



CPOL

★★★★★

4.73 (8 votes)


```

    <endpoint address="Regular" binding="basicHttpBinding"
contract="RegularComplexNoCalc.IComplexNumber" />
    <endpoint address="mex" kind="mexEndpoint" />
    <host>
        <baseAddresses>
            <add baseAddress="http://localhost:8081/ComplexNumberCalculator" />
        </baseAddresses>
    </host>
</service>
</services>

```

I've configured **ComplexNumberCalculator** service (v2.0) with following single endpoint along with a standard **mex** endpoint-

```

<services>
    <service name="ExtendedComplexNoCalc.ComplexNoCalc">
        <endpoint address="Extended" binding="basicHttpBinding"
contract="ExtendedComplexNoCalc.IComplexNumber" />
        <endpoint address="mex" kind="mexEndpoint" />
        <host>
            <baseAddresses>
                <add baseAddress="http://localhost:8082/ComplexNumberCalculator" />
            </baseAddresses>
        </host>
    </service>
</services>

```

Configuring the RoutingService

The problem described in the previous section can be resolved using the **RoutingService** by following two techniques-

- Context-based routing technique
- Content-based routing technique

In **Context-based** routing technique, a specific endpoint is exposed for each version of the service in the router and incoming messages are uniquely routed to a specific version of the service based on the specific endpoint on which messages arrive.

So in our case we'll need to expose two endpoints in the router one for each service version. But this technique isn't feasible because we'll need to expose more endpoints in the router for each newer version of the service.

In **Content-based** routing technique, a single specific endpoint is exposed for all version of the service in the router and incoming messages are uniquely routed to a specific version of the service based on the content of the message.

So in our case we'll need to expose a single endpoint in the router for two versions of the service. This technique would be preferable because it is based on inspecting content of the incoming messages in order to differentiate between requests for the different service versions.

But how would we differentiate between requests for the two service versions in our case by inspecting content of the incoming messages? Because **Binary Operations** are common in both versions of the service,

so requests for the two service versions would be identical in all aspects (their input parameters and return types are same) in this case. The content of the incoming messages would not be unique enough to determine the correct target service version. Although in case of **Unary Operations** requests, we could easily determine the correct target service version. But in overall, we can't use **Action** or **XPath** filter types in order to determine the correct target service version as requests for **Binary Operations** would be identical. You can verify the same by going through the **Action** values of each service version as shown below-

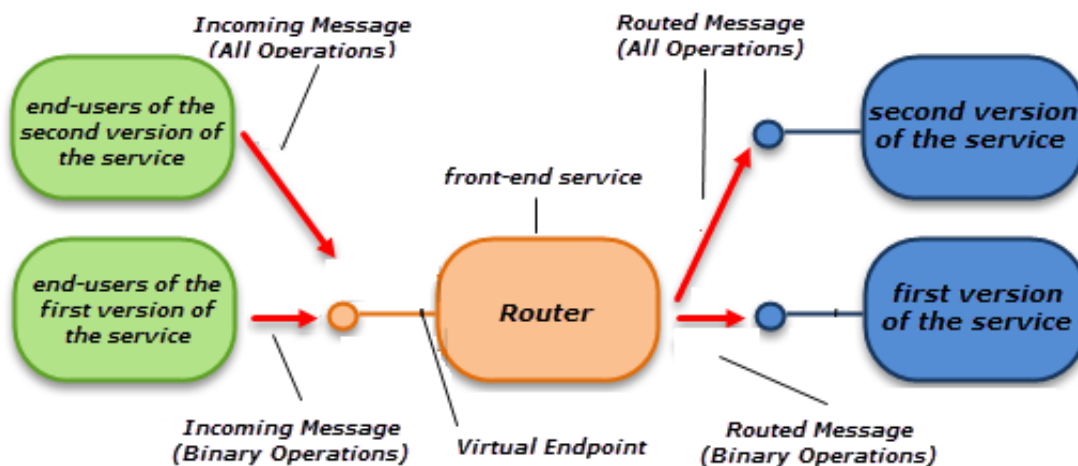
```
<!-- Action values of first version of the service-->

http://tempuri.org/IComplexNumber/Add
http://tempuri.org/IComplexNumber/Subtract
http://tempuri.org/IComplexNumber/Multiply
http://tempuri.org/IComplexNumber/Divide

<!-- Action values of second version of the service-->

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

So what would be the solution? Well this problem can be solved by inserting the service version information into a request message header. Each client application will insert the target service version information into the message header and router will use this version-specific information contained in the message header to route the incoming message to the appropriate version of the service.



Service Versioning

Now in our case, we'll insert a custom element '**Version**' into the request message header to indicate the target service version that the request message would be transmitted to. The custom element **Version** can have two possible values- v1.0 & v2.0. A value of v1.0 will indicate that the request message must be routed to the first version of the **ComplexNumberCalculator** service to be processed while a value of v2.0 will indicate that the request message would be routed to the second version of the **ComplexNumberCalculator** service to be processed.

So in our case, **SOAP** message will look like as shown down below-

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
<s:Header>
<Version xmlns="http://custom/namespace">v1.0</Version>
<To s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">http://localhost:8081/ComplexNumberC
alculator/Regular</To>
<Action s:mustUnderstand="1"
xmlns="http://schemas.microsoft.com/ws/2005/05/addressing/none">http://tempuri.org/IComplexNumber/Ad
d</Action>
</s:Header>
<s:Body>
...
</s:Body>
</s:Envelope>
```

In the above, '**http://custom/namespace**' is a namespace I've defined for the custom element inserted into the request message header.

I'll explain how we can insert a custom element into the message header in the next section. But before that, let's configure the **RoutingService** for our **Service Versioning** scenario.

So First I've configured the **RoutingService** with the following virtual endpoint-

```
<services>
  <service name="System.ServiceModel.Routing.RoutingService">
    <endpoint address="" 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 two target endpoints one for each service version-

```
<client>
  <endpoint address="http://localhost:8081/ComplexNumberCalculator/Regular"
binding="basicHttpBinding"
contract="*" name="firstVersion" />
  <endpoint address="http://localhost:8082/ComplexNumberCalculator/Extended"
binding="basicHttpBinding"
contract="*" name="secondVersion" />
</client>
```

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

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

Next I've defined a set of namespace prefix bindings using the the element as below-

```
<namespaceTable>
  <add prefix="s" namespace="http://schemas.xmlsoap.org/soap/envelope/" />
  <add prefix="cn" namespace="http://custom/namespace/" />
</namespaceTable>
```

Then I've configured second client application with following endpoint-

```
<system.serviceModel>
  <client>
    <endpoint address="http://localhost:8080/RoutingService/Router"
      binding="basicHttpBinding" contract="ExtendedComplexNoCalc.IComplexNumber"
      name="secondVersionEndUsers" />
  </client>
</system.serviceModel>
```

That's it. Nothing special.

Let's examine the client applications code and see how can we insert a custom element into the **SOAP** header of the outgoing message to indicate the target service version information. Below is the code of the first client application (end-users of the first version of the service)-

```
var cfV1 = new ChannelFactory<IComplexNumber>("firstVersionEndUsers");
var channelV1 = cfV1.CreateChannel();

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

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

z2.Real = 4D;
z2.Imaginary = 3D;

Console.WriteLine("*** Service Versioning: : end-users of the first version ***\n");
Console.WriteLine("\nPlease hit any key to run OR enter 'exit' to terminate.");
string command = Console.ReadLine();

while (command != "exit")
{
    Console.WriteLine("Please hit any key to simulate firstVersionEndUsers: ");
    Console.ReadLine();

    using (new OperationContextScope((IContextChannel)channelV1))
    {
        OperationContext.Current.OutgoingMessageHeaders.Add(MessageHeader.CreateHeader("Version",
"http://custom/namespace", "v1.0"));
        ComplexNumberArithmetics(channelV1, z1, z2);
    }

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

((IClientChannel)channelV1).Close();
```

In the above code, first I've created a client side proxy at runtime using the **ChannelFactory** class in order to invoke the members of the first version of the service. Now before doing the same, we'll need to insert a custom element for holding versioning information into the **SOAP** header of the outgoing message.

So next I've created an instance of the **OperationContextScope** class to get the current operation context scope, based on the client side proxy's **innerChannel**. Note that here **(IContextChannel)channelV1** represents the client side proxy's **innerChannel**. It is very important to note that, If you have created the client side proxy using the **svcutil** generated proxy class (in this

case `ComplexNumberClient` class, `ComplexNumberClient proxy = new ComplexNumberClient("firstVersionEndUsers")` at compile time, then you'll need to invoke `proxy.InnerChannel` in order to get the client side proxy's `innerChannel`. One more important point; the instance of the `OperationContextScope` class need to be created first, before accessing the outgoing message headers element from the context (`OperationContext.Current.OutgoingMessageHeaders`). If you do the same, the `OperationContext` will null and you'll face an exception- **System.NullReferenceException: {"Object reference not set to an instance of an object."}**

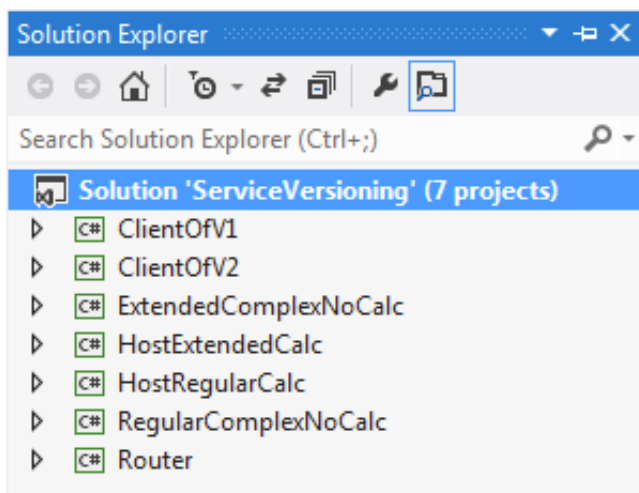
Next I've created a new custom element by invoking the `MessageHeader.CreateHeader` method and added it into the outgoing message headers by invoking the `OperationContext.Current.OutgoingMessageHeaders.Add` method. Here '**Version**', '**http://custom/namespace**' and '**v1.0**' represents name, namespace and value of the custom element respectively. Notice that here I've hard-coded the version of the target service. You can save it in the configuration file of the client application or in the database or even in the simple text file as per your requirement.

Finally I've invoked `ComplexNumberArithmetics` method to access the members of the first version of the service. You can find the code of the `ComplexNumberArithmetics` method in the sample code attached with this post.

The code of the second client application is similar except for the value of the '**Version**' custom element (v2.0). Please see the sample attached with this post.

Simulating the Service Versioning using the RoutingService

Now all is done. Let's run the demo application. Below is the screen shot of the solution-



Just set **ClientOfV1**, **ClientOfV2**, **HostExtendedCalc**, **HostRegularCalc** & **Router** projects as **Start Up projects** and hit **Ctrl+F5** keys in order to run the projects. Now minimize the **RoutingService** console window.

Next press any key on the first client application window (end-users of the first version of the service); you

can verify that the incoming messages are routed to the first version of the **ComplexNumberCalculator** service (**Regular**) by the intermediary **RoutingService** after inspecting the version information contained in the incoming messages header.

```

C:\Windows\system32\cmd.exe
*** Service Versioning: : end-users of the first version ***

Please hit any key to run OR enter 'exit' to terminate.
Please hit any key to simulate firstVersionEndUsers:
(3, 4) + (4, 3) = (7, 7)
(3, 4) - (4, 3) = (-1, 1)
(3, 4) * (4, 3) = (0, 25)
(3, 4) / (4, 3) = (0.96, 0.28)
Please hit any key to re-run OR enter 'exit' to terminate.

C:\Windows\system32\cmd.exe
RegularComplexNoCalc.ComplexNoCalc is up and running with following endpoint(s)-
A-> http://localhost:8081/ComplexNumberCalculator/Regular, B-> BasicHttpBinding,
C-> IComplexNumber
A-> http://localhost:8081/ComplexNumberCalculator/mex, B-> MetadataExchangeHttpB
inding, C-> IMetadataExchange
Invoked ADD operation.
Invoked SUBTRACT operation.
Invoked MULTIPLY operation.
Invoked DIVIDE operation.

```

Next press any key on the second client application window (end-users of the second version of the service); you can verify that the incoming messages are routed to the second version of the **ComplexNumberCalculator** service (**Extended**) by the intermediary **RoutingService** after inspecting the version information contained in the incoming messages header.

```

C:\Windows\system32\cmd.exe
*** Service Versioning: end-users of the second version ***

Please hit any key to run OR enter 'exit' to terminate.
Please hit any key to simulate secondVersionEndUsers:
(1, 2) + (2, 1) = (3, 3)
(1, 2) - (2, 1) = (-1, 1)
(1, 2) * (2, 1) = (0, 5)
(1, 2) / (2, 1) = (0.8, 0.6)
Conjugate[(1, 2)] = (1, -2)
Reciprocal[(1, 2)] = (0.2, -0.4)
Modulus[(1, 2)] = 2.23606797749979
Argument[(1, 2)] = 1.10714871779409 Radians
Please hit any key to re-run OR enter 'exit' to terminate.

C:\Windows\system32\cmd.exe
ExtendedComplexNoCalc.ComplexNoCalc is up and running with following endpoint(s)-
A-> http://localhost:8082/ComplexNumberCalculator/Extended, B-> BasicHttpBinding
, C-> IComplexNumber
A-> http://localhost:8082/ComplexNumberCalculator/mex, B-> MetadataExchangeHttpB
inding, C-> IMetadataExchange
Invoked ADD operation.
Invoked SUBTRACT operation.
Invoked MULTIPLY operation.
Invoked DIVIDE operation.
Invoked CONJUGATE operation.
Invoked RECIPROCAL operation.
Invoked MODULUS operation.
Invoked ARGUMENT operation.

```

Now just close all running applications (clients, services and router) and comment out following line of code from each client application (end-users of each service version) and re-build the solution in order to simulating the case of incoming messages without having '**Version Information**' in their headers.

```

...
using (new OperationContextScope((IContextChannel)channelV1))
{
    //OperationContext.Current.OutgoingMessageHeaders.Add(MessageHeader.CreateHeader("Version",
    "http://custom/namespace", "v1.0"));
    ComplexNumberArithmetics(channelV1, z1, z2);
}
...

```

Notice that I've already defined a filter in '**Configuring the RoutingService**' section to handle this type situation. Finally set **ClientOfV1**, **ClientOfV2**, **HostExtendedCalc**, **HostRegularCalc** & **Router** projects as **Start Up projects** and hit **Ctrl+F5** keys in order to run the projects and minimize the **RoutingService** console window.

Next just press any key on the first client application window (end-users of the first version of the service) followed by pressing any key on the second client application (end-users of the second version of the

service); you can verify that the incoming messages are routed to the second version of the **ComplexNumberCalculator** service (**Extended**) in both cases by the intermediary **RoutingService**. It is as per our expectation as we've already fingered this situation in the filter table.

```

C:\Windows\system32\cmd.exe
*** Service Versioning: : end-users of the first version ***
Please hit any key to run OR enter 'exit' to terminate.
Please hit any key to simulate firstVersionEndUsers:
<3, 4> + <4, 3> = <0, 0>
<3, 4> - <4, 3> = <0, 0>
<3, 4> * <4, 3> = <0, 0>
<3, 4> / <4, 3> = <0, 0>
Please hit any key to re-run OR enter 'exit' to terminate.

C:\Windows\system32\cmd.exe
*** Service Versioning: end-users of the second version ***
Please hit any key to run OR enter 'exit' to terminate.
Please hit any key to simulate secondVersionEndUsers:
<1, 2> + <2, 1> = <3, 3>
<1, 2> - <2, 1> = <-1, 1>
<1, 2> * <2, 1> = <0, 5>
<1, 2> / <2, 1> = <0.8, 0.6>
ConjugateI<1, 2>I = <1, -2>
ReciprocalI<1, 2>I = <0.2, -0.4>
ModulusI<1, 2>I = 2.23606797749979
ArgumentI<1, 2>I = 1.10714871779409 Radians
Please hit any key to re-run OR enter 'exit' to terminate.

C:\Windows\system32\cmd.exe
ExtendedComplexNoCalc.ComplexNoCalc is up and running with following endpoint(s)
A-> http://localhost:8082/ComplexNumberCalculator/Extended, B-> BasicHttpBinding
C-> IComplexNumber
A-> http://localhost:8082/ComplexNumberCalculator/mex, B-> MetadataExchangeHttpBinding, C-> IMetadataExchange
Invoked ADD operation.
Invoked SUBTRACT operation.
Invoked MULTIPLY operation.
Invoked DIVIDE operation.
Invoked ADD operation.
Invoked SUBTRACT operation.
Invoked MULTIPLY operation.
Invoked DIVIDE operation.
Invoked CONJUGATE operation.
Invoked RECIPROCAL operation.
Invoked MODULUS operation.
Invoked ARGUMENT operation.

```

So you have seen that how different versions of a service could be made available simultaneously using the **RoutingService**. Here we've achieved **Service Versioning** by inserting the service version information into the outgoing message headers but there are other techniques of **Service Versioning** that don't require client applications to pass additional information through outgoing messages headers. A message could be routed to the most recent or most compatible version of a service.

Multicasting

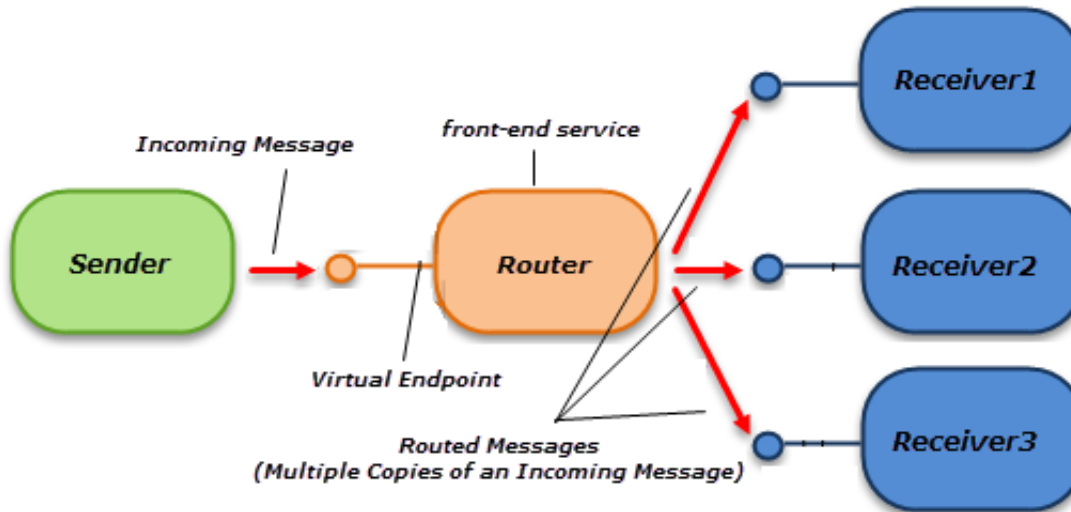
What is **Multicasting**? **Multicast** is the term used to describe communication where a piece of information is sent from one or more points to a set of other points. In **Multicasting**, sender ≥ 1 (i.e. there should be at least one sender) and receiver ≥ 0 (i.e. there may be no receivers).

In **Multicasting**, clients receive a stream of packets (data) only if they have previously subscribed to a specific multicast group and membership of a group is dynamic and controlled by the receivers. **Multicasting** is very useful if a group of end-users require a common set of data at the same time e.g. in stock exchanges, multimedia content delivery networks, IPTV applications (distance learning and televised company meetings), wireless networks and cable-TV etc. With the help of **Multicasting**, you can reach to many end-users by utilizing only one data stream and hence decreasing the amount of bandwidth and saving your money.

After brief description about **Multicasting**, let's see how can we implement **Multicasting** using the **WCF RoutingService**.

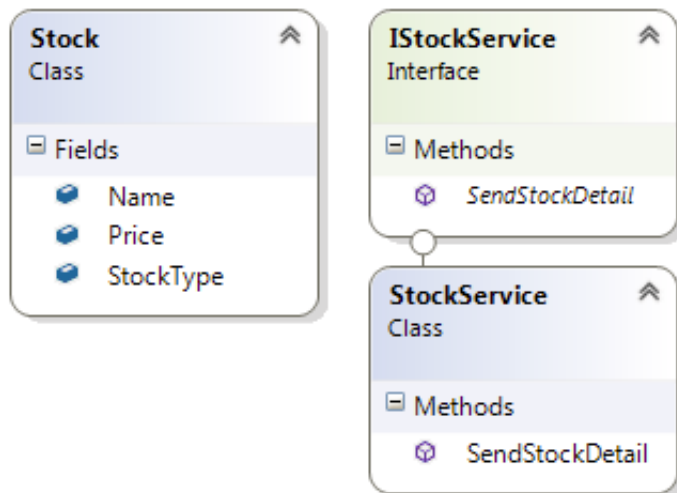
Multicasting Demo Service

I'll use a simple **Stock Service** to demonstrate the **Multicasting** using the **RoutingService** in this post. The **Stock Service** will send brief information of stocks (name, its type and its instantaneous price) to each subscribed receivers simultaneously to all subscribed receivers after each pre-defined time span.



Multicasting

Below is the class diagram of the service-



I've created a **StockService** using **One-Way messages**. Below are the definitions of **DataContract** & **ServiceContract** respectively along with the implementation details of the **StockService**-

```

[DataContract]
public class Stock
{
    [DataMember]
    public string Name;

    [DataMember]
    public string StockType;

    [DataMember]
    public double Price;
}

[ServiceContract]
public interface IStockService
{
    [OperationContract(IsOneWay = true)]
    void SendStockDetail(Stock stock);
  
```

```

    }

    public class StockService : IStockService
    {
        public void SendStockDetail(Stock stock)
        {
            Console.WriteLine(string.Format("Stock Name: {0}, Stock Type: {1}, Price: ${2:000.00}",
            stock.Name, stock.StockType, stock.Price));
        }
    }
}

```

Finally I've hosted **StockService** in a console application and configured a single endpoint along with a standard **mex** endpoint as shown below-

```

<services>
  <service name="StockInformation.StockService">
    <endpoint address="" binding="basicHttpBinding" contract="StockInformation.IStockService" />
    <endpoint address="mex" kind="mexEndpoint" />
    <host>
      <baseAddresses>
        <add baseAddress="http://localhost:8081/StockService" />
      </baseAddresses>
    </host>
  </service>
</services>

```

Please note that each instance of this service will act as a receiver in our **Multicasting** demo. Let's move to the next section to configure the **RoutingService** for the **Multicasting**.

Configuring the RoutingService for Multicasting

I'll re-configure our **RoutingService** used in '**Service Versioning**' demo earlier in this post for the **Multicasting** demo. So first I've re-configured the **RoutingService** with the following virtual endpoint-

```

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

```

Notice that I've used here **ISimplexDatagramRouter ServiceContract** as **StockService** is designed to process only **One-Way messages**. Next I've re-defined following three target endpoints one for each service instance. These instances will act as receivers in our **Multicasting** demo. Actually here I've just subscribed receivers in order to receive multicast information from the sender(s).

```

<client>
  <endpoint address="http://localhost:8081/StockService1" binding="basicHttpBinding"
    contract="*" name="stockservice1" />
  <endpoint address="http://localhost:8081/StockService2" binding="basicHttpBinding"

```

```

        contract="*" name="stockservice2" />
    <endpoint address="http://localhost:8081/StockService3" binding="basicHttpBinding"
        contract="*" name="stockservice3" />
</client>

```

Next I've re-defined a single filter using the **MatchAll** filter type that will matches all incoming messages.

```

<filters>
    <filter name="wildCardFilter" filterType="MatchAll" />
</filters>

```

As our goal is to just multicast all incoming messages from the sender(s) to the subscribed receiver(s), so next I've re-configured the filter table '**RoutingTable**' by mapping the above defined **wildCardFilter** filter to all target endpoints (receivers) as shown down below.

```

<filterTables>
    <filterTable name="RoutingTable">
        <add filterName="wildCardFilter" endpointName="stockservice1" />
        <add filterName="wildCardFilter" endpointName="stockservice2" />
        <add filterName="wildCardFilter" endpointName="stockservice3" />
    </filterTable>
</filterTables>

```

This finishes the **RoutingService** configuration for **Multicasting** demo. Let's move to the next section in order to mimic the sender(s) for the **Multicasting**.

Mimicking the Sender for Multicasting

The next step would be to mimic the sender(s) application for our **Multicasting** demo in order to send stock information after each pre-defined time span to the subscribed receivers. For this I've created a console client application of the **StockService** and configured it with following endpoint-

```

<client>
    <endpoint address="http://localhost:8080/RoutingService/Router" binding="basicHttpBinding"
        contract="Sender.IStockService" name="stockDetails" />
</client>

```

Finally let's go through the client application code.

```

var cf = new ChannelFactory<IStockService>("stockDetails");
var channel = cf.CreateChannel();

Console.WriteLine("*** Multicasting using RoutingService Demo ***\n");
Console.WriteLine("Please hit any key to start: ");
string command = Console.ReadLine();

while (command != "exit")
{
    var stock = GetStockDetails();
    channel.SendStockDetail(stock);

    Console.WriteLine("Details of the Stock: {0} has been sent; Sender: {1}", stock.Name);
    System.Threading.Thread.Sleep(3000);
}

```

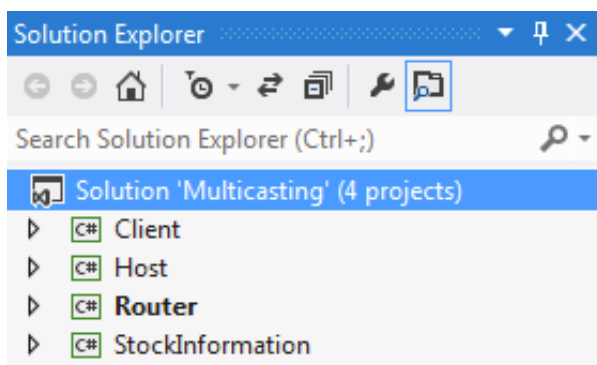


```
((IClientChannel)channel).Close();
```

In the above code, first I've created a client side proxy at runtime using the **ChannelFactory** class in order to invoke the only member '**SendStockDetail**' of the **StockService**. Next I've generated the stock information using the **GetStockDetails** method and multicasted it by invoking the **SendStockDetail** method. Note that I've simulated the stock information generation after each 3 seconds time span using the **GetStockDetails** method. You can find the code of the same in the sample example provided with this post.

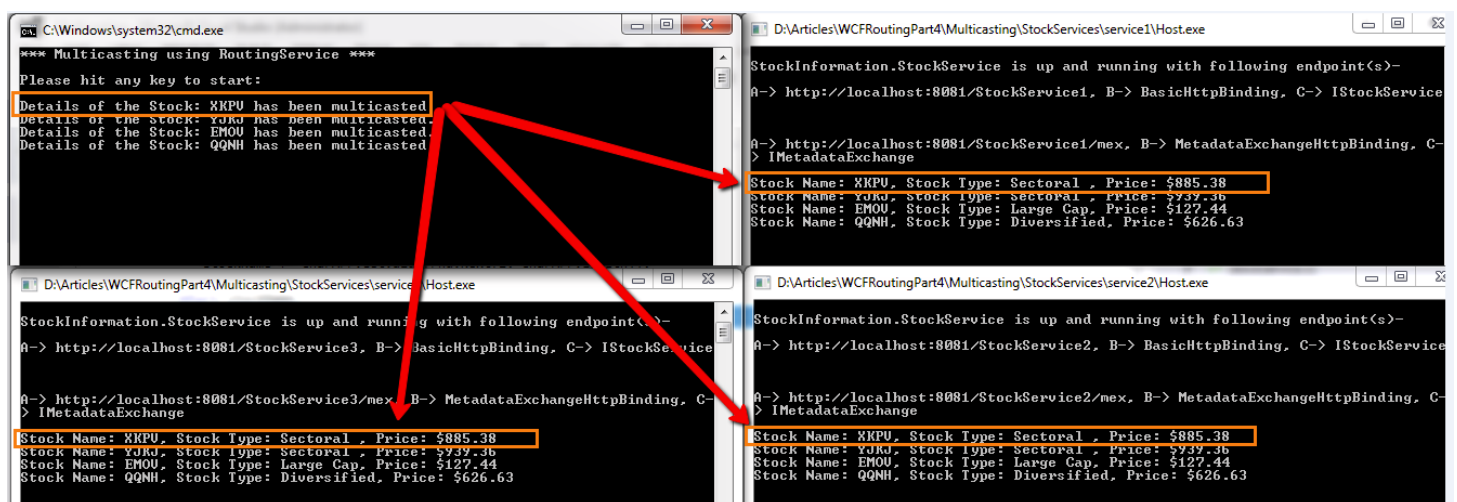
Simulating the Multicasting using the RoutingService

Before running our demo, just go through the screen shot of the **Multicasting** solution provided with this post-



Now just set **Client** & **Router** 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 **WCFRoutingPart4\Multicasting\StockServices\StartAllServices.cmd** file from the **Visual Studio Developer Command Prompt** (in **Administrator** mode) in order to start **StockService1**, **StockService2** & **StockService3** services.

Next press any key on the console client window (sender); you can verify that all incoming messages are being multicasted to all subscribed receivers by the intermediary **RoutingService**.



Next just stop any one receiver; you can verify that now incoming messages are being multicasted to the available (active) receivers by the intermediary **RoutingService**.

```

C:\Windows\system32\cmd.exe
*** Multicasting using RoutingService ***
Please hit any key to start:
Details of the Stock: XKPU has been multicast.
Details of the Stock: YJKJ has been multicast.
Details of the Stock: EMOU has been multicast.
Details of the Stock: QQNH has been multicast.
Details of the Stock: USRT has been multicast.
Details of the Stock: MNEB has been multicast.
Details of the Stock: FUHY has been multicast.
Details of the Stock: UPTG has been multicast.
Details of the Stock: NJFM has been multicast.
Details of the Stock: NUWE has been multicast.
Details of the Stock: FQIL has been multicast.
Details of the Stock: KSNX has been multicast.

D:\Articles\WCFRoutingPart4\Multicasting\StockServices\service1\Host.exe
StockInformation.StockService is up and running with following endpoint(s)-
A-> http://localhost:8081/StockService1, B-> BasicHttpBinding, C-> IStockService
A-> http://localhost:8081/StockService1/mex, B-> MetadataExchangeHttpBinding, C-> IMetadataExchange
Stock Name: XKPU, Stock Type: Sectoral , Price: $885.38
Stock Name: YJKJ, Stock Type: Sectoral , Price: $939.36
Stock Name: EMOU, Stock Type: Large Cap, Price: $127.44
Stock Name: QQNH, Stock Type: Diversified, Price: $626.63
Stock Name: USRT, Stock Type: Index, Price: $812.71
Stock Name: MNEB, Stock Type: Small Cap, Price: $477.50
Stock Name: EIQH, Stock Type: Large Cap, Price: $141.29
Stock Name: FUHY, Stock Type: Large Cap, Price: $164.77
Stock Name: UPTG, Stock Type: Index, Price: $826.56
Stock Name: NJFM, Stock Type: Small Cap, Price: $491.35
Stock Name: NUWE, Stock Type: Diversified, Price: $513.84
Stock Name: FQIL, Stock Type: Mid Cap, Price: $178.62
Stock Name: KSNX, Stock Type: Small Cap, Price: $364.71

D:\Articles\WCFRoutingPart4\Multicasting\StockServices\service2\Host.exe
StockInformation.StockService is up and running with following endpoint(s)-
A-> http://localhost:8081/StockService2, B-> BasicHttpBinding, C-> IStockService
A-> http://localhost:8081/StockService2/mex, B-> MetadataExchangeHttpBinding, C-> IMetadataExchange
Stock Name: XKPU, Stock Type: Sectoral , Price: $885.38
Stock Name: YJKJ, Stock Type: Sectoral , Price: $939.36
Stock Name: EMOU, Stock Type: Large Cap, Price: $127.44
Stock Name: QQNH, Stock Type: Diversified, Price: $626.63
Stock Name: USRT, Stock Type: Index, Price: $812.71
Stock Name: MNEB, Stock Type: Small Cap, Price: $477.50
Stock Name: EIQH, Stock Type: Large Cap, Price: $141.29
Stock Name: FUHY, Stock Type: Large Cap, Price: $164.77
Stock Name: UPTG, Stock Type: Index, Price: $826.56
Stock Name: NJFM, Stock Type: Small Cap, Price: $491.35
Stock Name: NUWE, Stock Type: Diversified, Price: $513.84
Stock Name: FQIL, Stock Type: Mid Cap, Price: $178.62
Stock Name: KSNX, Stock Type: Small Cap, Price: $364.71

```

Next stop all active receivers; you can verify that incoming messages are still being multicast by the intermediary **RoutingService** although there is no receiver. This is as per rule of the multicasting where there might be no receivers.

```

C:\Windows\system32\cmd.exe
*** Multicasting using RoutingService ***
Please hit any key to start:
Details of the Stock: XKPU has been multicast.
Details of the Stock: YJKJ has been multicast.
Details of the Stock: EMOU has been multicast.
Details of the Stock: QQNH has been multicast.
Details of the Stock: USRT has been multicast.
Details of the Stock: MNEB has been multicast.
Details of the Stock: FUHY has been multicast.
Details of the Stock: UPTG has been multicast.
Details of the Stock: NJFM has been multicast.
Details of the Stock: NUWE has been multicast.
Details of the Stock: FQIL has been multicast.
Details of the Stock: KSNX has been multicast.
Details of the Stock: WXLJ has been multicast.
Details of the Stock: CZQU has been multicast.
Details of the Stock: SUCD has been multicast.
Details of the Stock: THTU has been multicast.
Details of the Stock: LBFB has been multicast.
Details of the Stock: CURI has been multicast.

```

Finally do one more experiment. Just start multiple instances of the client application (senders) as well as multiple instances of the **StockService** (receivers); you can verify that in this case incoming messages are being multicast from the multiple senders to all subscribed receivers by the

intermediary **RoutingService**. This is again as per rule of the multicasting where there might be more than one sender.

```

C:\Windows\system32\cmd.exe
*** Multicasting using RoutingService ***
Please hit any key to start:
Details of the Stock: CGGE has been multicasted.
Details of the Stock: DCCR has been multicasted.
Details of the Stock: UUNX has been multicasted.

D:\Articles\WCFRoutingPart4\Multicasting\Client\bin\Debug\Client.exe
*** Multicasting using RoutingService ***
Please hit any key to start:
Details of the Stock: OPST has been multicasted.
Details of the Stock: TEBS has been multicasted.
Details of the Stock: LYNZ has been multicasted.

D:\Articles\WCFRoutingPart4\Multicasting\StockServices\service1\Host.exe
A-> http://localhost:8081/StockService1/mex, B-> MetadataExchangeHttpBinding, C-
> IMetadataExchange
Stock Name: CGGE, Stock Type: Sectoral, Price: $046.07
Stock Name: OPST, Stock Type: Diversified, Price: $527.58
Stock Name: DCCR, Stock Type: Large Cap, Price: $100.05
Stock Name: TEBS, Stock Type: Index, Price: $744.15
Stock Name: UUNX, Stock Type: Index, Price: $762.84
Stock Name: LYNZ, Stock Type: Small Cap, Price: $408.94

D:\Articles\WCFRoutingPart4\Multicasting\StockServices\service2\Host.exe
A-> http://localhost:8081/StockService2/mex, B-> MetadataExchangeHttpBinding, C-
> IMetadataExchange
Stock Name: CGGE, Stock Type: Sectoral, Price: $046.07
Stock Name: OPST, Stock Type: Diversified, Price: $527.58
Stock Name: DCCR, Stock Type: Large Cap, Price: $100.05
Stock Name: TEBS, Stock Type: Index, Price: $744.15
Stock Name: UUNX, Stock Type: Index, Price: $762.84
Stock Name: LYNZ, Stock Type: Small Cap, Price: $408.94

D:\Articles\WCFRoutingPart4\Multicasting\StockServices\service3\Host.exe
A-> http://localhost:8081/StockService3/mex, B-> MetadataExchangeHttpBinding, C-
> IMetadataExchange
Stock Name: CGGE, Stock Type: Sectoral, Price: $046.07
Stock Name: OPST, Stock Type: Diversified, Price: $527.58
Stock Name: DCCR, Stock Type: Large Cap, Price: $100.05
Stock Name: TEBS, Stock Type: Index, Price: $744.15
Stock Name: UUNX, Stock Type: Index, Price: $762.84
Stock Name: LYNZ, Stock Type: Small Cap, Price: $408.94

```

Conclusion

Actually I've considered **Service Contract Versioning** as a **Service Versioning** scenario in this post. But there might be other scenarios of the **Service Versioning** like: **Data Contract Versioning**, **Message Contract Versioning**, **Address & Binding Versioning** etc. You'll need to outline and handle each cases of the **Service Versioning** scenarios before implementing the same.

As far as **Multicasting** is concerned, apart from just **Multicasting** the incoming messages to the subscribed receivers, you can also portioned receivers into the groups and can multicast incoming messages to a specific group based on some defined criteria. Let's say there are two groups of subscribed receivers of the **StockService**: G1 & G2, then we can multicast incoming messages with **Price > \$500** to the group G1 and incoming messages with **Price <= \$500** to the group G2 using the **RoutingService**.

So you have seen how **Service Versioning** & **Multicasting** features can easily be implemented using the **RoutingService**. Hope you've enjoyed this post.

References

- Service Versioning
 - <http://msdn.microsoft.com/en-us/library/ms731060%28v=vs.110%29.aspx>
 - <http://msdn.microsoft.com/en-us/library/ee517423%28v=vs.110%29.aspx>
 - <http://msdn.microsoft.com/en-us/library/ee816862%28v=vs.110%29.aspx>
 - <http://msdn.microsoft.com/en-us/library/bb491124.aspx>

- Multicasting
 - <http://en.wikipedia.org/wiki/Multicast>
 - <http://www.techterms.com/definition/multicasting>

History

- 7th Jun, 2014 -- Original version posted

License


This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

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 in 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 diploma (**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.**