).Close();

...
```

Now just follow the instructions of the previous section to run the demo, you'll notice that all incoming messages on virtual endpoints defined for the complex number's **Binary Operations** are routed to

the ComplexNumberService2 service.



Protocol Bridging

Generally two services communicate with each other if they are configured with the same binding. But what if both services are configured using different bindings? Would they be able to communicate with each other in any ways or not? There would be two possible ways to communicate theoretically. The first option would be to update the client application with the binding of the back-end service. But it may not be always possible e.g. in case of interoperability. The second option would be to re-configure the back-end service with the binding the client application wants to communicate with. But again this also may not be always possible e.g. if you don't own the service, and hence causes a maintainability issue where you will need to expose more endpoints.

Protocol bridging eliminates such conditions and service always communicates with the client applications in the same way as previously. This kind of decoupling is very helpful because it ensures that the additional communication requirements don’t impact the back-end service.

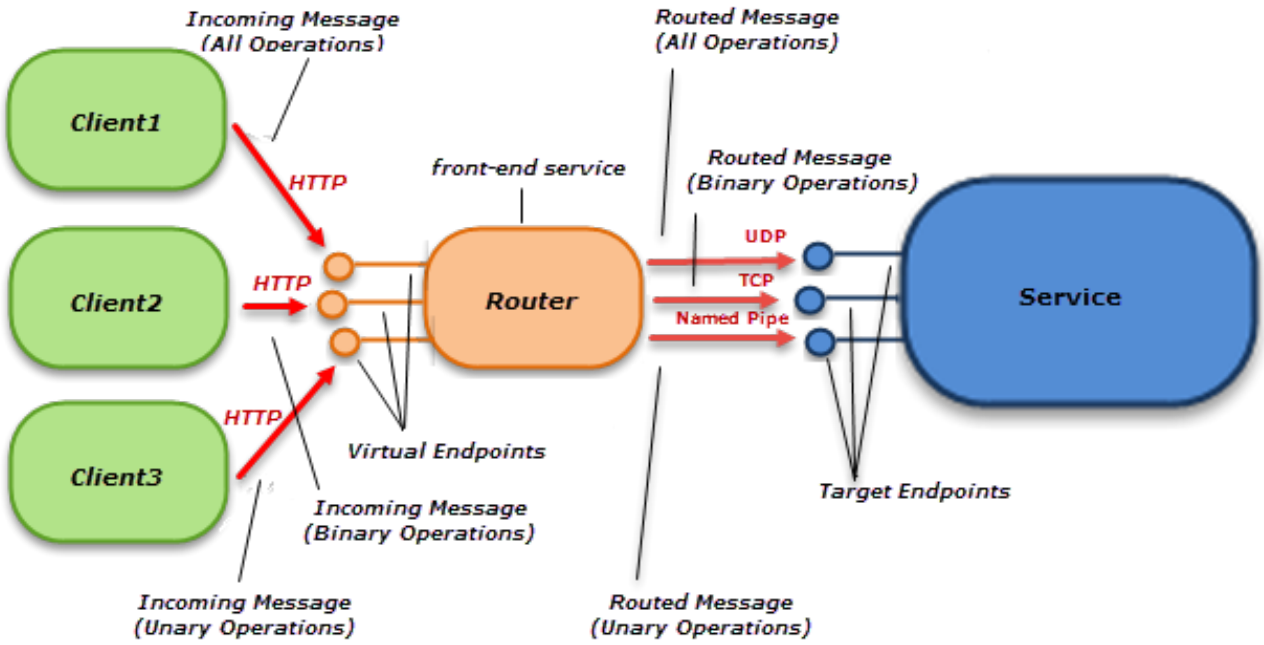
But please note that the communication shape at the routing service must always be symmetrical i.e. if the communication is request-response type between the client and the routing service, then it must also be request-response type between the routing service and the back-end service. If the communication is request-response type between the client application and the routing service and it is one-way type between the routing service and the back-end service OR vice versa, then it don't make any sense.

In all previous examples, I've used the same `basicHttpBinding` to communicate between the client and the router, and between the router and the target service(s). The **WCF** `RoutingService` also provides strong support of bridging the communication across most of the **WCF** bindings. e.g. you may configure the router with `basicHttpBinding` so that client applications can communicate with it over the `basicHttpBinding`, but the router communicates with the target services using the `NetTcpBinding`, `NetNamedPipeBinding` or `udpBinding`.

Let's re-consider our previous example of the "**Context Based Routing**" section and re-configure the same for the protocol bridging demo. Suppose that`ComplexNumberCalculator` service exposes three endpoints

using `udpBinding`,`NetTcpBinding` and `NetNamedPipeBinding` bindings with `IComplexNumber`,`IBinaryOperation` & `IUnaryOperation` service contracts respectively; and router is configured with virtual endpoints using `basicHttpBinding` to talk with client applications.

So **Client1**, **Client2** and **Client3** will communicate with the router using`basicHttpBinding` while the router will communicate with the target service using `udpBinding`, `NetTcpBinding` and `NetNamedPipeBinding` for **Client1**,**Client2** and **Client3** respectively.



So as per the example stated above, first I've configured`ComplexNumberCalculator` service with following three endpoints using`udpBinding`, `NetTcpBinding` and `NetNamedPipeBinding` bindings along with a standard **mex** endpoint as below-

```
<services>
      <service name="CalculatorService.ComplexNumberCalculator">
        <endpoint address="udp" binding="udpBinding" contract="CalculatorService.IComplexNumber" />
        <endpoint address="tcp" binding="netTcpBinding" contract="CalculatorService.IBinaryOperation"
/>
        <endpoint address="pipe" binding="netNamedPipeBinding"
contract="CalculatorService.IUnaryOperation" />
        <endpoint address="mex" kind="mexEndpoint" />
        <host>
          <baseAddresses>
            <add baseAddress="soap.udp://224.1.1.1:40000/ComplexNumberService" />
            <add baseAddress="net.tcp://localhost:8081/ComplexNumberService" />
            <add baseAddress="net.pipe://localhost/ComplexNumberService" />
            <add baseAddress="http://localhost:8082/ComplexNumberService" />
          </baseAddresses>
        </host>
      </service>
 </services>
```

Next I've re-configured `RoutingService` with following three virtual endpoints one for each client type-

```
<services>
           <service name="System.ServiceModel.Routing.RoutingService">
             <endpoint address="all" binding="basicHttpBinding"
                       contract="System.ServiceModel.Routing.IRequestReplyRouter" name="All" />
```

```
            <endpoint address="binary" binding="basicHttpBinding"
                      contract="System.ServiceModel.Routing.IRequestReplyRouter" name="Binary" />
            <endpoint address="unary" binding="basicHttpBinding"
                      contract="System.ServiceModel.Routing.IRequestReplyRouter" name="Unary" />
            <host>
                  <baseAddresses>
                      <add baseAddress="http://localhost:8080/RoutingService/Router" />
                  </baseAddresses>
              </host>
          </service>
</services>
```

Next I've re-defined three endpoints for each target service-

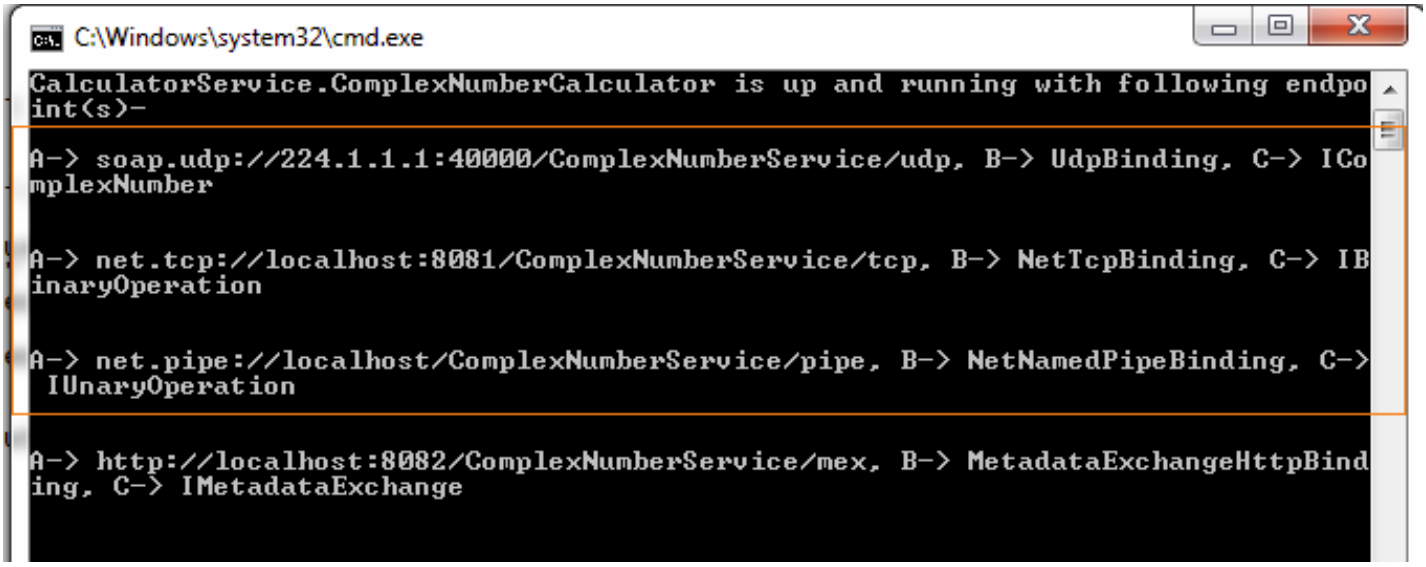
```
<client>
       <endpoint address="soap.udp://224.1.1.1:40000/ComplexNumberService/udp" binding="udpBinding"
                  contract="*" name="AllOperation" />
        <endpoint address="net.tcp://localhost:8081/ComplexNumberService/tcp" binding="netTcpBinding"
                  contract="*" name="BinaryOperation" />
         <endpoint address="net.pipe://localhost/ComplexNumberService/pipe"
binding="netNamedPipeBinding"
                  contract="*" name="UnaryOperation" />
</client>
```

Then I've re-defined following filters for each virtual endpoint of the`RoutingService` by setting `filterData` attribute to the address of the virtual endpoint using the `filterType`: '`EndpointAddress`'-

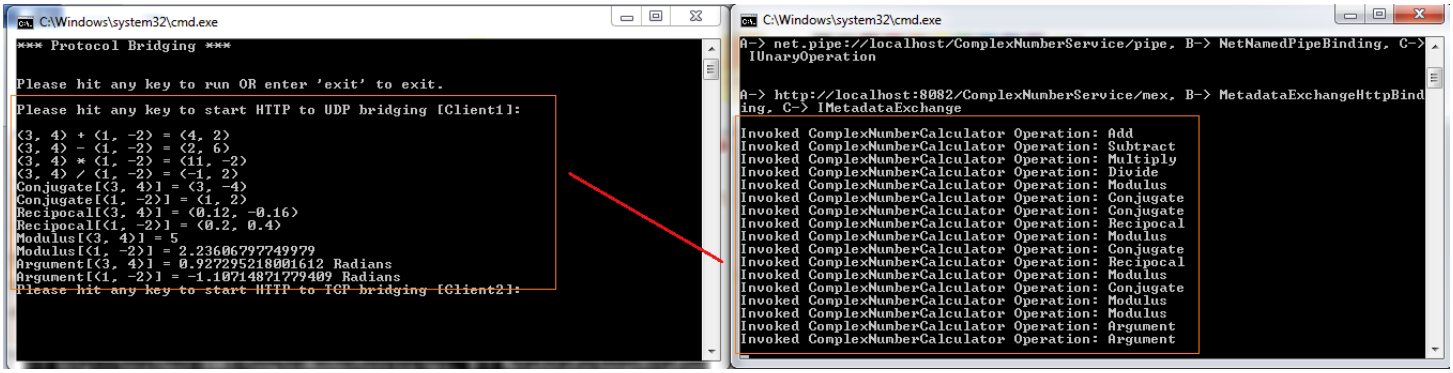
```
<filters>
          <filter name="AllOperationFilter" filterType="EndpointAddress"
filterData="http://localhost:8080/RoutingService/Router/all" />
          <filter name="BinaryOperationFilter" filterType="EndpointAddress"
filterData="http://localhost:8080/RoutingService/Router/binary" />
          <filter name="UnaryOperationFilter" filterType="EndpointAddress"
filterData="http://localhost:8080/RoutingService/Router/unary" />
</filters>
```

Finally I've also made some changes in the client application code. You can find the same in the sample application attached with this post.

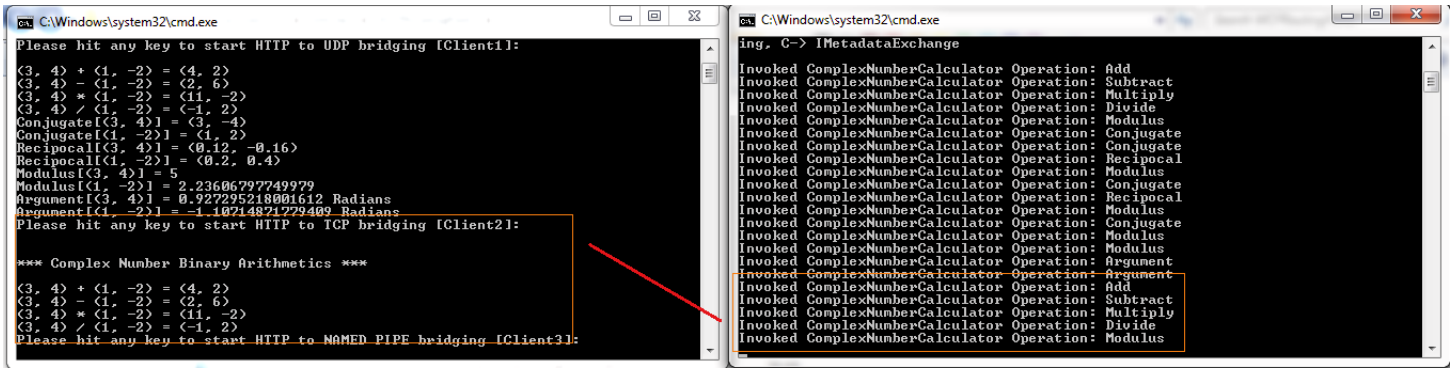
All done. Now set **ConsoleClient**, **ConsoleHostComplexNo** & **ConsoleHostRouter**projects as **Start Up projects** and hit **Ctrl+F5 keys** in order to run the projects. Now just see the target service `ComplexNumberCalculator` with three endpoints along with standard **mex** endpoint-



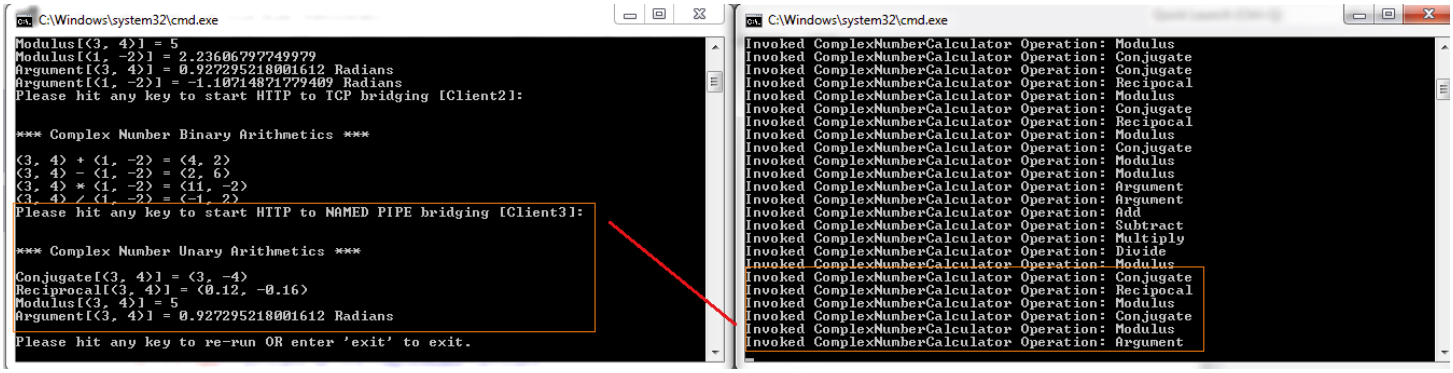
Next press any key on the console client window in order to simulate the **HTTP** to**UDP** protocol bridging, you can verify that the **Client1** communicates with the router using the `basicHttpBinding` and then router communicates with the '**udp**' endpoint of the `complexNumberCalculator` service using the`udpBinding`.



Next again press any key on the console client window in order to simulate the**HTTP** to **TCP** protocol bridging, you can verify that the **Client2** communicates with the router using the `basicHttpBinding` and then router communicates with the '**tcp**' endpoint of the `complexNumberCalculator` service using the`netTcpBinding`.



Next again press any key on the console client window in order to simulate the**HTTP** to **NamedPipe** protocol bridging, you can verify that the **Client3**communicates with the router using the `basicHttpBinding` and then router communicates with the '**pipe**' endpoint of the `complexNumberCalculator`service using the `netNamedPipeBinding`.



Note that There are two key aspect of protocol bridging in the context of`RoutingService`- **Transport Mediation** (means the client and the service don't necessarily need the same transport to communicate on the wire.) & **Protocol Conversion** (means implicitly handling of any conversion of the communication protocol if neede e.g. from **SOAP 1.1** [`basicHttpBinding`] to **SOAP 1.2**[`NetTcpBinding`]).

Conclusion

In this post, I've explored **context-based** routing using `EndpointName` &`PrefixEndpointAddress` filter types. Then I've explained **protocol bridging**concept. I've tried my level best to demonstrate these concepts using demo applications. In the next post I'll explore some more concepts like **multicasting**,**load balancing** etc. Till then, happy coding.

History

- 7th Jun, 2014 -- Article updated (Added a new entry for the fourth part of the series in 'All Posts' section)
- 28th May, 2014 -- Article updated (Added a new entry for the third part of the series in 'All Posts' section)
- 27th May, 2014 -- Article updated (updated the URL of the first part of the series in 'All Posts' section)
- 24th May, 2014 -- Article updated
    - (updated the article's title)
    - (updated the entries of the 'All Posts' section)
- 23rd May, 2014 -- Article updated (uploaded the updated sample code)
- 22nd May, 2014 -- Article updated
    - (updated the 'Context Based Routing using EndpointName filterType' section by adding the content for enabling the RoutingBehavior)
    - (corrected typo mistakes)
- 21th May, 2014 -- Article updated (updated the entries of the 'All Posts' section)
- 20th May, 2014 -- Article updated
    - (Re-arranged entries of the 'All Posts' section)
    - (Added the table of contents section)
- 19th May, 2014 -- Original version posted

License

Share

About the Author

Samir NIGAM

Technical Lead Infogain India Pvt Ltd

India

Samir NIGAM is a **Microsoft Certified Professional**. He is an insightful IT professional with results-driven comprehensive technical skill having rich, hands-on work experience n web-based applications using **ASP.NET**, **C#**, **AJAX**, **Web Service**, **WCF**, **jQuery**, **Microsoft Enterprise Library** , **LINQ**, **MS Entity Framework**, **nHibernate**, **MS SQL Server** & **SSRS**.

He has earned his master degree (**MCA**) from U.P. Technical University, Lucknow, INDIA, his post graduate dipoma (**PGDCA**) from Institute of Engineering and Rural Technology, Allahabad, INDIA and his bachelor degree (**BSc - Mathematics**) from University of Allahabad, Allahabad, INDIA.

He has good knowledge of **Object Oriented Programming**, **n-Tier Architecture**, **SOLID Principle**, and **Algorithm Analysis & Design** as well as good command over cross-browser client side programming using **JavaScript** &**jQuery**,.

Awards:

- **Code Project MVP 2009.**
- **Best ASP.NET article of December 2008.**
- **Best ASP.NET article of June 2008.**