Microsoft Robotics Developer Studio 2008 Express Edition *Send feedback on this topic*

Service Tutorial 5 (C#) - Subscribing

Writing an application using the Decentralized Software Services (DSS) is a simple matter of orchestrating the input and output between a set of services. Services represent the interface to software or hardware and allow you to communicate between processes that perform specific functions.

This tutorial is provided in the C# language. You can find the project files for this tutorial at the following location under the installation folder:

Sample location
```
  Samples\ServiceTutorials\Tutorial5\CSharp
```

This tutorial teaches you how to:

- Add a Reference to a Service Proxy.
- Subscribe to a Partner Service.
- Handle Notifications.
- Synchronize State.

The service written in Service Tutorial 4 will be modified to support subscriptions.

Prerequisites

This tutorial uses the service written in Service Tutorial 1 (C#) - Creating a Service. We will extend it to subscribe to the service written in Service Tutorial 4 (C#) - Supporting Subscriptions. The service you will create is referred to as **ServiceTutorial5**. Feel free to start a new project or keep working in the **ServiceTutorial1** project, keeping in mind that some names may differ.

Hardware
This tutorial requires no special hardware.

Software
This tutorial is designed for use with Microsoft Visual C#. You can use:

- Microsoft Visual C# Express Edition
- Microsoft Visual Studio Standard, Professional, or Team Edition.

You will also need Microsoft Internet Explorer or another conventional web browser.

Step 1: Add a Reference to a Service Proxy

You need to establish a partner relationship with the service from **ServiceTutorial4** and then subscribe to that service. The first stage in this is to create a reference to the partner service. To do this, you need to understand a little bit about proxies.

When a service is built, three assemblies are created.

- The main service (or implementation) assembly
  For example, **ServiceTutorial5.Y2006.M06.dll** -- contains the functionality of the service.
- The proxy assembly
  For example, **ServiceTutorial5.Y2006.M06.proxy.dll** -- contains stubs for all the public interfaces of the service.
- The transform assembly
  For example, **ServiceTutorial5.Y2006.M06.transform.dll** -- contains code that allows types to be converted between the proxy and the service assemblies.

To send messages to another service, reference the service's proxy assembly rather than the implementation assembly. This is done for a number of reasons: if you are communicating with a service on another node, you may not have a copy of the implementation assembly on the local machine; also, by using proxies, the partner service does not have any code that executes within the context of your service.

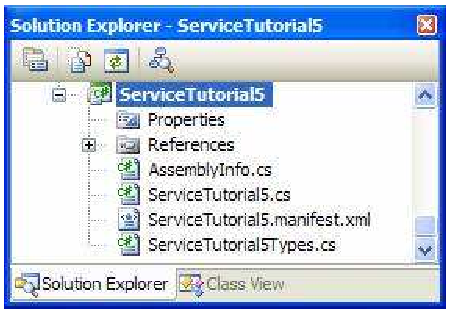
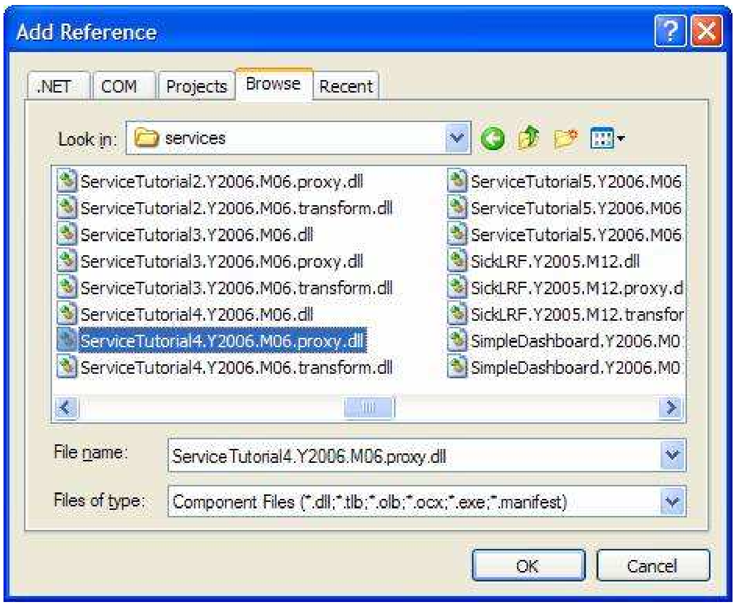


Figure 1 - Add a reference to the proxy assembly from **Solution Explorer**.

To use **ServiceTutorial4** as a partner, you need to add a reference to the assembly (**ServiceTutorial4.Y2006.M06.proxy.dll**) to the project. In Microsoft Visual Studio, you do this by selecting the project and then selecting the **Project** > **Add Reference...** menu command, or by right clicking on the project in the Solution Explorer and selecting the **Add Reference...** command from the popup menu.

This displays the **Add Reference** dialog box. Choose the **Browse** tab and navigate to the **bin** directory under the DSS installation directory. Select the file, **ServiceTutorial4.Y2006.M06.proxy.dll**.

The **ServiceTutorial4.Y2006.M06.proxy.dll** file is the proxy for the **ServiceTutorial4** project that comes with DSS. If you are walking through these tutorials and have created a separate project using **DssnewService** tool, then the name of the proxy for your project will be different.



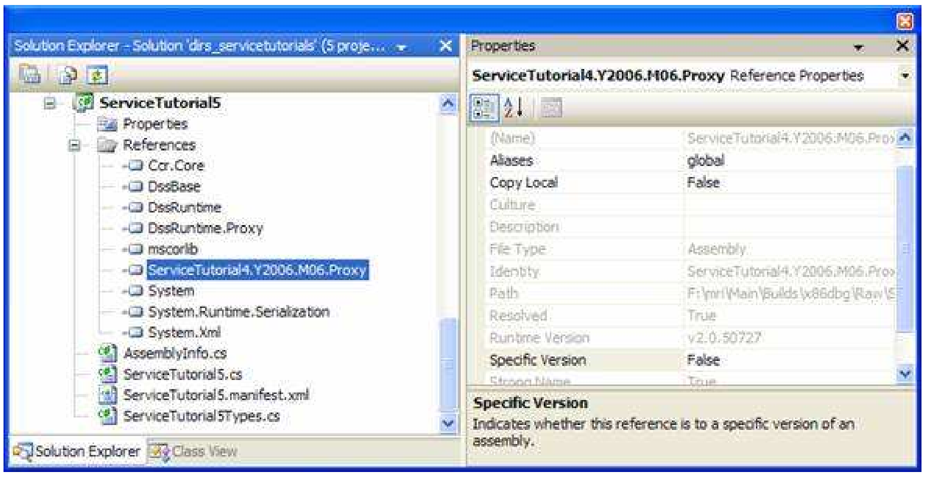
Look in: services

ServiceTutorial2.Y2006.M06.proxy.dll
ServiceTutorial2.Y2006.M06.transform.dll
ServiceTutorial3.Y2006.M06.dll
ServiceTutorial3.Y2006.M06.proxy.dll
ServiceTutorial3.Y2006.M06.transform.dll
ServiceTutorial4.Y2006.M06.dll
ServiceTutorial4.Y2006.M06.proxy.dll
ServiceTutorial4.Y2006.M06.transform.dll
ServiceTutorial5.Y2006.M06
ServiceTutorial5.Y2006.M06
ServiceTutorial5.Y2006.M06
SickLRF.Y2005.M12.dll
SickLRF.Y2005.M12.proxy.d
SickLRF.Y2005.M12.transfor
SimpleDashboard.Y2006.M0
SimpleDashboard.Y2006.M0

File name: ServiceTutorial4.Y2006.M06.proxy.dll
Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

Figure 2 - Browse to the proxy assembly in the **bin** directory.

Select the referenced assembly from the **References** list in the Solution Explorer and set the **Copy Local** and **Specific Version** fields to **False**.


Solution Explorer - Solution 'dirs_servicetutorials' (5 proje...
Properties
Solution Explorer
Class View

ServiceTutorial5
Properties
References
Ccr.Core
DssBase
DssRuntime
DssRuntime.Proxy
mscorlib
ServiceTutorial4.Y2006.M06.Proxy
System
System.Runtime.Serialization
System.Xml
AssemblyInfo.cs
ServiceTutorial5.cs
ServiceTutorial5.manifest.xml
ServiceTutorial5Types.cs

ServiceTutorial4.Y2006.M06.Proxy Reference Properties

(Name)	ServiceTutorial4.Y2006.M06.Pro
Aliases	global
Copy Local	False
Culture	
Description	
File Type	Assembly
Identity	ServiceTutorial4.Y2006.M06.Pro
Path	F:\mri\Main\Builds\x86dbg\Raw\S
Resolved	True
Runtime Version	v2.0.50727
Specific Version	False
Strong Name	True

Specific Version
Indicates whether this reference is to a specific version of an assembly.

Figure 3 - Set the reference properties for the proxy.

Use the same method to add a reference to **Microsoft.Dss.Runtime.Proxy.dll**. This DLL is also found in the **bin** directory.

Step 2: Subscribe to a Partner Service

Make the following changes in the **ServiceTutorial5.cs**.

Add an alias for the Service Tutorial 4 proxy. Proxies derive their namespace from the service **namespace** in the implementation assembly. The suffix **Proxy** is added to that namespace and assigned to the proxy.

```
using rst4 = Robotics.ServiceTutorial4.Proxy
```

Add a partner using the same method used to add the subscription manager partner in Service Tutorial 4. You also need to add the field **_clockNotify** of the type **rst4.ServiceTutorial4Operations**, the type of the operations port of the service created in Service Tutorial 4. This field is the port on which this service receives notifications when you have created the subscription.

```
// Partner with ServiceTutorial4 and refer to it by the name Clock
[Partner("Clock", Contract = rst4.Contract.Identifier, CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
rst4.ServiceTutorial4Operations _clockPort = new rst4.ServiceTutorial4Operations();
rst4.ServiceTutorial4Operations _clockNotify = new rst4.ServiceTutorial4Operations();
```

All that remains now is to subscribe to the partner service.

As you will recall from Service Tutorial 4 (C#) - Supporting Subscriptions, the service created there supported the message, **Subscribe**. When the proxy is generated, a utility function is generated for each message supported. In this case, the service calls the utility function for the **Subscribe** message. The single parameter for this overload is the notification port to which the subscription should send notifications.

Add the following code at the end of the **Start** method.

```
_clockPort.Subscribe(_clockNotify);
```

If we build and run this service, we see the following messages in the **http://localhost:50000/console/output** log.

```
.
.
.
Starting manifest load: file:///C:/MSRS/samples/config/ServiceTutorial5.manifest.xml
Manifest load complete
Service uri: http://localhost:50000/servicetutorial5
Service uri: http://localhost:50000/servicetutorial4
Subscribe request from: http://localhost:50000/servicetutorial5/NotificationTarget/7cf955e7-73ff-47cd-bdc0-d3a3a8251c49
```

```
Tick: 21
Tick: 22
Tick: 23
Tick: 24
```

Two things to note in this log: firstly, both the **ServiceTutorial4** and **ServiceTutorial5** services are started; and secondly, the log message noting the subscribe request being handled. The Uniform Resource Identifier (URI) passed here is the address of the notification port. This is used internally within the DSS infrastructure to route notifications to the port, **_clockNotify**.

Step 3: Handle Notifications

The purpose of subscribing to a service is to handle the notifications sent by that service. You will recall that the **ServiceTutorial4** service sends notifications for the **IncrementTick** and **Replace** messages. That being the case, the next step in this tutorial is to handle those notifications.

Begin by adding handlers for those notifications. Because the notifications are not being sent to the main service port of this service, you cannot use the **ServiceHandler** attribute. Instead you need to activate a receiver task for each notification message that you will handle. This is done in the **Start** method.

```
protected override void Start()
{
    base.Start();

    // Add the handlers for notifications from ServiceTutorial4.
    // This is necessary because these handlers do not handle
    // operations in this service, so you cannot mark them with
    // the ServiceHandler attribute.
    MainPortInterleave.CombineWith(
        new Interleave(
            new TeardownReceiverGroup(),
            new ExclusiveReceiverGroup(),
            new ConcurrentReceiverGroup(
                Arbiter.Receive<rst4.IncrementTick>(true, _clockNotify, NotifyTickHandler),
                Arbiter.Receive<rst4.Replace>(true, _clockNotify, NotifyReplaceHandler)
            ))
    );

    _clockPort.Subscribe(_clockNotify);
}
```

As you can see, the two receivers specify:

- the message type that they handle:
  - rst4.IncrementTick and
  - rst4.Replace
- the port on which the messages arrive:
  - _clockNotify
- and the handler method for the message.

Next, implement handlers for each of these message types.

In the main service implementation class, **ServiceTutorial5**, add the following methods.

```
private void NotifyTickHandler(rst4.IncrementTick incrementTick)
{
    LogInfo("Got Tick");
}

private void NotifyReplaceHandler(rst4.Replace replace)
{
    LogInfo("Tick Count: " + replace.Body.Ticks);
}
```

Neither of these handlers sends a response to the message it handles. They handle notification messages which do not expect a response.

Now when you run this service, you see a **Got Tick** message on the console following every **Tick: [tick number]** message and a single **Tick Count: [tick number]** message following the **Subscribe request from: [notification port]**.

Step 4: Synchronize State

The **ServiceTutorial5** service now has all the information required to keep its state synchronized with its partner from Service Tutorial 4 (C#) - Supporting Subscriptions.

Begin by making some changes in **ServiceTutorial5Types.cs**.

Change the state of the service so that it contains the **Tick** count. To keep this service distinct from Service Tutorial 4, delete the **Member** property and create an integer property named **TickCount**.

```
private int _tickCount;

[DataMember]
public int TickCount
{
    get { return _tickCount; }
    set { _tickCount = value; }
}
```

Now, change the operations port to support:

- the **IncrementTick** message defined in **ServiceTutorial4**
- and a new message for this service, **SetTickCount**.

```
/// <summary>
/// ServiceTutorial5 Main Operations Port
/// </summary>
[ServicePort]
public class ServiceTutorial5Operations : PortSet<
    DsspDefaultLookup,
    DsspDefaultDrop,
    Get,
    HttpGet,
    Replace,
    rst4.IncrementTick,
    SetTickCount>
{
}
```

Add a **using** directive to the file, **ServiceTutorial5Types.cs**, the same way you did in the first code snippet of Step 2.

The message, **SetTickCount**, and its body type, **SetTickCountRequest**, are declared in **ServiceTutorial5Types.cs** as shown in the following code snippet.

```
public class SetTickCount : Update<SetTickCountRequest, PortSet<DefaultUpdateResponseType, Fault>>
{
    public SetTickCount()
    {
    }

    public SetTickCount(int tickCount)
        : base(new SetTickCountRequest(tickCount))
    {
    }
}

[DataContract]
[DataMemberConstructor]
public class SetTickCountRequest
{
    public SetTickCountRequest()
    {
    }

    public SetTickCountRequest(int tickCount)
    {
        _tickCount = tickCount;
    }

    private int _tickCount;

    [DataMember]
    public int TickCount
    {
        get { return _tickCount;}
        set { _tickCount = value;}
    }
}
```

The **[DataMemberConstructor]** attribute when combined with a **[DataContract]** specifies that in the generated proxy an initializer constructor should be created containing each of the **[DataMember]** members of that class. In other words, there will be a constructor **SetTickCountRequest(int tickCount)** included in the proxy.

If you remove the **[DataMemberConstructor]** from the **SetTickCountRequest** class in this example, then only the default constructor **SetTickCountRequest()** (which takes no parameters) will be created in the proxy class. This is an important point because declaring overloaded constructors in your code, as shown above, does not make them carry through to the proxy.

Add handlers for these two messages in the service implementation class in **ServiceTutorial5.cs**. Both of these messages are marked as exclusive because they modify state.

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> IncrementTickHandler(rst4.IncrementTick incrementTick)
{
    _state.TickCount++;
    incrementTick.ResponsePort.Post(DefaultUpdateResponseType.Instance);
    yield break;
}

[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> SetTickCountHandler(SetTickCount setTickCount)
{
    _state.TickCount = setTickCount.Body.TickCount;
    setTickCount.ResponsePort.Post(DefaultUpdateResponseType.Instance);
    yield break;
}
```

Finally, connect the notification handlers to these message handlers by sending appropriate messages to the main service port from the notification handlers.

The **NotifyTickHandler** and **NotifyReplaceHandler** methods should look like this:

```
private void NotifyTickHandler(rst4.IncrementTick incrementTick)
{
    LogInfo("Got Tick");
    _mainPort.Post(new rst4.IncrementTick(incrementTick.Body));
}

private void NotifyReplaceHandler(rst4.Replace replace)
{
    LogInfo("Tick Count: " + replace.Body.Ticks);
    _mainPort.Post(new SetTickCount(replace.Body.Ticks));
}
```

In the **NotifyTickHandler** method, a message of type **IncrementTick** is handled and a message of type **IncrementTick** is sent to the main service port. Why is the **IncrementTick** argument not just sent on to the main port directly? A new message needs to be created to ensure that the **ResponsePort** is set up correctly.

In the **NotifyTickHandler** method, an **IncrementTick** message is handled and an **IncrementTick** message is sent to the main port. In the **NotifyReplaceHandler** method, an **rst4.Replace** message is handled and a **SetTickCount** message is sent to the main port. Why is a new message type needed? The **rst4.Replace** message uses the DSSP action verb **Replace**. This is appropriate in **ServiceTutorial4**, where it replaces the state of the service, but is inappropriate in **ServiceTutorial5** where, conceptually, it represents a change to one member of the state. For this reason, the **SetTickCount** message is defined using the **Update** verb and used to modify the state.

Also, the design could have used a different notification message type to convey the current tick count instead of **IncrementTick** which just triggers an increment. This implementation is "free running", i.e. after an initial **Replace** message to establish the starting value, all future notifications just tell ServiceTutorial5 to increment its own counter. It would be more robust to send the current tick count with every notification, but that would require an additional message type because **IncrementTick** cannot contain the tick count -- it is merely a trigger message fired off by a timer.

Summary

In this tutorial, you learned how to:

- Add a Reference to a Service Proxy.
- Subscribe to a Partner Service.
- Handle Notifications.
- Synchronize State.