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

Service Tutorial 4 (C#) - Supporting Subscriptions

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\Tutorial4\CSharp
```

This tutorial teaches you how to:

- Support a Subscribe Message.
- Use the Subscription Manager.
- Maintain the Subscriber List.
- Send Notifications.
- Synchronize With a New Subscriber.

See Also:

- Appendix A: Subscription Manager Message Flow

The service written in Service Tutorial 3 will be modified to support subscriptions.

Prerequisites

This tutorial uses the service written in Service Tutorial 3 (C#) - Persisting State. The service you will create is referred to as **ServiceTutorial4**. Feel free to start a new project or keep working in the previous 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: Support a Subscribe Message

The first part of supporting subscriptions is to support a **Subscribe** message.

In the file, **ServiceTutorial4Types.cs**, add the definition of a **Subscribe** message and include it in the list of messages supported by the service. The classes **SubscribeRequestType** and **SubscribeResponseType** are standard types defined in the **Microsoft.Dss.ServiceModel.Dssp** namespace.

The definition of **Subscribe** message:

```
public class Subscribe : Subscribe<SubscribeRequestType, PortSet<SubscribeResponseType, Fault>>
{
}
```

After including the **Subscribe** message to the service's operaions port, it should look like this:

```
/// <summary>
/// ServiceTutorial4 Main Operations Port
/// </summary>
[ServicePort]
public class ServiceTutorial4Operations : PortSet<
    DsspDefaultLookup,
    DsspDefaultDrop,
    Get,
    HttpGet,
    Replace,
    IncrementTick,
    Subscribe>
{
}
```

In order for the code to be more readable we have expanded the operations port into multiple lines.

In the file, **ServiceTutorial4.cs**, add the **SubscribeHandler** method to the service implementation class.

This handler currently logs the subscriber's address and nothing else. The next step illustrates how to maintain the list of subscribers.

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> SubscribeHandler(Subscribe subscribe)
{
    SubscribeRequestType request = subscribe.Body;
    LogInfo("Subscribe request from: " + request.Subscriber);

    yield break;
}
```

The sample file, ServiceTutorial4.cs, is a modified version of Service Tutorial 3 (C#). The lines of code pertaining to the timer port used in Service Tutorial 3 were removed and replaced with the new "Interval=" **ServiceHandler** Attribute. This attribute schedules a periodic timer event to invoke the handler.

See example below from ServiceTutorial4.cs:

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive, Interval=1)]
public IEnumerator&lt;ITask&gt; IncrementTickHandler(IncrementTick incrementTick)
```

Step 2: Use the Subscription Manager

The next task for supporting subscriptions is maintaining a list of subscribers. Because supporting subscriptions is a common function for a service, there is a standard DSS component (the **SubscriptionManager**) which we use to manage our subscribers. So take a brief detour and start the **SubscriptionManager** as a Partner service.

Open the file **ServiceTutorial4.cs** to add the **SubscriptionManager**. Because the **SubscriptionManager** is a standard service, we do not need to add another reference to the project. To help minimize typing, we add a **using** line with an alias--in this case, **submgr**. It is common to use aliases like this when using other services because services frequently have common names for their classes. For example, all classes have a **Contract** class, most have a **Get** class, etc. You will see this in Service Tutorial 5 (C#) - Subscribing.

```
using submgr = Microsoft.Dss.Services.SubscriptionManager;
```

Add code to declare a partner service in the service implementation class.

The **Partner** attribute specifies:

- a unique name for the partner,
- the Contract of the service to use as a partner, and
- a creation policy.

For this tutorial, we will create a partner (named **SubMgr**) whose Contract is **http://schemas.microsoft.com/xw/2005/01/subscriptionmanager.html**. When the service is started, it creates a new instance of this partner. If that creation fails, the service creation also fails and the **Start** method will not be called.

Add the following definition to the service implementation class after the line that defines **_mainPort**:

```
// Declare and create a Subscription Manager to handle the subscriptions
[SubscriptionManagerPartner]
private submgr.SubscriptionManagerPort _submgrPort = new submgr.SubscriptionManagerPort();
```

Here, the **Partner** attribute is applied to the **_submgrPort** field. This **_submgrPort** field is declared as the SubscriptionManager's main service port. Any message posted to this port will be forwarded to the SubscriptionManager instance created for this service.

Step 3: Maintain the Subscriber List

Now that you have a **SubscriptionManager** as a partner you can send messages to it. Begin by maintaining the list of subscribers. **DsspServiceBase** provides a method to do exactly what you need.

Add the following code in **SubscribeHandler** just before **yield break**.

```
SubscribeHelper(_submgrPort, request, subscribe.ResponsePort);
```

This method sends an **Insert** message to the SubscriptionManager. The SubscriptionManager then assumes responsibility for managing the list of subscriptions.

Step 4: Send Notifications

The final part of supporting subscriptions is sending notifications to subscribers. Because this service uses the **SubscriptionManager**, all you need to do is post state changes to the SubscriptionManager. SubscriptionManager sends notifications on to the subscribed partners.

There are two places where the state changes, the **IncrementTickHandler** and the **ReplaceHandler** methods.

When the state is entirely replaced, send a notification to our subscribers with the new service state. **DsspServiceBase** provides a helper function, **SendNotification**, that internally sends a **Submit** message to the **SubscriptionManager** with the body of the message to send to the subscribers and the action verb (in this case

ReplaceRequest).

Add the following code to the **ReplaceHandler** after the line that modifies **_state**:

```
// Echo the Replace message to all subscribers
base.SendNotification(_submgrPort, replace);
```

Likewise, when the state is updated by the service handling an **IncrementTick** message, send a notification of this event to the subscribers.

Add the following code to the **IncrementTickHandler** after the call to **LogInfo**:

```
// Notify all subscribers.
// The incrementTick message does not contain the tick count.
base.SendNotification(_submgrPort, incrementTick);
```

The subscription model has a symmetry. The service can notify its subscribers with any state changing message that it can itself process. (In other words, a notification message must be one of the messages listed in the Operations Port). So in the case of this tutorial service, the only two messages that modify the service state are **Replace** and **IncrementTick**. Service Tutorial 5 (C#) - Subscribing shows you how to subscribe to this service.

Step 5: Synchronize With a New Subscriber

The **IncrementTick** update does not contain the current tick count, merely carrying the semantic content that the **Tick** count should be incremented by one. Because of that, a subscriber does not know the tick count, only the number of ticks that have occurred since the subscription started. If you want, you can send the current tick count by sending a **Replace** notification to new subscribers.

Begin by changing the **SubscribeHandler** method to detect when a new subscriber is successfully added.

The **SubscribeHelper** method works by sending an **Insert** message to the SubscriptionManager service. This is an asynchronous operation.

SubscribeHelper returns a **SuccessFailurePort**. This is a **PortSet** that can handle two message types: **SuccessResult** and **Exception**. To handle this result, we return a **Choice** task using **yield return**. That **Choice** task then takes two delegates: one which handles the **SuccessResult**, and one that handles an **Exception**.

When a **SuccessResult** message is returned, we know that the subscription has been successfully established and we send the current state of the service (as a **Replace** notification) to the subscriber. If an **Exception** message is returned, we log this error.

The code below uses the **SendNotificationToTarget** method, this is similar in functionality to the **SendNotification** method, but takes three parameters rather than the usual two. The first parameter is the address of the subscriber. This causes the subscription manager to send this notification only to the specified subscriber. Because there isn't a constructed **Replace** message available at this point in the code, this call uses an overload for **SendNotificationToTarget**. This is accomplished by taking the message body and using a generic type-parameter (in this case, **Replace**) to indicate which message type is being notified.

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> SubscribeHandler(Subscribe subscribe)
{
    SubscribeRequestType request = subscribe.Body;
    LogInfo("Subscribe request from: " + request.Subscriber);

    // Use the Subscription Manager to handle the subscribers
    yield return Arbiter.Choice(
        SubscribeHelper(_submgrPort, request, subscribe.ResponsePort),
        delegate(SuccessResult success)
        {
            // Send a notification on successful subscription so that the
            // subscriber can initialize its own state
            base.SendNotificationToTarget<Replace>(request.Subscriber, _submgrPort, _state);
        },
        delegate(Exception e)
        {
            LogError(null, "Subscribe failed", e);
        }
    );

    yield break;
}
```

Summary

In this tutorial, you learned how to:

- Support a Subscribe Message.
- Use the Subscription Manager.
- Maintain the Subscriber List.
- Send Notifications.
- Synchronize With a New Subscriber.

Appendix A: Subscription Manager Message Flow

The following sequence diagram shows the message flow when **ServiceTutorial5** subscribes to **ServiceTutorial4**.

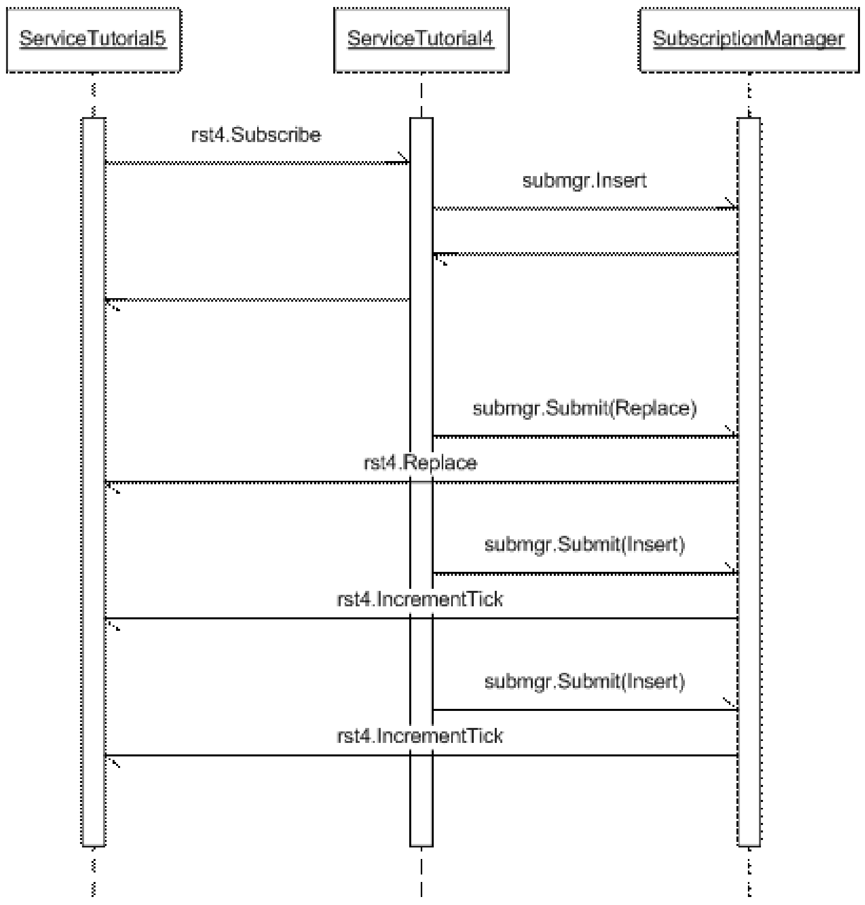

Figure 1 - Subscription message flow between two services.