WCF Routing Service - Part III: Failover & Load Balancing

Samir NIGAM, 7 Jun 2014 CPOL

4.90 (15 votes)

This article describes Failover & Load Balancing using WCF RoutingService.

Download the sample code - 401 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

This is the third part of the **WCF Routing** series. In this post, I'll explore Failover and**Load Balancing** features related to **WCF** `RoutingService`. **Failover** or **High- Availability** is basically used to provide redundancy with minimum down-time in case of application failure or crash. **Load Balancing** is related to provide the high performance requests processing in peak loads. Let's start to explore these features one by one in coming sections.

Failover

It is very important for a critical service to be both reliable and highly available. It should be always available for its end-user(s) in case of errors due to the single server failure or the hosting applications failure. That is a client application should never be interrupted in any case of failure.

So a highly available application infrastructure tier should be designed to protect against loss of service due to any kind of failure. The service must be hosted on multiple servers to provide the redundancy with minimum down time. If one of the servers becomes unavailable, another server takes over the charge and continues to provide the service to the end-user(s). This phenomenon is known as failover. When failover occurs, users continue to use the service and are unaware of service providing source (server).

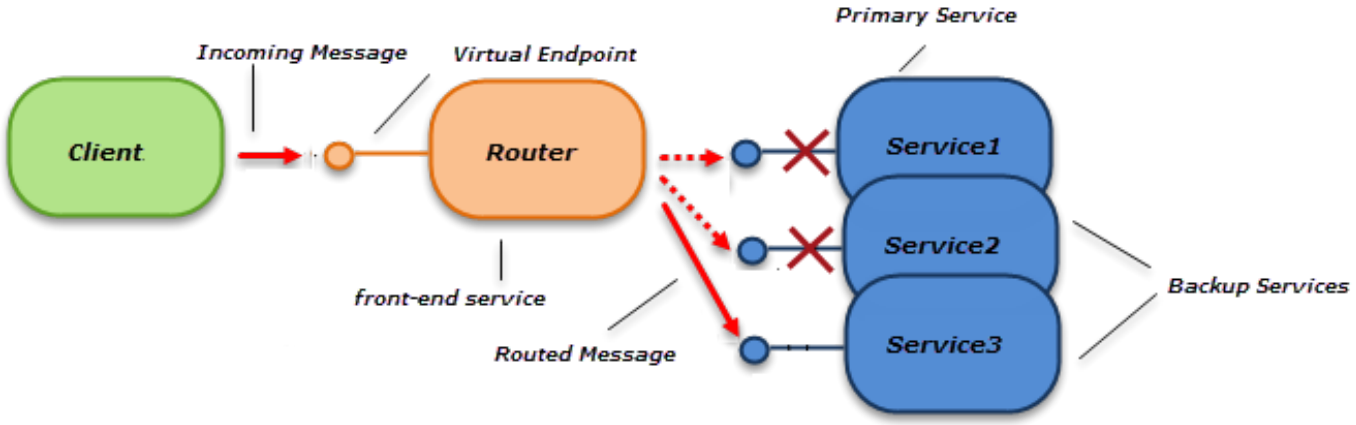
Failover using RoutingService

You can implement failover using the `RoutingService`. The `RoutingService`provides a built-in supports of basic level fault tolerance to cope with the run-time communication errors. You can define different lists of alternative endpoints (backup endpoints), when defining filter table, and that will be used by the`RoutingService` in case of communication failure with the initial target endpoint.

Let's implement failover using the `RoutingService`. But first please note that I'll continue to use the same service '`ComplexNumberCalculator`' of the previous post (Part II of the series) throughout this post too. This service is configured with following endpoints and hosted in a console application-

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

I'll use three instances of this service for this demo; one as a primary service and the other two as a backup services.



Failover using RoutingService

Next task would be to configure the `RoutingService` to support failover. So First I've configured the `RoutingService` with the following virtual endpoint-

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

Next I've defined three target endpoints-

```
<client>
        <endpoint address="http://localhost:8081/ComplexNumberCalculator1/binary"
binding="basicHttpBinding"
                  contract="*" name="BinaryOperation1" />
        <endpoint address="http://localhost:8081/ComplexNumberCalculator2/binary"
binding="basicHttpBinding"
                  contract="*" name="BinaryOperation2" />
        <endpoint address="http://localhost:8081/ComplexNumberCalculator3/binary"
binding="basicHttpBinding"
                  contract="*" name="BinaryOperation3" />
</client>
```

For the sake of simplicity I've used only one endpoint of the`ComplexNumbrCalculator` service dealing **Binary Operations** only.

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

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

Next I've defined following filter using `MatchAll filterType`-

```
<filters>
         <filter name="BinaryOperationFilter" filterType="MatchAll" />
</filters>
```

Next I've defined a list of backup endpoints within the **<backupLists>** element like down below-

```
<backupLists>
         <backupList name="BackUps">
           <add endpointName="BinaryOperation2"/>
           <add endpointName="BinaryOperation3" />
         </backupList>
</backupLists>
```

Next I've mapped '`BinaryOperationFilter`' filter to the first target service endpoint '`BinaryOperation1`' in

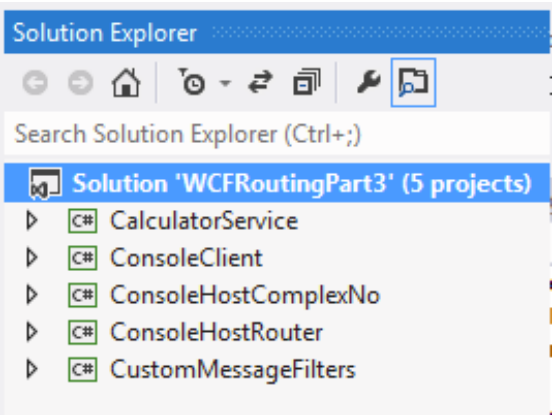
the filter table 'RoutigTable' and associated the filter table with the list of the backup endpoints-

```
<filterTables>
          <filterTable name="RoutingTable">
            <add filterName="BinaryOperationFilter" endpointName="BinaryOperation1" backupList="BackUps"
/>
          </filterTable>
</filterTables>
```

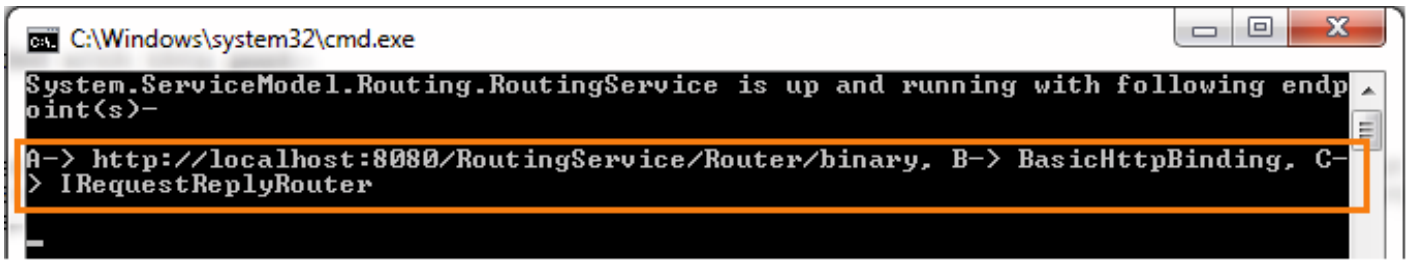
Finally I've configured the client application with following single endpoint-

```
<client>
        <endpoint address="http://localhost:8080/RoutingService/Router/binary"
binding="basicHttpBinding"
                  contract="IUnaryOperation" name="BasicHttpBinding_IUnaryOperationr" />
</client>
```

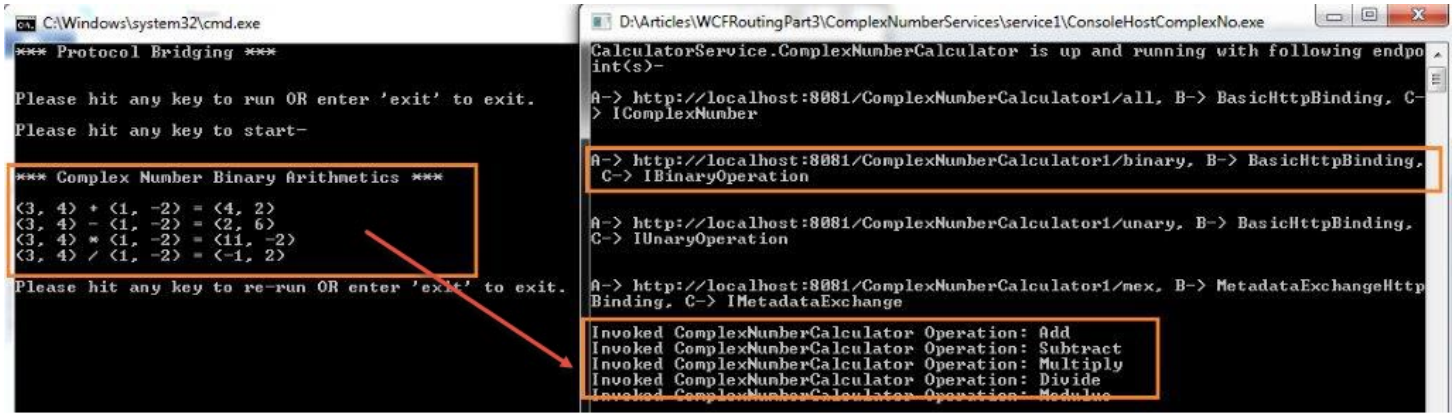
Below is the screen shot of the solution provided with this post-



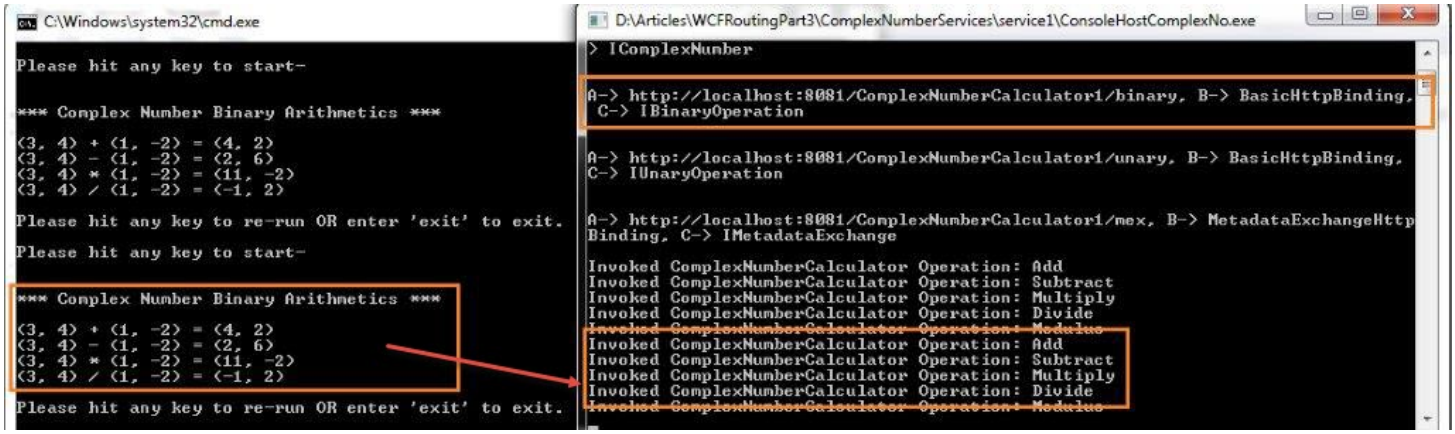
Now set **ConsoleClient** & **ConsoleHostRouter** projects as **Start Up projects** and hit **Ctrl+F5** keys in order to run the projects. Next run the**WCFRoutingPart3\ComplexNumberServices\StartThreeServices.cmd** file (see the sample code) from the **Visual Studio Developer Command Prompt** (in**Administrator** mode) in order to start ComplexNumberCalculator1,ComplexNumberCalculator2 and ComplexNumberCalculator3 services.



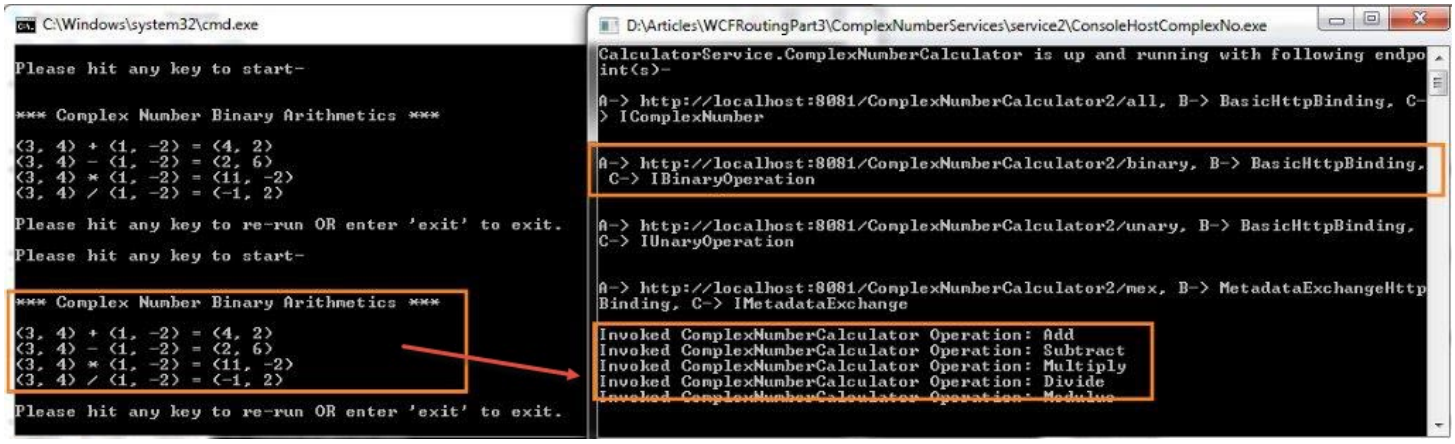
Now minimize the RoutingService console window and press any key on the console client window; you can verify that the complex number **Binary Operations**are routed to the ComplexNumberCalculator1 service by the intermediaryRoutingService.



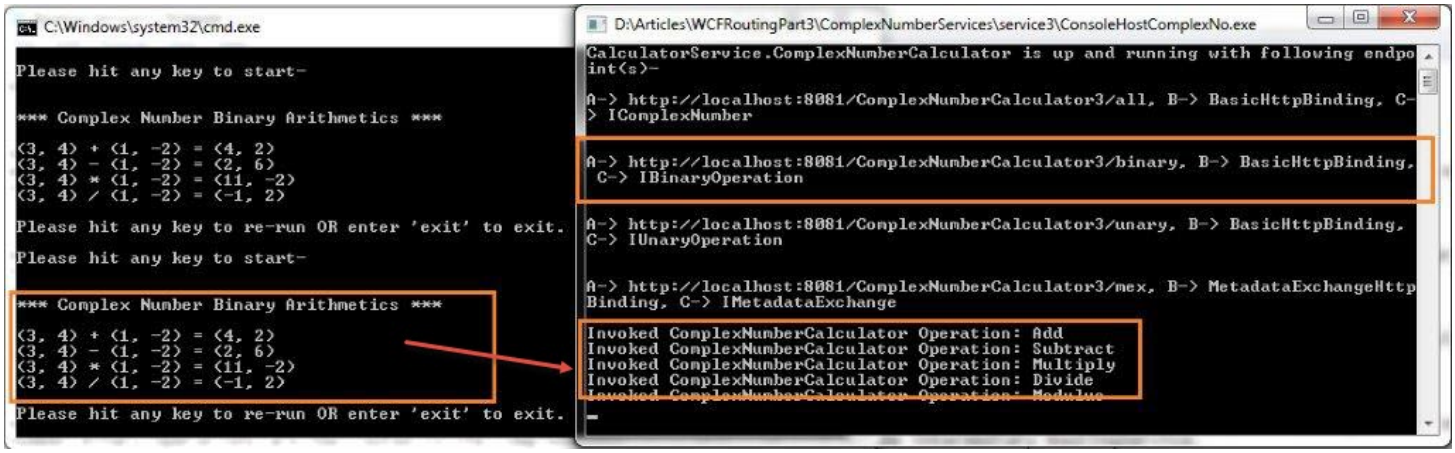
If you again press any key on the console client window; you will see that the incoming messages are still routed to the same `ComplexNumberCalculator1`service as expected by the intermediary `RoutingService`.



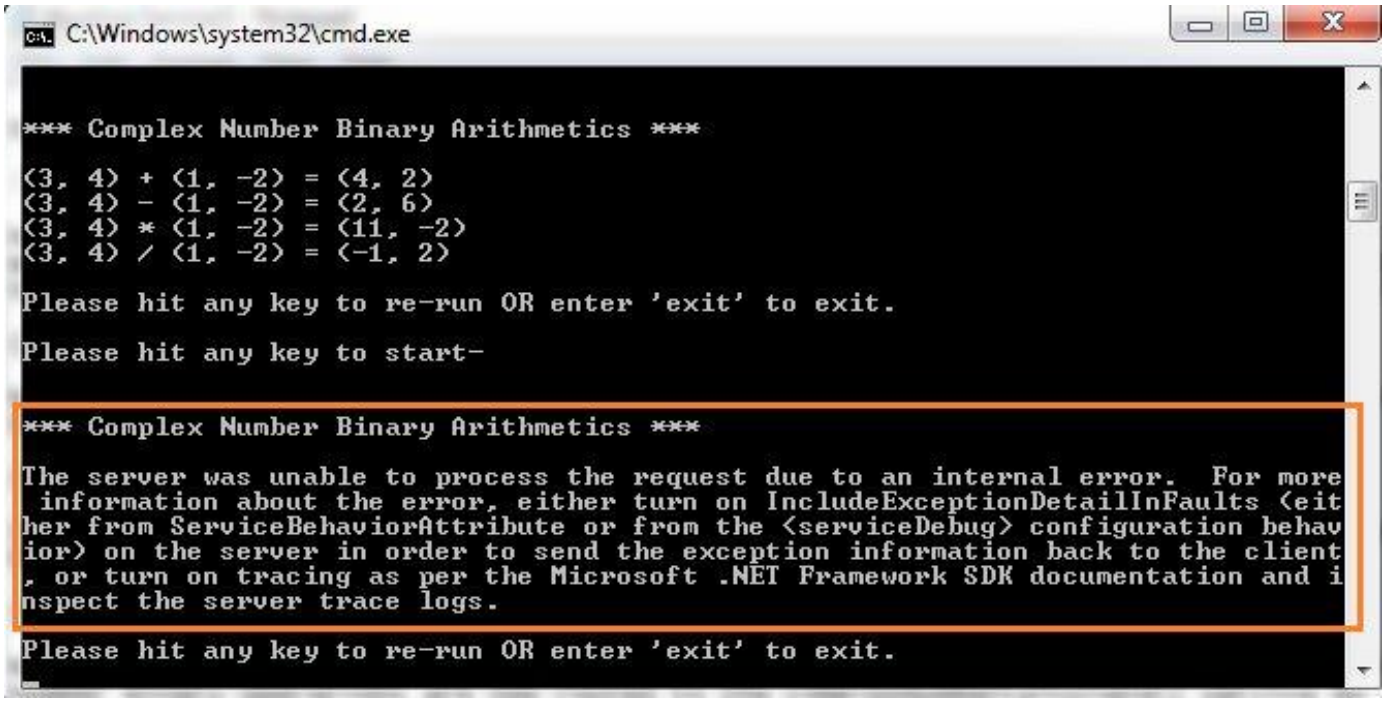
Now just close the console window that is running `ComplexNumberCalculator1`and again press any key on the console client window; you will notice that the complex number **Binary Operations** are now routed to the`ComplexNumberCalculator2` service by the intermediary `RoutingService`.



Next close the console window that is running `ComplexNumberCalculator2` and press any key on the console client window one more time; you can verify that the complex number **Binary Operations** are routed to the`ComplexNumberCalculator3` service this time by the intermediary`RoutingService`.



Finally close the console window that is running `ComplexNumberCalculator3`and press any key on the console client window last time, you will face an error this time as there is no more backup service available in the backup list to route the incoming messages. Actually this is a worst case scenario and we should have enough backup services in our backup list.



So you have seen that the `RoutingService` routed the incoming messages to the next available service in each failover.

Load Balancing

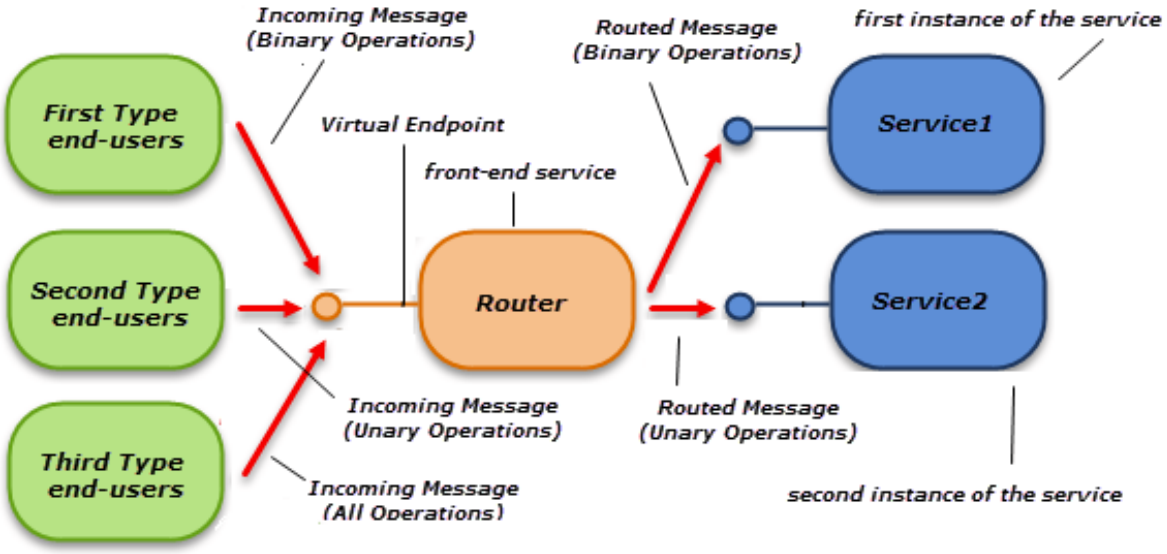
Apart from high availability and reliability, performance is also of very key importance for a critical service. Performance refers to the time taken by a service to complete a request. The service should be able to provide high speed request processing to its end-user(s) in peak loads too. The performance of a service can be improved using the load balancing techniques. Multiple instances of a service can be deployed on various machines of a distributed environment in order to maintain acceptable performance. When multiple concurrent requests are received, they are typically distributed among the available machines (services) by using an algorithm (e.g. **round-robin approach**, **random approach**, **weighted round-robin** etc.). The role of an algorithm is to determine the machine (service) with the least active requests processing.

We can also implement the **Load Balancing** using the `RoutingService`. Let's consider a scenario using our `ComplexNumberCalculator` service. Suppose that there are three types of end-users of the service; first type that can perform only**Binary Operations**, second type that can perform only **Unary Operations** and the third type that can perform **Binary** as well as **Unary Operations**. A single instance of the `ComplexNumberCalculator` service can easily handle these three types of end-users requests. But in peak loads, performance could be a concern (assumption for this demo). So in order to provide high speed processing in peak loads, we can load balance the incoming requests based on the content or using some algorithms (e.g. **round-robin approach**, **random approach**, **weighted round-robin** etc.).

I'll demonstrate the Load Balancing using the `RoutingService` in coming sections by using the content-based routing technique as well as using the round robin approach for the scenario I've described above.

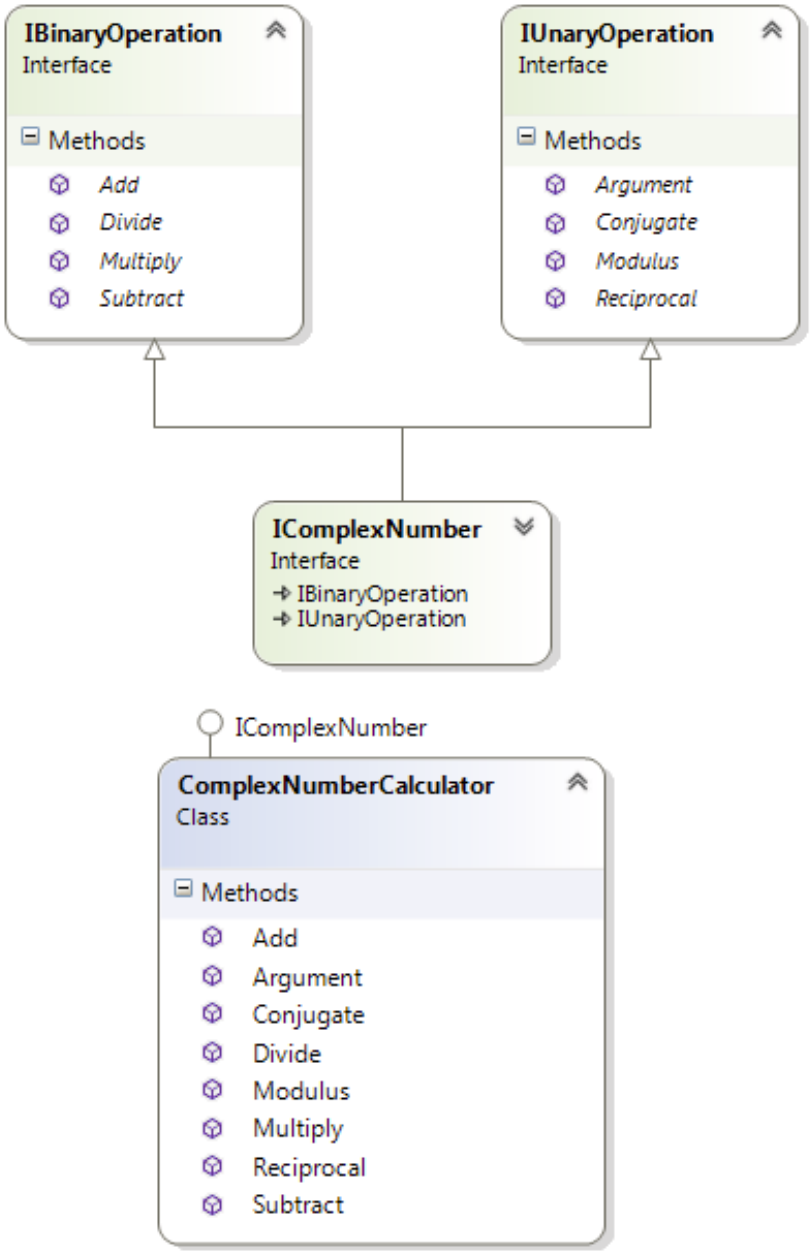
Load Balancing using Content-based Routing

Let's consider two instances of the `ComplexNumberCalculator` service instead of one to load balance the incoming requests based on the content of the message (operation types). The first instance would process the **Binary Operations** only while the second one would process the **Unary Operations** only. So the **first type end-users** requests would be processed by the first instance of the`ComplexNumberCalculator` service and the **second type end-users** requests would be processed by the second instance of the `ComplexNumberCalculator`service. What about the **third type end-users**? How its request would be processed? Note that the third type end-users have access of both types of operations: **Binary** as well as **Unary**. Well the **third type end-users** request would be partitioned based on the operation requested (content) and would be forwarded to the concern instance for the processing; means **Binary Operations** requests would be forwarded to the first instance of the `ComplexNumberCalculator`service for the processing and **Unary Operations** requests would be forwarded to the second instance of the `ComplexNumberCalculator` service for the processing.



Content-based Load Balancer

Now as per the design of our `ComplexNumberCalculator` service (see theprevious post for the details and following class diagram), **first type end-users** will use the `IUnaryOperation` service contract, **second type end-users** will use the`IUnaryOperation` service contract and the **third type end-users** will use the`IComplexNumber` service contract in order to communicate with the`ComplexNumberCalculator` service via intermediary `RoutingService`.



So in order to simulate the **Load Balancing** for our scenario, I've configured the client application with following three endpoints (one for each end-users type)-

```
<client>
        <endpoint address="http://localhost:8080/RoutingService/Router"
          binding="basicHttpBinding" contract="IUnaryOperation" name="firstTypeEndUsers" />
        <endpoint address="http://localhost:8080/RoutingService/Router"
          binding="basicHttpBinding" contract="IUnaryOperation" name="secondTypeEndUsers" />
        <endpoint address="http://localhost:8080/RoutingService/Router"
          binding="basicHttpBinding" contract="IComplexNumber" name="thirdTypeEndUsers" />
</client>
```

Let's re-configure the `RoutingService` for our **content-based load balancer**. So first I've re-defined the following two target endpoints (one for each instance of the`ComplexNumberCalculator` service) -

```
<client>
        <endpoint address="http://localhost:8081/ComplexNumberCalculator1/binary"
binding="basicHttpBinding"
                  contract="*" name="binaryOperationInstance" />
```

```
        <endpoint address="http://localhost:8081/ComplexNumberCalculator2/unary"
binding="basicHttpBinding"
                  contract="*" name="unaryOperationInstance" />
 </client>
```

Next task would be to define the filters following by configuring the filter table using the same for our **content-based load balancer** to full-fill the conditions of our described scenario. But before doing the same, let's examine the action values of the supported operations for each service contract of the`ComplexNumberCalculator` service by browsing to the **WSDL** definition of the service and you'll find there three **<portType>** elements.

Below are the action values of the `IBinaryOperation` **portType**-

```
http://tempuri.org/IBinaryOperation/Add
http://tempuri.org/IBinaryOperation/Subtract
http://tempuri.org/IBinaryOperation/Multiply
http://tempuri.org/IBinaryOperation/Divide
```

Below are the action values of the `IUnaryOperation` **portType**-

```
http://tempuri.org/IUnaryOperation/Modulus
http://tempuri.org/IUnaryOperation/Argument
http://tempuri.org/IUnaryOperation/Conjugate
http://tempuri.org/IUnaryOperation/Reciprocal
```

Below are the action values of the `IComplexNumber` **portType**-

```
http://tempuri.org/IBinaryOperation/Add
http://tempuri.org/IBinaryOperation/Subtract
http://tempuri.org/IBinaryOperation/Multiply
http://tempuri.org/IBinaryOperation/Divide
http://tempuri.org/IUnaryOperation/Modulus
http://tempuri.org/IUnaryOperation/Argument
http://tempuri.org/IUnaryOperation/Conjugate
http://tempuri.org/IUnaryOperation/Reciprocal
```

You'll observe that instead of `IComplexNumber` service contract,`IBinaryOperation` & `IUnaryOperation` service contracts are part of the action values for the `IComplexNumber` **portType**. For the **Binary Operations**,`IBinaryOperation` service contract is used and for the **Unary Operations**,`IUnaryOperation` service contract is used. Why? Because the `IComplexNumber`is an empty service contract & it implements `IUnaryOperation` &`IBinaryOperation` service contracts respectively (**interface inheritance**); it doesn't define any operation. In fact all binary and unary operations are members of the `IBinaryOperation` & `IUnaryOperation` service contracts not the members of the `IComplexNumber` service contract. So it is omitted in the action values. Actually the role of the `IComplexNumber` service contract is to just provide an access level to binary & unary complex number operations.

Now after examine the action values, you can define filters for the RoutingService using the **Action values** or using the **XPath Expressions** easily. But I would prefer to use a different approach. I'll create a **custom message filter** for our **content-based load balancer** that will inspect the **Action value** of each incoming message for a particular service contract (`IBinaryOperation` or `IUnaryOperation`) and will return the `Boolean` result accordingly.

So next I've created a custom message filter `ServiceContractMessageFilter`as I've described above. Below is the code of the same-

```
namespace CustomMessageFilters
{
    public class ServiceContractMessageFilter : MessageFilter
    {
        string _serviceContractName;

        public ServiceContractMessageFilter(string serviceContractName)
        {
            if (string.IsNullOrEmpty(serviceContractName)) { throw new
ArgumentNullException("serviceContractName"); }

            this._serviceContractName = serviceContractName;
        }

        public override bool Match(Message message)
        {
            return message.Headers.Action.Contains(_serviceContractName);
        }

        public override bool Match(MessageBuffer buffer)
        {
            return buffer.CreateMessage().Headers.Action.Contains(_serviceContractName);
        }
    }
}
```

Next I've re-defined following filters using the custom message filter`ServiceContractMessageFilter`-

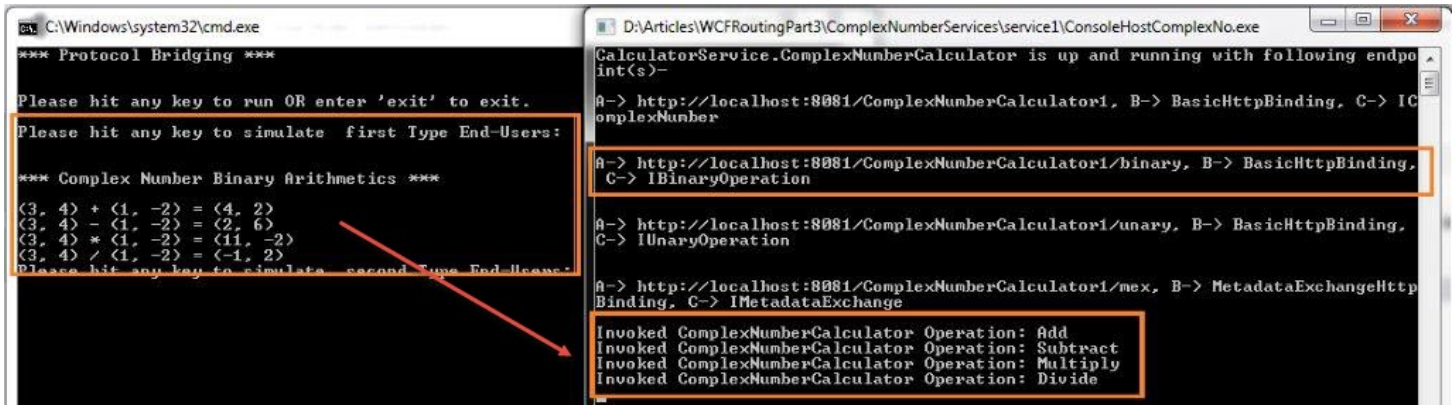
```
<filters>
          <filter name="serviceContractFilter1" filterType="Custom"
customType="CustomMessageFilters.ServiceContractMessageFilter, CustomMessageFilters"

filterData="IBinaryOperation"/>
          <filter name="serviceContractFilter2" filterType="Custom"
customType="CustomMessageFilters.ServiceContractMessageFilter, CustomMessageFilters"

filterData="IUnaryOperation"/>

</filters>
```

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

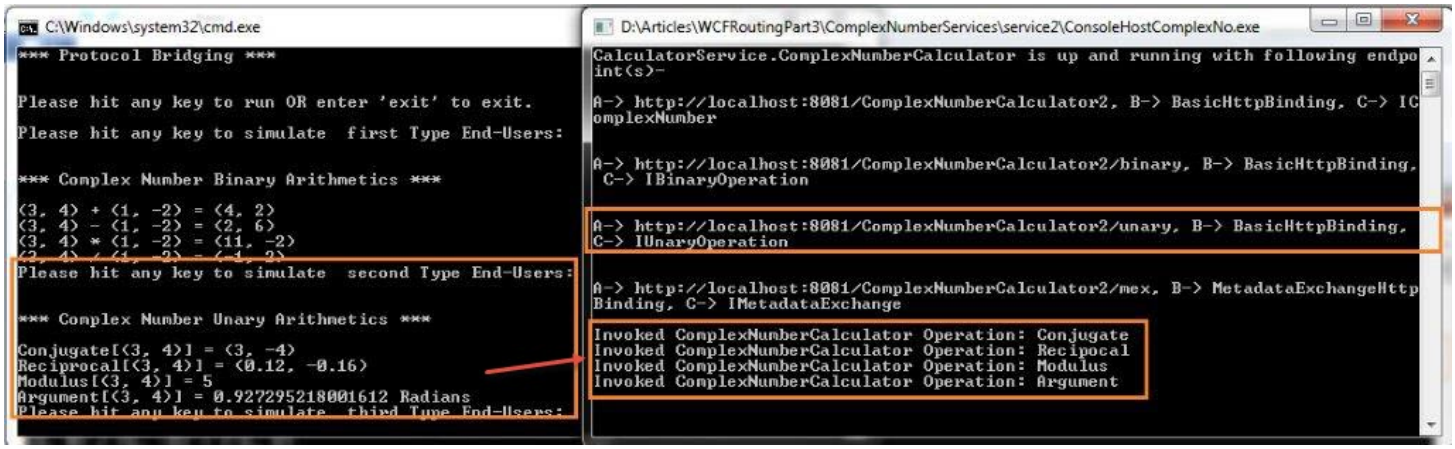
```
<filterTables>
          <filterTable name="RoutingTable">
            <add filterName="serviceContractFilter1" endpointName="binaryOperationInstance"/>
            <add filterName="serviceContractFilter2" endpointName="unaryOperationInstance"/>
          </filterTable>
 </filterTables>
```

Let's realize **Content-based Load Balancer**. Just set again **ConsoleClient** &**ConsoleHostRouter** projects as **Start Up projects** and hit **Ctrl+F5** keys in order to run the projects. Now minimize the `RoutingService` console window and run the**WCFRoutingPart3\ComplexNumberServices\StartTwoServices.cmd** file from the**Visual Studio Developer Command Prompt** (in **Administrator** mode) in order to start `ComplexNumberCalculator1` & `ComplexNumberCalculator2` services.

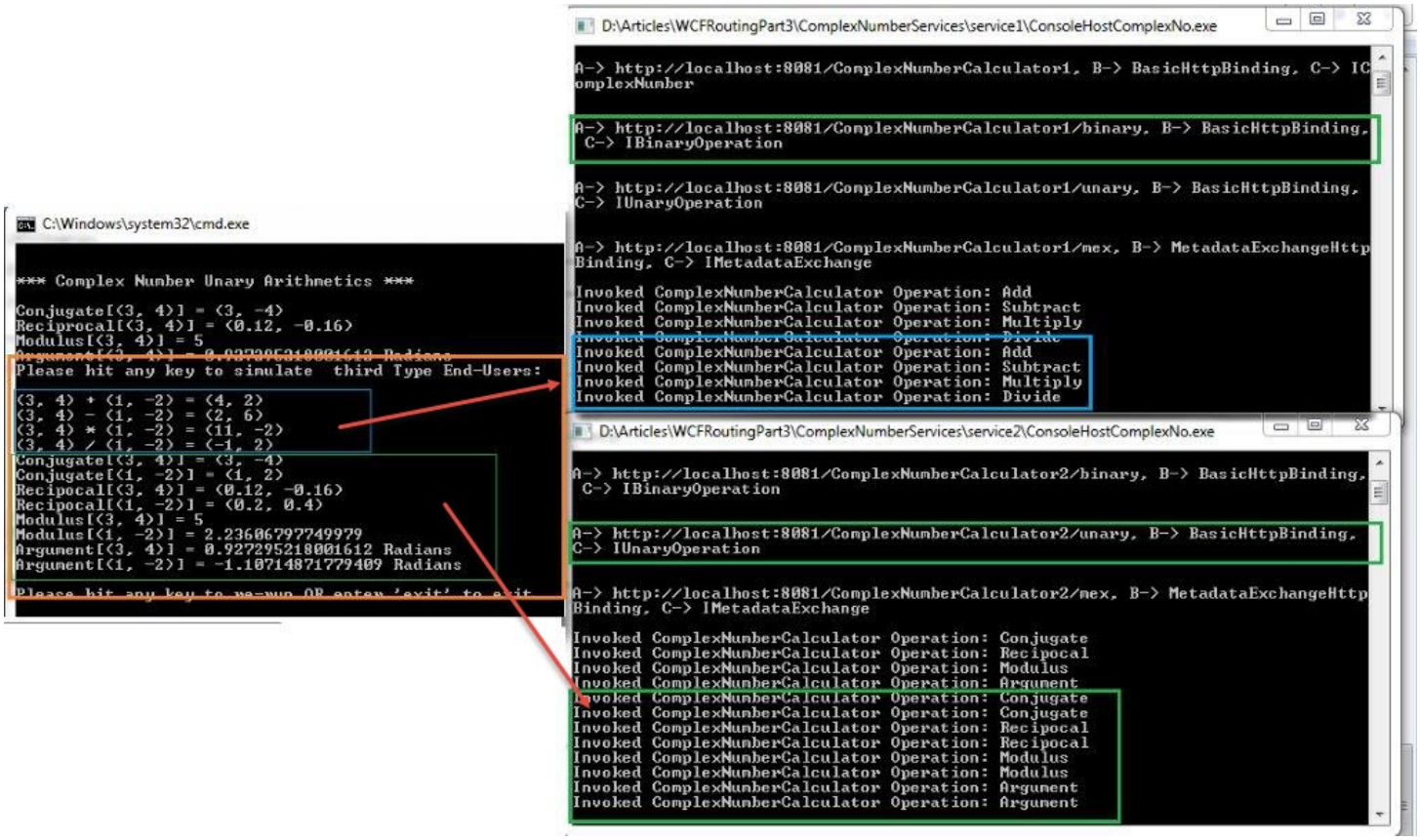
Next press any key on the console client window; you can verify that the **first type end-users** requests (complex number **Binary Operations**) are routed to the first instance of the `ComplexNumberCalculator` service (`ComplexNumberCalculator1`) by the intermediary `RoutingService` to be processed.



Next press any key on the console client window; you can verify that the **second type end-users** requests (complex number **Unary Operations**) are routed to the second instance of the `ComplexNumberCalculator` service (`ComplexNumberCalculator2`) by the intermediary `RoutingService` to be processed.



Next press any key on the console client window, you can verify that the **third type end-users** requests are partitioned. **Unary Operations** requests are routed to the first instance of the `ComplexNumberCalculator` service (`ComplexNumberCalculator1`) while **Binary Operations** requests are routed to the second instance of the `ComplexNumberCalculator` service (`ComplexNumberCalculator2`) by the intermediary `RoutingService` to be processed.



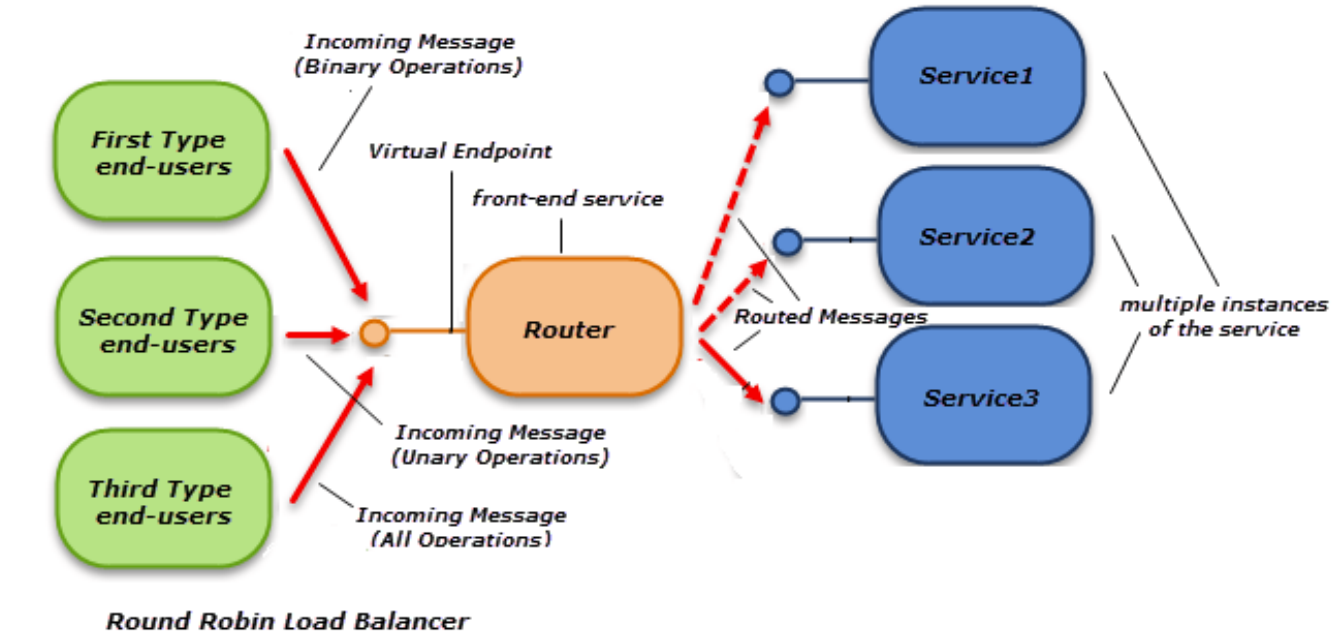
So this was the simple **Content-based Load Balancer** using the `RoutingService`. You can build your **Content-based Load Balancer** by using multiple instances of the service(s) as per your requirement.

Load Balancing using Round Robin Approach

In this section I'll demonstrate the **Load Balancing** using the `RoutingService`based on the **Round Robin** approach. But before that let's try to understand what is**Round Robin** algorithm approach based load balancing? In round-robin approach, the incoming requests (messages) are assigned to a list of the servers (services) on a rotating basis by the **request sprayer**. The first incoming request (message) is allocated to a server (service) picked randomly from the participating group (list of the services) and the subsequent requests (messages) would be redirected by the**request sprayer** by following the circular order. Once a server (service) is assigned a request (message) to process, the server (service) is pushed to the end of the list of the servers (services). This keeps the servers (services) equally assigned.

Let's discuss this in detail. Suppose that there are three services in the group in the order: **{service1, service2, service3}**. Let's say the first incoming message is allocated to the **service1** by the **request sprayer**. So the next incoming messages, say second, third, fourth, fifth ..., would be allocated in the sequence: **service2, service3, service1, service2** ... by the **request sprayer** by following the circular order.

Let's simulate the **Round Robin** approach based load balancing using the`RoutingService`. I'll use three instances of our `ComplexNumberCalculator`service in the participant group in order to handle the incoming messages in circular order to provide high speed messages processing in peak loads. Please note that in the **Round Robin-based Load Balancer**, `RoutingService` will act as a '**Request Sprayer**'.



So first I've re-configured the `RoutingService` with the following three target endpoints (participants of the group) -

```
<client>
        <endpoint address="http://localhost:8081/ComplexNumberCalculator1" binding="basicHttpBinding"
                  contract="*" name="firstInstance" />
        <endpoint address="http://localhost:8081/ComplexNumberCalculator2" binding="basicHttpBinding"
                  contract="*" name="secondInstance" />
        <endpoint address="http://localhost:8081/ComplexNumberCalculator3" binding="basicHttpBinding"
                  contract="*" name="thirdInstance" />
</client>
```

Next we'll need to re-define the filters of the `RoutingService` in order to spray the incoming messages to a group of the services on rotation basis. As there is no built-in filter available in the **WCF** to follow the round robin approach, we'll need to create a custom filter for the same. But there is already a sample custom filter based on the round robin approach available on msdn and I'm going to use the same for this demo.

So next I've re-defined following filters using the **custom Round Robin Message Filter**-

```
<filters>
          <filter name="roundRobinContractFilter1" filterType="Custom"
customType="CustomMessageFilters.RoundRobinMessageFilter, CustomMessageFilters"
filterData="roundRobinGroup"/>
          <filter name="roundRobinContractFilter2" filterType="Custom"
customType="CustomMessageFilters.RoundRobinMessageFilter, CustomMessageFilters"
filterData="roundRobinGroup"/>
          <filter name="roundRobinContractFilter3" filterType="Custom"
customType="CustomMessageFilters.RoundRobinMessageFilter, CustomMessageFilters"
filterData="roundRobinGroup"/>
</filters>
```

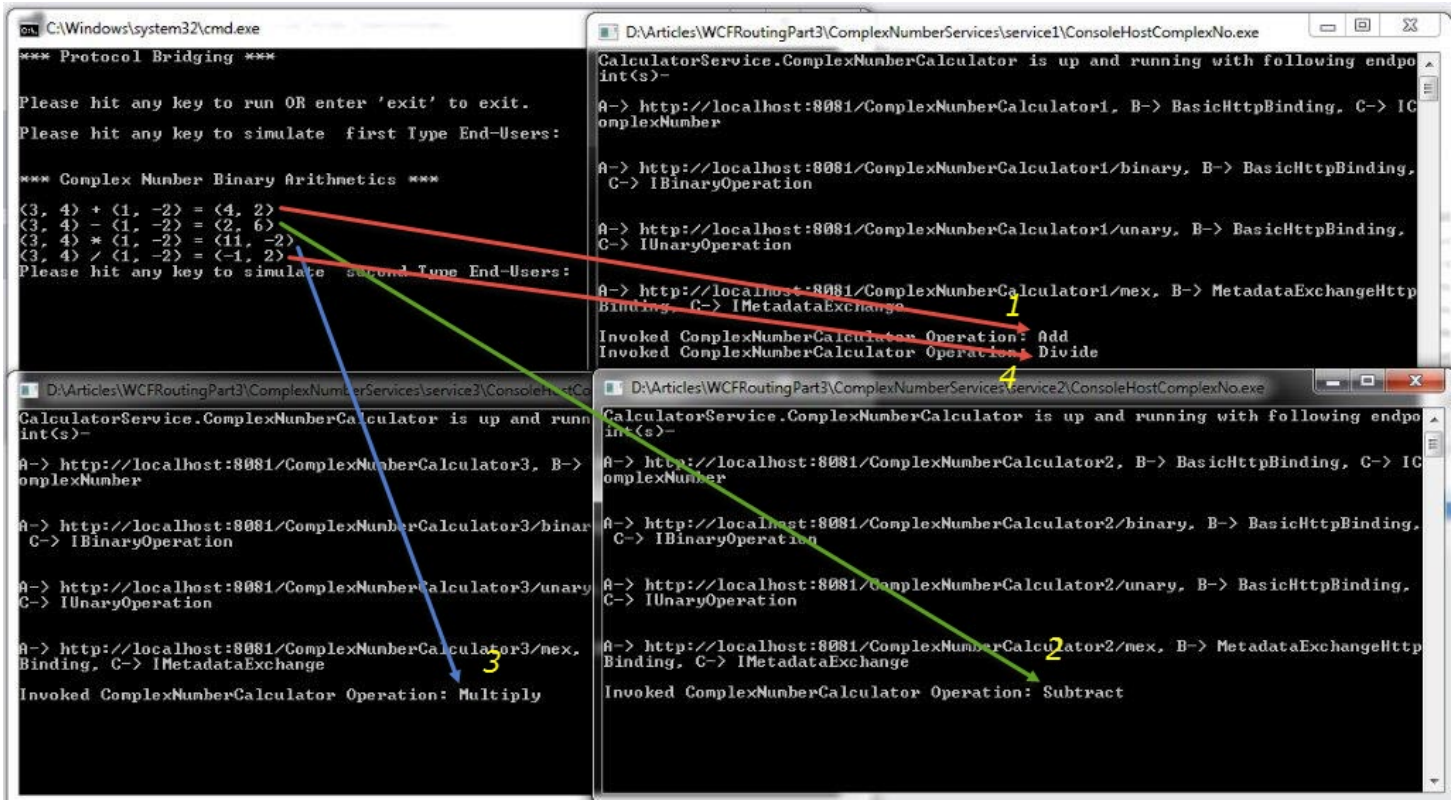
Finally I've re-mapped each filter to the respective target service endpoint in the filter table '`RoutigTable`' as below-

```
<filterTables>
     <filterTable name="RoutingTable">
            <add filterName="roundRobinContractFilter1" endpointName="firstInstance"/>
            <add filterName="roundRobinContractFilter2" endpointName="secondInstance"/>
```

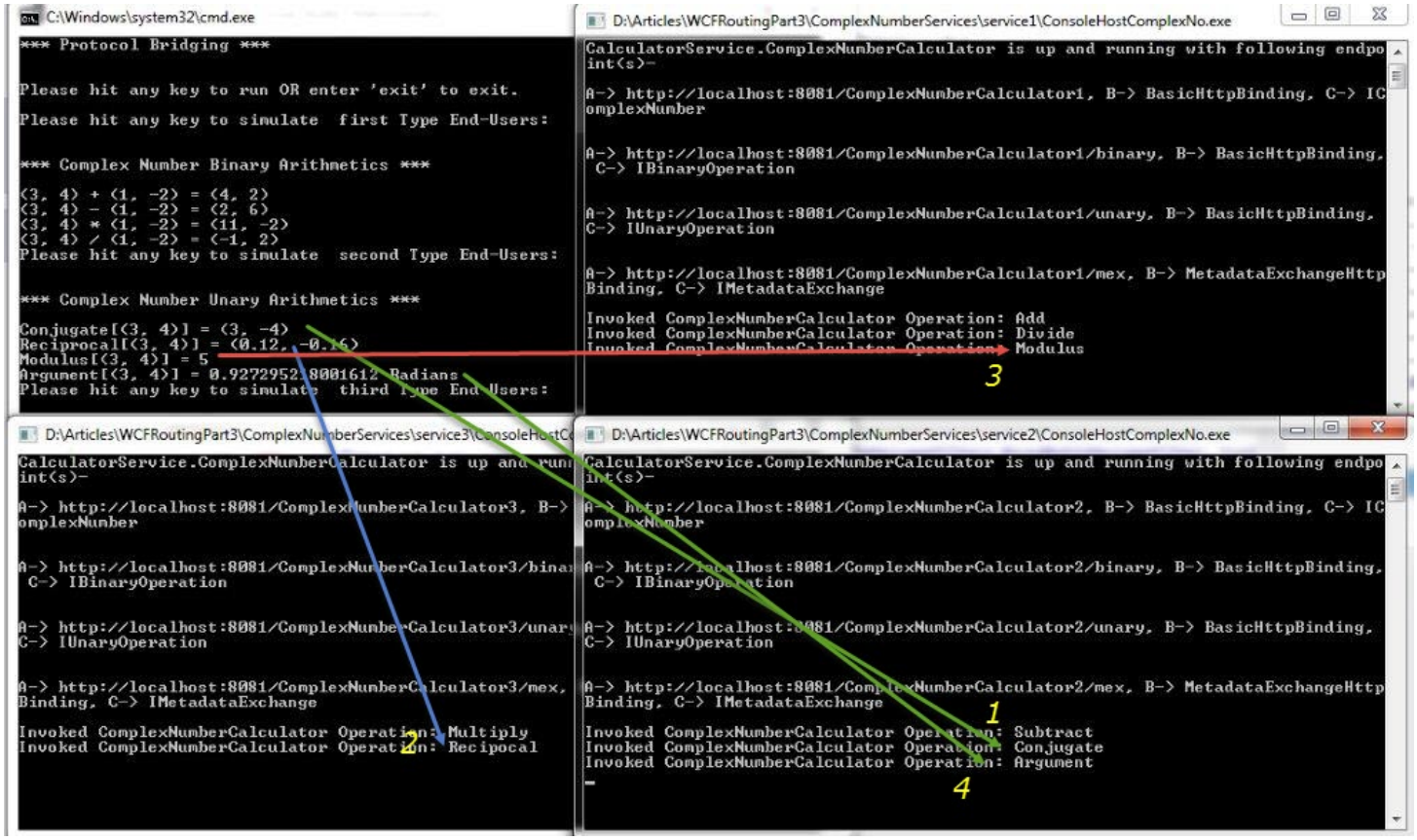
```
            <add filterName="roundRobinContractFilter3" endpointName="thirdInstance"/>
        </filterTable>
    </filterTables>
```

That's it. Let's run our demo. Just set **ConsoleClient** & **ConsoleHostRouter** projects as **Start Up projects** and hit **Ctrl+F5** keys in order to run the projects. Next minimize the `RoutingService` console window and run the**WCFRoutingPart3\ComplexNumberServices\StartThreeServices.cmd** file from the **Visual Studio Developer Command Prompt** (in **Administrator** mode) in order to start `ComplexNumberCalculator1`, `ComplexNumberCalculator2` &`ComplexNumberCalculator3` services.

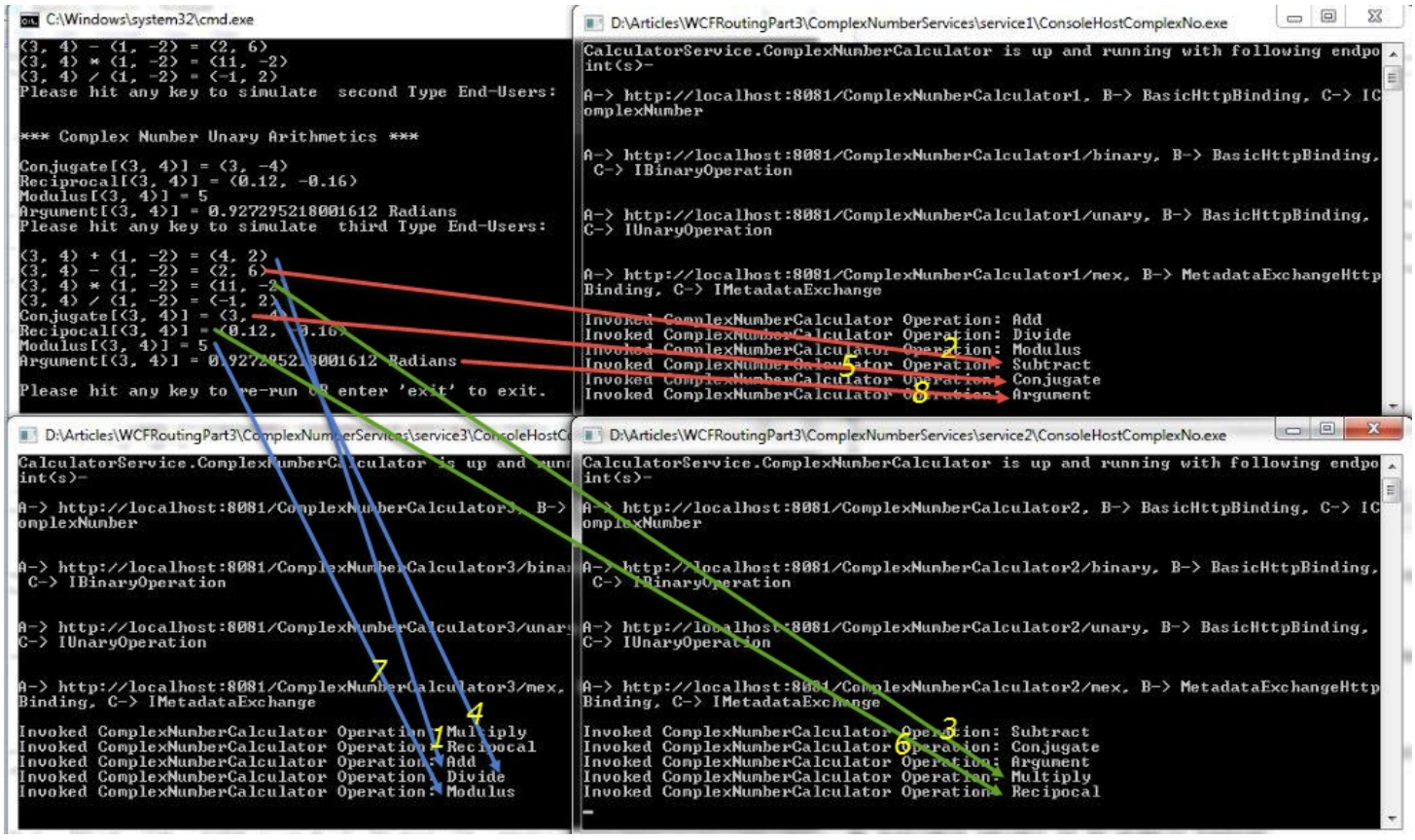
Next press any key on the console client window; you can verify that the **first type end-users requets** are sprayed to the list of the services in the group on rotation basis by the intermediary `RoutingService` (**Request Sprayer**).



Again press any key on the console client window; you can verify that this time**second type end-users** requets are sprayed to the list of the services in the group on rotation basis by the intermediary `RoutingService` (**Request Sprayer**).



Press one more time any key on the console client window; you can verify that this time **third type end-users** requets are sprayed to the list of the services in the group on rotation basis by the intermediary `RoutingService` (**Request Sprayer**).



So you have seen that in each case, requests are equally distributed among the available servers in an orderly

manner.

Conclusion

So you have seen that how can we implement **Failover** or **High-Availability** and**Load Balancing** features using the `RoutingService` easily. These are very important features and should be considered very carefully as per your need and requirement. Till the next part of the series, happy coding.

History

- 7th Jun, 2014 -- Article updated (Added a new entry for the fourth part of the series in 'All Posts' section)
- 29th May, 2014 -- Article updated (Added the table of contents section)
- 28th 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.**